DSC-204A: Scalable Data Systems, Winter 2024

ML System - 1

Lecturer: Hao Zhang *Scribe: Yi Gu, Shibo Hao, Laurel Li*

1 ML System History

- ML era roughly starts in 2008
- ML was in a mess in 2012. There are lots of systems and ML models built by different individuals.
  Cons: Every model can be the best model.
  - There is no unified model/computation.
  - Hard to build a programming model/interface that covers a diverse range of applications
  - No idea where the system bottleneck it

  Pros: There are a lot of opportunities (high risk, high return).

ML systems evolve as more and more ML components (models/optimization algorithms) are unified.

More and more unified scopes becoming narrow and narrow.

- Ad-hoc: diverse model family, optimization algorithms, and data
  There is no ML system because it is so hard to build one based on this understanding.
- Opt algorithm: iterative-convergent
  The generate method to train models is to iterate until it converges.
- Model family: neural nets
  People realize to focus on the neural networks because they can speed up using GPUs and start building neural network libraries.
- Model: CNNs / transformers / GNNs
  People started scaling up these models.
- LLMs: transformer decoders
  People started looking at LLMs.

ML systems have a breakthrough when every time ML has a breakthrough.

2 Iterative-convergence Algorithm

Iterative-convergence algorithm is the first unified component.

- Decision tree: Gradient Boosting Tree
- Matrix Representation: Coordinate descent
- LDA: EM algorithm
- Neural networks: Gradient Descent

Even different models can all be trained by iterative-convergence algorithm.

2.1 Formulation

A step of iterative-convergent algorithm is described as follows:

$$\theta^{(t)} = \theta^{(t-1)} + \epsilon \cdot \nabla_{\mathcal{L}}(\theta^{(t-1)}, D^{(t)}),$$

The ∇ symbol can be gradient (in the case of gradient descent algorithm), or any backward computation. If we have P workers, each update is to summarize the local update from all workers.

$$\theta^{(t)} = \theta^{(t-1)} + \epsilon \cdot \sum_{p=1}^{P} \nabla_{\mathcal{L}}(\theta^{(t-1)}, D_p^{(t)}),$$

If expressing this with Spark, there is very heavy communication per iteration. In the era of 2012, the cost of this update has compute : communication = 1 : 10.

2.2 Strict Consistency

Bulk Synchronous Parallel (BSP, Figure 1):

- BSP suffers from stragglers: slow devices (stragglers) force all devices to wait; more devices means a higher chance of stragglers
- Stragglers are usually transient, e.g., temporary computer/network load in a multi-user environment, fluctuating environmental conditions like temperature and vibrations
- BSP's throughput is greatly decreased in large cluster/clouds, where stragglers are unavoidable

BSP has strict consistency, so execution is serializable.

MapReduce, Spark, and many DistML Systems are based on the baseline of BSP. Devices compute updates between global barriers (iteration boundaries) and Messages exchanged only during barriers.

2.3 Asynchronous Communication

Asynchronous communication removes all communication barriers:

- Maximizes computing time
- Transient stragglers will cause messages to be extremely stale

If no communication, each worker is training with their own data to update their own objective and makes no progress. If random communication, we cannot guarantee ML model can converge the objective.

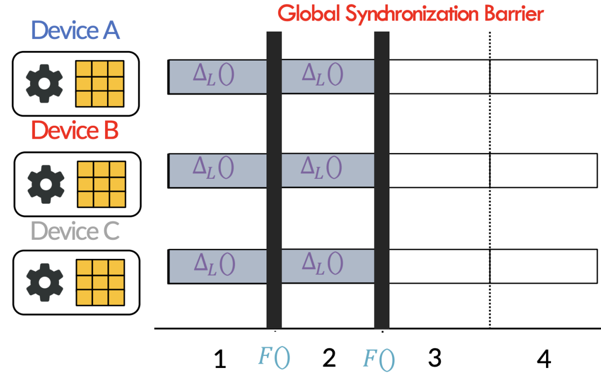

Figure 1: An illustration of Balk Synchronous Parallel (BSP): Devices compute updates between global barriers, and it suffers from stragglers.

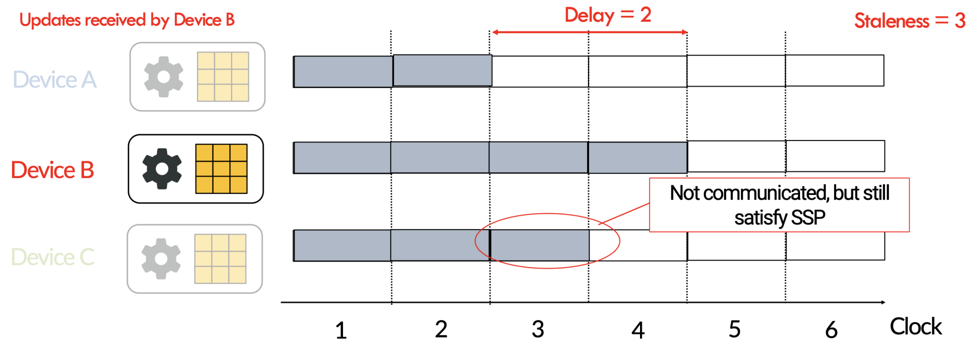

Figure 2: An illustration of Stale Synchronous Parallel (SSP). To avoid communication overhead, it relaxes the constraint of strict consistency and allows for lazy communication.

2.4 Bounded Consistency

Machine learning is error-tolerant (under certain conditions). We can allow the algorithm to go wrong a little bit.

Stale Synchronous Parallel (SSP): Devices allowed to iterate at different speeds:

- Fastest and slowest device must not drift $> s$ iterations apart
- s is the maximum staleness

In SSP, devices avoid communication unless necessary (Figure 2).

- i.e. when staleness condition is about to be violated
- Favors throughput at the expense of statistical efficiency

theorem 1 (SSP Expectation bound)**.** *The objective is L-Lipschitz, the problem diameter is bounded by F^2, the staleness s, and P devices are used. The step size is $\eta_t = \frac{\sigma}{\sqrt{t}}$, where $\sigma = \frac{F}{L\sqrt{2(s+1)P}}$. Where T is the*

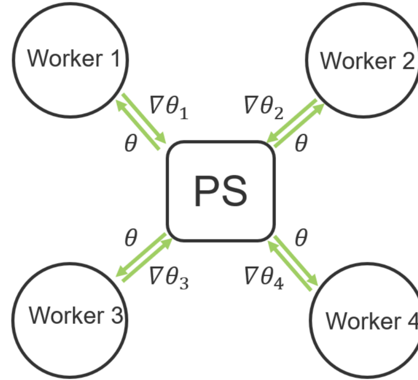

Figure 3: The conceptual architecture of Parameter Server.

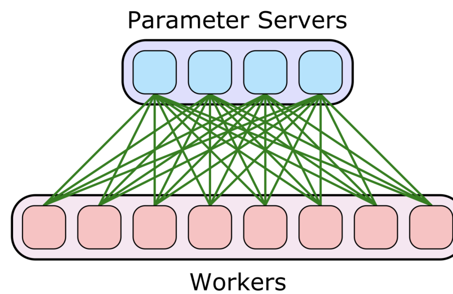

Figure 4: The architecture of sharded Parameter Server.

number of iterations, we can bound the difference between SSP estimate and the true optimum as follows:

$$R[\mathbf{X}] := \left[\frac{1}{T}\sum_{t=1}^{T} f_t\left(\tilde{\mathbf{x}}_t\right)\right] - f\left(\mathbf{x}^*\right) \leq 4FL\sqrt{\frac{2(s+1)P}{T}}$$

3 Parameter Server

Parameter Server is a system architecture to implement SSP for iterative-convergence optimization algorithm (Figure 3). Conceptually, there is a server that holds parameter updates, and multiple workers that derive the updates. Each time the workers send the update to the server, these updates are aggregated on the server and sent back to the workers. Then the workers apply the update and also update the clock. The timing of update is controlled by the staleness value.

3.1 Implementation

There are some key considerations in the implementation of Parameter Server.

- How to prevent the server from being the communication bottleneck?
- Fault tolerance: what if the server breaks down?

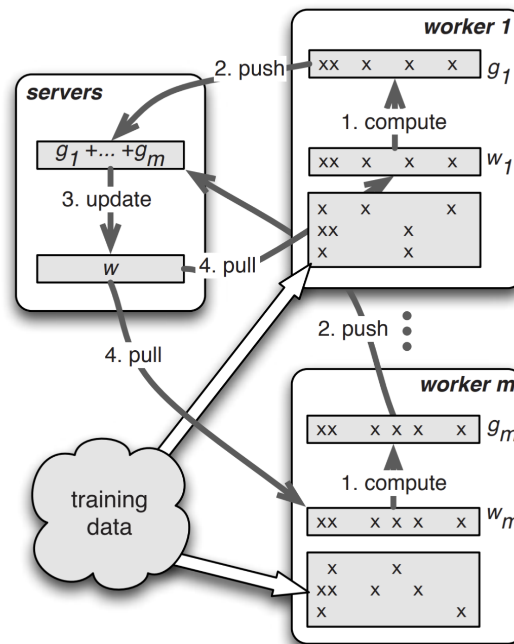

Figure 5: An illustration of the programming model of Parameter Server.

- How to design the programming model?
- Handling GPUs

To solve the first and second problems, we may set up many servers with sharded KV stores (Figure 4), which can even be co-located with the workers. We can also create redundancy across different servers for fault tolerance.

The programming model is of a similar spirit to Map Reduce. It allows great flexibility for users to customize. The APIs for a client include `Push()`, `Pull()`, and `Compute()`, and the main API for a server is `Update()`. The Programming model is illustrated in Figure 5.

3.2 Summary

- Why does it merge? Unification of iterative-convergence optimization algorithm.
- What problem does it address? Heavy communication via flexible consistency.
- Pros? Cope well with iterative-convergent algorithm.
- Cons? Doesn't support GPUs well. Strong assumption on communication bottleneck (Doesn't hold anymore because of NVLink)