Optimization: Algorithms, Complexity & Approximations

Anastasios Kyrillidis *

*Instructor, Computer Science at Rice University

Contributors: Nick Sapoval, Carlos Quintero Pena, Delaram Pirhayatifard, McKell Stauffer, Mohammad Taha Toghani, Senthil Rajasekaran, Gaurav Gupta, Pranay Mittal, Yi Lin, Shawn Fan

Chapter 11

This lecture considers distributed approaches in computing unconstrained optimization on very large datasets. We will discuss how to distribute optimization in large-scale settings, study synchrony vs. asynchrony in gradietn descent, provide rough theoretical results on how asynchrony affects performance, and take a look at alternatives and state of the art.

Distributed Gradient Descent

Recall: Stochastic gradient descent. (SGD) is used almost everywhere, both in training classical ML tasks (linear prediction, linear classification) and training modern ML tasks (non-linear classification, neural networks). Mathematically, it satisfies:

$$x_{t+1} = x_t - \eta \nabla f_{i_t} x_t$$

Intuitively, we select a training sample, compute the gradient on this sample, and update the model based on this gradient. We can observe these properties:

1. The current model x_t is used for the computation of $\nabla f_{i_t}(\cdot)$.
2. When we update the model, the state of the system is as when we read x_t
3. The whole process is sequential.

SGD as presented above operates on a single machine (single CPU, single memory, single communication bus line). Let us first observe where computation and communication happens in the SGD iteration.

1. Model x_t needs to be "transferred" to the location where computation of $\nabla f_{i_t}(\cdot)$ happens
2. Data point $f_{i_t}(\cdot)$ needs to be traansferred where the computation of $\nabla f_{i_t}(\cdot)$ happens
3. The update $x_t - \eta \nabla f_{i_t}(x_t)$ overwrites the current model

Note that SGD is not easily parallelizable on GPU due to its memory constraint; the model and data do not fit in the GPU memory.

Distributed computing. We can generally split distributed computing into two categories: **Single node distributed computing** and **Multi-node distributed computing**.

Single node distributed computing

1. Single machine, many cores (up to 100s)
2. Shared memory (all processors have access to it)
3. Communication to RAM is relatively cheap

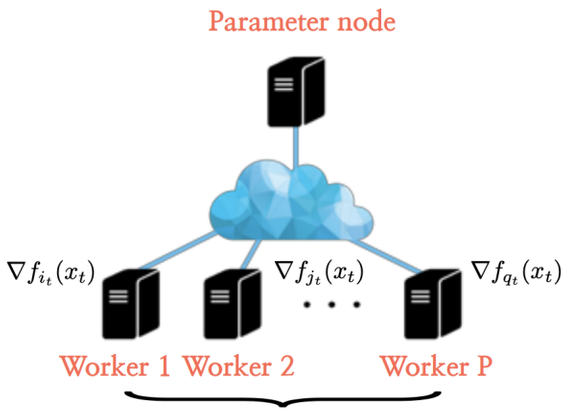
Multi-node distributed computing

1. Single machine, many cores (up to 100s)
2. Shared memory (all processors have access to it)
3. Communication to RAM is relatively cheap

It is not apparent how multi-node distributed computing could be utilized in stochastic gradient descent, but it is clear in the full gradient descent case:

$$x_{t+1} = x_t - \eta \sum_{i=1}^{n} \nabla f_i(x_t)$$

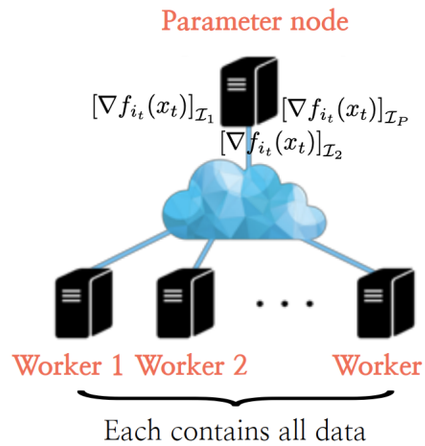
Data-parallel distribution. We can consider a data-parallel system where a parameter node keeps and distributes model x_t to worker nodes at every iteration, then each worker node computes a part of the full gradient ($\nabla f_i(\cdot)$) based on the data they have, and finally, the parameter node waits for all gradient parts to be collected to update the model. This is illustrated in the diagram below.



Notice that synchronization is required at the end of each iteration; the parameter node has to wait for the slowest worker node to finish its computations. While gradient descent neatly distributes in this fashion, this model would not work in an online learning setting where data samples arrive one at a time, or if we don't have access to all data for any reason. Even if there is finite and fixed data, full gradient descent suffers from generalization error – it does not perform well on unseen data. This is because gradient descent converges to the "first-seen" stationary point, causing it to overfit the landscape of the training data. Meanwhile, SGD explores the landscape before converging.

Coordinate-parallel distribution. Since SGD inherently performs computations on a much smaller subset of the data compared to full GD, distributing SGD computations makes more sense when we consider a scenario where even calculating $\nabla f_{i_t}(x_t) \in \mathbb{R}^p$ is expensive for a single node. We can therefore introduce coordinate-based parallelism. Just like in the data-parallel system, a parameter node keeps and distributes model x_t to worker nodes at every iteration. In addition, the parameter node also distributes indices corresponding to coordinates to worker nodes. The worker nodes then compute

part of the stochastic gradient based on the coordinates given by the parameter node. Finally, the parameter node waits for all gradient parts to be collected to update the model.

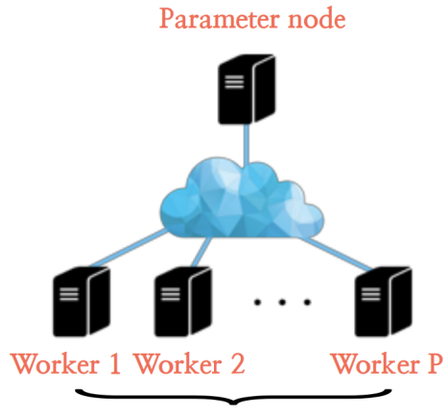


We observe that this model is related to coordinate descent algorithms. It is effective in a large-scale implementation, where just a part of the model is too large to be computed in a centralized fashion. However, in a smaller scale, it could be overkill to only compute updates for a small subset of entries.

Mini-batch SGD. The mini-batch SGD iteration can be mathematically described as

$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

This sits in between the previous two distributed computing schemes. As before, the parameter node keeps and distributes model x_t at every iteration. Worker nodes then compute their part of the mini-batch gradient. The parameter node waits for all gradient parts to be collected to do the mini-batch step.



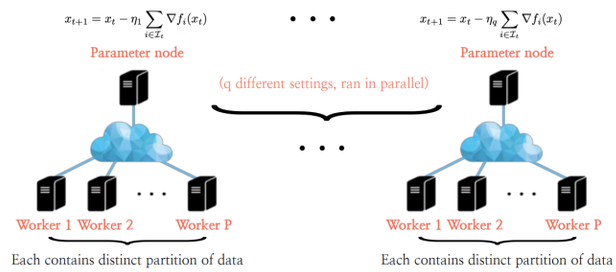
Like the previous schemes, this method still requires a synchronization step. Since each worker has less to do, and computing $\nabla f_{i_t}(x_t)$ is usually cheap per node, synchronization happens more often, thus introducing higher synchronization step overheads. This and highlights that there is a tradeoff between statistical efficiency, computations efficiency in terms of convergence, and communication efficiency.

Mini-batch SGD in parallel. Another scenario is to run mini-batch SGD in parallel and average combine the results at the end. Each worker node would independently run:

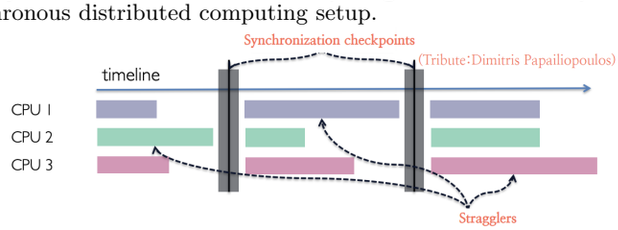
$$x_{t+1} = x_t - \eta \sum_{i \in \mathcal{I}_t} \nabla f_i(x_t)$$

The parameter node would be idle while workers nodes do mini=batch SGD as if there is no distributed computation. After collecting models from all workers, the parameter node averages the models. This facilitates minimal communication. Every node works on its own and only sends the model at the end of its execution. The final decision is prediction averaging – similar ideas hold for random forests. Designed for convex problems, the theoretical basis of this model is that each subproblem has a solution that is close to the global one, such that averaging models from independent mini-batch SGD executions does not hurt.

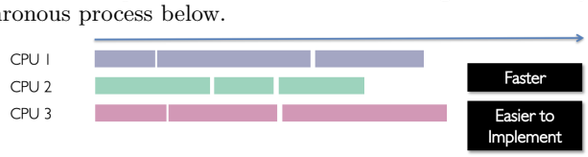
Running code in parallel for hyperparameter optimization. This is another way to use distributed computing in SGD. The code is run in parallel with different settings, without being aggregated in any way, to optimize hyperparameters.



Asynchronous distributed computing. Consider the synchronous distributed computing setup.

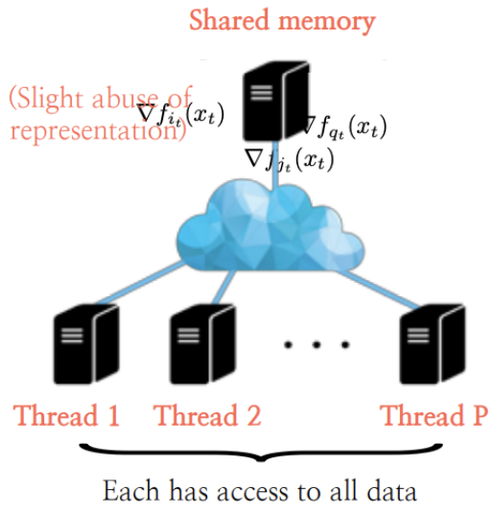


As shown in the figure above, there are synchronization checkpoints that wait for every worker node at the end of every iteration. This means the parameter node waits for the slowest worker before synchronizing all workers, and keeps all nodes aware of each other's updates to the model. This makes synchronization very expensive. Consider a setting with P workers. If P–1 of them have already sent their updates to the parameter server, the whole system needs to wait for the one straggler to complete before proceeding. This is apparent when we compare the video above to the figure of an asynchronous process below.



There are several ways to avoid this unnecessary overhead. Firstly, multicore systems can host lar-ge scale problems. After preprocessing, large-scale problems may involve only a few terrabytes of data. Thus, instead of running SGD across clusters of processing nodes, one can use a single inexpensive mul-

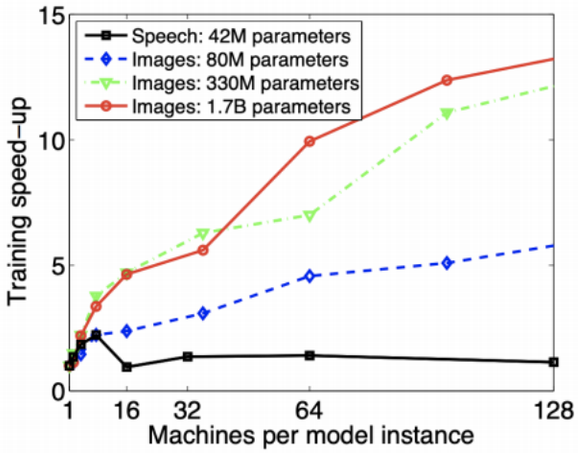
ticore work station. This model boasts advantages such as low latency and high throughput shared high memory, high bandwidth of multiple disks, and fast multithread processors. However, synchronization and locking amongst processors is still the main bottleneck in this model. This led to the idea of running SGD in parallel without locks.



In this model, a CPU core acts as the parameter server, and other CPU cores act as threads. All threads in a single machine have access to shared memory, each thread can independently ask for the current model in memory, and each thread computes a gradient and and updates the shared memory. Since the order that updates are sent is random due to the different runtimes of different worker nodes, and since this model is completely asynchronous, the controller in shared memory updates the model in a first-in-first-served fashion. Assuming that all threads have collected x_t, the model will be updated to

$$x_{t+1} = x_t - \eta(\nabla f_{i_t}(x_t) + \nabla f_{j_t}(x_t) + ... + \nabla f_{q_t}(x_t))$$

However, this is the most straightforward case. Due to asynchrony, threads might process an older model version and complete their tasks in an arbitrary order. It is also possible that threads read a model state that only saved in memory for a short time and between other memory writes. Despite this complexity and uncertainty, it has been shown in a paper by Google titled *Large Scale Distributed Deep Networks* that asynchronous SGD works very well for training deep networks, particularly when combined with Adagrad adaptive learning rates. Additionally, while asynchronous SGD is rarely applied to nonconvex problems and there is little theoretical grounding for the safety of these operations for these problems, they found that relaxing consistency requirements is remarkably effective in practice.



We can see in the graph above that asynchrony can lead to upwards of 10x speedup in some cases, but in other cases there may not be any speedup because as previously mentioned, communication bottlenecks may cause workers to read an older model state and consequently overwrite progress.

HOGWILD!: an update scheme that allows processors access to shared memory with the possibility of overwriting each other's work. Consider the following setting:

$$\min_x f(x) := \sum_{e \in E} f_e(x_e)$$

where:

E is a collection of items, say samples$x \in \mathbb{R}^n$, $e \subset [n]$ each element e is a col

To clarify, while $f_e(\cdot)$ denotes a component of a sum of functions indexed by sample e, x_e is the sub-vector indexed by an index set e. In other words, each sample $e \in E$ corresponds to an index set $e \subset [n]$. This is more tangible when we consider the following key observation:

n and $|E|$ are large, while individual $f_e(\cdot)$ act on a small number of components $x \in \mathbb{R}^n$

Example: Sparse SVM

Given data $E = \{(z_1, y_1), ..., (z_{|E|}, y_{|E|})\}$ where y_i are labels and $z_i \in \mathbb{R}^n$ are features, we solve:

$$\min_x \sum_{\alpha \in E} \max(1 - y_\alpha \cdot x^\top z_\alpha, 0) + \lambda \|x\|_2^2$$

Often, z_α is sparse, meaning $x^\top z_\alpha = x_\alpha^\top z_\alpha$, where x_α is a sub-vector of x whose nonzero components correspond to the nonzero components of z_α. Analogously, for each sample $e \in E$, only a set of components of x indexed by elements of $e \subset [n]$ would be affected by the corresponding function $f_e(\cdot)$.

Algorithm 1 HOGWILD! update for individual processors

1: **loop**
2: Sample e uniformly at random from E
3: Read current state x_e and evaluate $G_e(x)$
4: **for** $v \in e$ **do** $x_v \leftarrow x_v - \gamma b_v^T G_e(x)$ ← (coordinate−wise)
5: **end loop**

where $G_e \in \mathbb{R}^n$ is a gradient with non-zeros indexed by e and scaled such that $\mathbb{E}[G_e(x_e)] = \nabla f(x)$.

Verbally, we can describe this algorithm as follows:

1. Each processor samples e uniformly at random
2. Each processor computes the gradient ∇f_e at x_e
3. Each processor applies update on each coordinate in e

Notice that even if two processes update the model asynchronously, by construction, the sparsity of the update made by each process means that the probability of overlap between the indices affected by the two processes is very small. This allows computations to run in parallel without the need for any synchronization. This is reminiscent of the coordinate-parallel distribution of SGD mentioned earlier, except without the need for synchronization. This is because of the small overlap between indices affected by any two processes, and the assumption of a small delay between the models taken in by each process.

These properties hold for the asynchronous HOGWILD! algorithm:

1. When the data access is sparse, memory overwrites could be rare
2. This indicates that asynchrony introduces barely any error in the computations (this can explain why asynchronous SGD caused massive speedup in some cases and no speedup in others).
3. The authors show both theoretically and experimentally a near-linear speedup with the number of processors used.
4. In practice, lock-free SGD exceeds even theoretical guarantees

Alternatives to asynchrony. Asynchrony is used to reduce communication overheads. There are other ways to achieve this:

1. One form of lower level optimization that can be done is, instead of representing each entry of the gradient as a float number (the size of each gradient sent over the network is $O(32 \cdot p)$ bits) as done in Standard SGD, Quantized SGD quantizes each entry of some gradient to some levels such that the size of data sent over the network is $O(l \cdot p)$ bits, where $l << 32$ bits represent the levels of quantization. Intuitively, this introduces a tradeoff in the form of error, but it has been shown that 4-bit quantization with 16 GPUs could lead to 3.5x speedup.
2. In the synchronization step of each iteration, instead of waiting for stragglers, the parameter node should proceed when most of the worker nodes have finished their execution.
3. Instead of quantizing all entries of a gradient, only keep the most important entries and zero out the rest.
4. Give more "work" to workers by increasing the batch size. This allows the update to use a larger step size, thus reducing the number of iterations, communications and synchronizations. This needs careful parameter utning to work.
5. Variants of HOGWILD! that minimize communication conflicts: some computatioon is performed to distribute examples to different cores so that examples do not "conflict"

Conclusion. Distributed computing is at the heart of developments in modern ML; there are conferences dedicated to it, and there are a variety of approaches from high-level algorithms to low-level implementations such as quantized SGD. We have looked at the different ways that distributed computing could be exploited: **hyper parameter optimization, coordinate descent, mini-batch synchronous SGD, asynchronous SGD**. The type of distributed computing configurationt to use depends on the problem and resources at hand; asynchronous SGD is optimal for sparse problems, synchronous SGD running on multicore workstations may be more effective in handling non-sparse problems, and so on. Distributed computing for SGD is still a highly researched topic.