10-708: Probabilistic Graphical Models 10-708, Spring 2014

29: Distributed Systems for ML

Lecturer: Eric P. Xing *Scribes: Petar Stojanov, Christoph Dann*

1 Case Studies for Distributed Machine Learning Algorithms

In the first part of the lecture, several examples for how to distribute machine common machine learning algorithms have been discussed.

1.1 Coordinate Descent

Coordinate descent is a general strategy for convex optimization problems. The basic idea is iteratively solve the problem by optimizing the objective only with respect to one optimization variable at a time while keeping all other dimensions fixed. While the order in which the dimensions are optimized can be chosen arbitrarily, it is crucial for convergence guarantees that updates occur sequentially.

Coordinate descent can for example be used to efficiently solve the LASSO objective

$$\min_{\beta} \frac{1}{2}||y - X\beta||_2^2 + \lambda \sum_j |\beta_j|, \tag{1}$$

where we assume $X_i^\top X_i = 1$. Using the KKT conditions, one can derive that

$$\arg\min_{\beta_j} \frac{1}{2}||y - X\beta||_2^2 + \lambda \|\beta\|_1 = S_\lambda \left(X_i^\top (y - X_{-i}\beta_{-i} \right) \tag{2}$$

with $S_\lambda(x) = \mathrm{sgn}(x)\max\{0, |x| - \lambda\}$ being the soft-threshold operator. Similarly, in *block coordinate descent*, a batch of variables are updated at a time while all others are kept fixed.

One algorithm for parallel coordinate descent is *Shotgun* [Bradley et al., 2011]. In each iteration, this algorithm chooses P variables to update uniform at random and then updates these variables in parallel. Shotgun scales linearly when the features are almost uncorrelated. For perfectly correlated features, Shotgun is not faster than sequential updates.

The analysis of Scherrer et al. [2012] shows that one can mitigate this issue by instead of choosing the variables to update in parallel at random, it is beneficial to partition the dimensions into P blocks by optimizing the block spectral radius. Since this is itself a challenging combinatorial optimization problem, the authors propose a heuristic: greedily cluster dimensions with largest absolute inner product in each block. This heuristic yields good empirical results.

The parameter scheduler STRADS by Lee et al. [2014] for model-parallel machine learning is applicable to coordinate descent and can further improve upon these results. STRADS does so by leveraging two ideas for scheduling parameter updates: First, parameters are chosen randomly with higher probability assigned to parameters that changed much recently (prioritization). Second, STRADS checks the dependency by avoiding updating parameters which are too linearly dependent in parallel. See Figure 1 for example results.

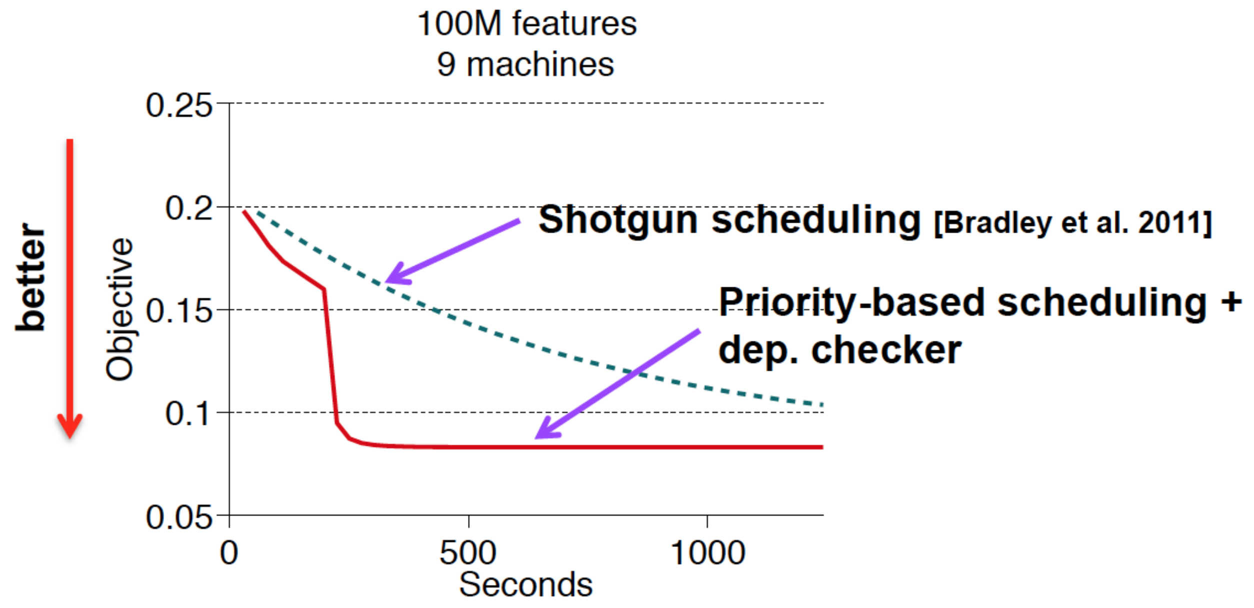

Figure 1: Performance comparison of STRADS and Shotgun

1.2 Proximal Gradient Descent

From what we learned in the vast majority of this course, we saw objective functions that are smooth and for which gradient updates are derived in a straight forward fashion. However, criteria to be optimized sometimes consist of regularization terms which are not smooth. One example of such a criterion is the LASSO, given by:

$$\min_{\beta} ||y - X\beta||_2^2 + \lambda \sum_j |\beta_j| \tag{3}$$

Let's treat the two components as a smooth component f and a non-smooth component g, yielding the criterion:

$$min_{\beta} f(\beta) + g(\beta) \tag{4}$$

Then in order to construct a gradient update, one would have to solve what is called a proximal map, an auxiliary problem given by:

$$\text{prox}_t(\beta) = \arg\min_z \frac{1}{2t}||\beta - z||^2 + g(z) \tag{5}$$

Then instead of our regular gradient update, the update for the next iteration's β becomes:

$$\beta^k = \text{prox}_t(\beta^{k-1} - t\nabla f(\beta^{k-1})) \tag{6}$$

To provide a little bit of intuition, this update is equivalent to a quadratic approximation of the smooth part f. It can be interpreted as staying close to the gradient update while also making g small. For some non-smooth regularizers, the proximal map has a closed form. There are certain properties of the proximal mapping problem that enable it to be easily computed for more complex non-smooth regularizers.

In particular, if the non-smooth regularizer is a sum of two non-smooth terms ($g = g_1 + g_2$), then we have the following property:

$$P^t_{g_1+g_2} = P^t_{g_1}(P^t_{g_2}) \tag{7}$$

Given this proximal mapping framework, there is a way to implement it in a distributed environment. Namely, given servers and workers, we perform the gradient update on the workers:

$$\delta^{k-1} = \beta^{k-1} - t\nabla f(\beta^{k-1}) \tag{8}$$

where t is the step size. Then we send this information to the servers and perform the proximal map to get the next iteration update for β:

$$\beta^k = P^t_g(\delta^{k-1}) \tag{9}$$

The new update is then returned to the workers and this process is iterated.

1.3 ADMM

In the previous section we saw how to deal with and parallelize criteria that decompose in a smooth and a non-smooth part. However there is a wide variety of problems whos objective functions do not decompose into neat sums of terms. Here is a general representation of the problem:

$$\min_{w,z} f(w) + g(z) \tag{10}$$

$$\text{s.t. } Aw + Bz = c \tag{11}$$

In this general objective function, we see that the variables w and z are uncoupled, but the constraint couples them and makes it difficult to derive proper updates. To make this problem manageable, we introduce what is called the "augmented Lagrangian", which is given by:

$$L_\mu(w, z; \lambda) = f(w) + g(z) + \lambda^T(Aw + Bz - c) + \frac{\mu}{2}||Aw + Bz - C||_2^2 \tag{12}$$

From the theory of duality, it can be shown that maximizing this augmented Lagrangian represents the initial objective in equation 8, so now we have to solve the following dual problem:

$$\min_{w,z} \max_{\lambda} L_\mu(w, z; \lambda) = f(w) + g(z) + \lambda^T(Aw + Bz - c) + \frac{\mu}{2}||Aw + Bz - C||_2^2 \tag{13}$$

Solving this problem with fixed dual variable λ yields the following updates for the $(k+1)$th step :

$$w^{k+1} \leftarrow \arg\min_w L_\mu(w, z^k, \lambda^k) = f(w) + \mu/2||Aw + Bz^k - c + \lambda^k/\mu||^2; \tag{14}$$

$$z^{k+1} \leftarrow \arg\min_z L_\mu(w^{k+1}, z; \lambda^k) = g(z) + \mu/2||Aw^{k+1} + Bz - c + \lambda^t/\mu||^2 \tag{15}$$

Now we fix the primal variables w, z, and perform the gradient update of the dual λ:

$$\lambda^{k+1} \leftarrow \lambda^k + t(Aw^{k+1} + Bz^{k+1} - c) \tag{16}$$

An obvious way to parallelize this algorithm is to perform all the primal variable updates on worker machines. The problem is that waiting for all of them to complete before updating the dual λ creates a bottleneck. This can be avoided by doing the dual update asynchronously (after seeing s out of n primal updates).

1.4 MCMC for Dirichlet Processes

Parallelizing MCMC inference often relies on the idea of auxiliary variables. By introducing additional variables into the model to introduce (conditional) independences. One example for such a strategy can be used to parallelize inference in Dirichlet processes. Here, the idea is to leverage the fact that the following process

$$\begin{aligned} D_j &\sim \mathrm{DP}(\alpha/P, H) \quad j = 1, \dots, p && (17)\\ \phi &\sim \mathrm{Dirichlet}(\alpha/P) && (18)\\ \pi_i &\sim \mathrm{Categorical}(\phi) && (19)\\ \theta_i &\sim D_{\pi_i} && (20) \end{aligned}$$

has the same marginal distribution as

$$\begin{aligned} D &\sim \mathrm{DP}(\alpha, H) && (21)\\ \pi_i &\sim \mathrm{Categorical}(\phi) && (22)\\ \theta_i &\sim D_{\pi_i}. && (23) \end{aligned}$$

Hence, inference can be conducted in parallel by randomly assigning the data to local Dirichlet process models. Then, until convergence iteratively perform Gibbs sampling in parallel for each local model and then swap clusters across models using Metropolis-Hastings.

1.5 Embarrassingly Parallel MCMC

One way to speed up MCMC algorithms is to run MCMC in parallel on subsets of the data, without communication between any of the machines. The objective is to recover the full posterior distribution after the machines have completed, given by:

$$p(\theta|x^N) \propto p(\theta)p(X^N|\theta) = p(\theta)\prod_{i=1}^{N} p(x_i|\theta) \tag{24}$$

The approach consists of partitioning the data into M subsets $\{x^{n_1}, \dots, x^{n_M}$. Then each machine has a sub-posterior defined by:

$$p(\theta|x^N) \propto p_m(\theta)^{\frac{1}{M}} p(X^{n_M}|\theta) \tag{25}$$

The approach works by:

- by sampling from the respective sub-posteriors p_m
- combining the sub-posterior samples via a non-parametric kernel density estimator. Given T samples $\{\theta^m_{t_m}\}_{t_m=1}^{T}$ from each subposterior p_m:

$$\hat{p(\theta)} = \frac{1}{T}\sum_{t_m=1}^{T}\frac{1}{h_d}K\left(\frac{||\theta - \theta^m_{t_m}}{h}\right) = \frac{1}{T}\sum_{t_m=1}^{T}\mathcal{N}_d(\theta|\theta^m_{t_m}, h^2 I_d) \tag{26}$$

- then the sub-posterior kernel density estimators can be combined:

$$\widehat{p_1, \dots, p_M}(\theta) = \hat{p}_1, \dots, \hat{p}_M(\theta) = \frac{1}{T^M}\prod_{m=1}^{M}\sum_{t_m=1}^{T}\mathcal{N}_d(\theta|\theta^m_{t_m}, h^2 I_d) \propto \sum_{t_1=1}^{T}, \dots, \sum_{t_M=1}^{T} w_t \mathcal{N}_d(\theta|\bar{\theta_{t,\cdot}}, h^2/M I_d) \tag{27}$$

  where $\theta_{t,\cdot}$ is the sample mean of the sub-posterior estimators, and $w_t = \prod_{m=1}^{M}\mathcal{N}_d(\theta|\bar{\theta_{t,\cdot}}, h^2 I_d)$

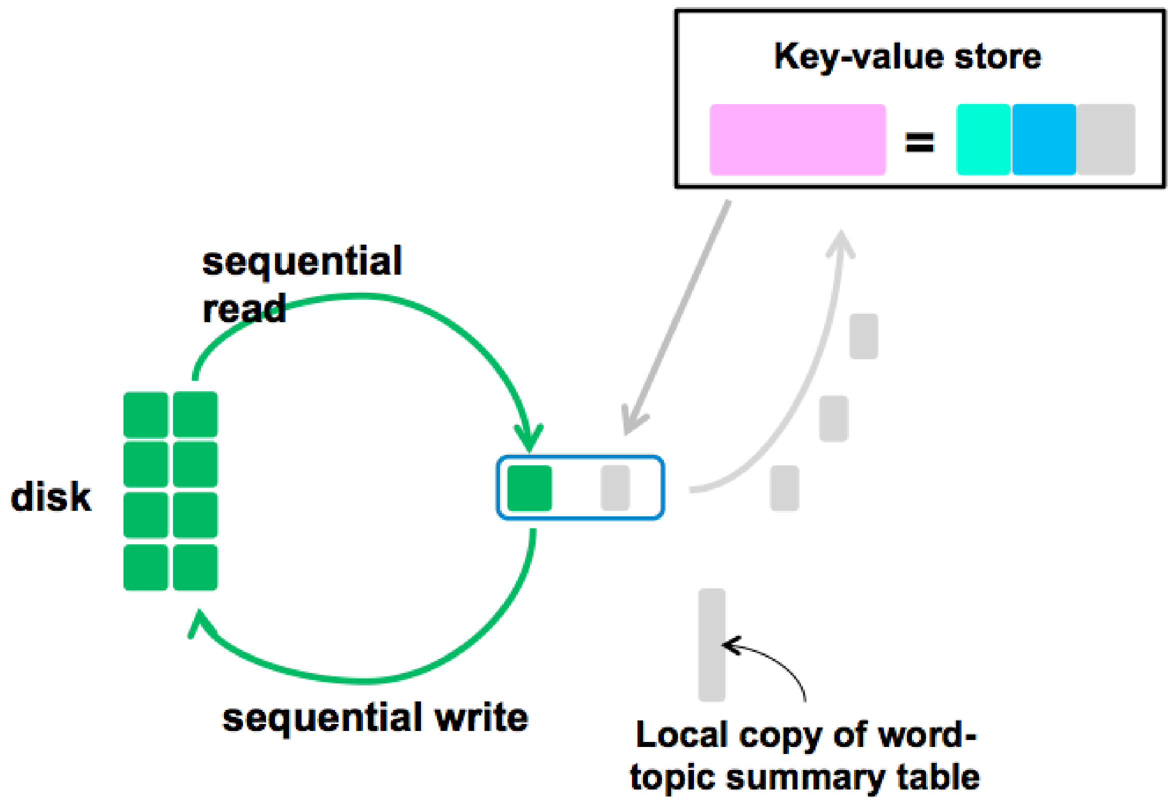

Figure 2: Diagram of the LightLDA algorithm

1.6 Gibbs Sampling for Topic Models

The most famous way of parallelizing MCMC is collapsed Gibbs sampling. It aims to solve a simple equation:

$$p(z_{ij} = k|x_{ij}, \delta_i, B) \propto (\delta_{ik} + \alpha_k)\frac{\beta_{x_{ij}} + B_{k,x_{ij}}}{V\beta + \sum_{v=1}^{V} B_{k,v}} \tag{28}$$

where δ and B are the document and word-specific (word count matrix) model parameters. The documents can be partitioned into pieces with replicated model parameters, and distributed into workers, where parallel Gibbs sampling is performed. This approach has several problems:

- Convergence is not guaranteed, because Markov chain ergodicity is broken.
- CPU cycles are wasted while synchronizing the model

The parallelism incurs error in B, and the parameters become stale during the sampling process. One way to deal with this is to have asynchronicity and more frequent communication between the machines. Another way is to partition the word count matrix into non-overlapping matrices. This method is called LightLDA and is shown in Figure 2. Briefly, the algorithm works by keeping the data on disk, and send model portions to machines as needed. A short outline:

- Preprocess the data, and determine which set of words goes to which block and organize it on disk.

- Pull a set of words from a Key-Value store.
- Perform sampling and write the result to disk sequentially, then send changes back to the key-value store.

This way, the errors due to partitioning the data are eliminated because its integrity is kept intact on disk

2 Systems and Architectures for Distributed Machine Learning

While the first part of the lecture was denoted to specific examples of distributed machine learning algorithms, higher-level architectures and general challenges for distributed ML have been discussed in the last part.

Many machine learning methods can be cast as iterative-convergent methods – for examples see previous section. There are two basic approaches for parallelization:

- **Data-Parallelism:** The data is partitioned and distributed onto the different workers. Each worker typically updates all parameters based on its share of the data (see Section 1.4 for an example).
- **Model-Parallelism:** Each worker has access to the entire dataset but only updates a subset of the parameters at a time (see Section 1.1 for an example)

Of course, both principle approaches can also be combined as data- and model-parallel approaches.

There are two major challenges in data parallelism:

1. **Need for partial synchronicity:** Synchronization (and therefore communication costs) should only happen when necessary. Workers waiting for synchronization should be avoided when possible.
2. **Need for straggler tolerance:** Slow worker need to be allowed to catch up.

One attempt at overcoming these challenges is the Stale Synchronous Parallel (SSP) model where each thread runs at its own pace without synchronization but for theoretical guarantees the threads are not allowed to drift more than some number of steps apart. To reduce communication overhead, each worker has its own cached version of parameters (which need not be up-to date). This approach has been further improved in the Eager SSP protocol (ESSP).

References

Joseph K Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. *arXiv preprint arXiv:1105.5379*, 2011.

Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing. On model parallelization and scheduling strategies for distributed machine learning. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2834–2842. Curran Associates, Inc., 2014.

Chad Scherrer, Ambuj Tewari, Mahantesh Halappanavar, and David Haglin. Feature clustering for accelerating parallel coordinate descent. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 28–36. Curran Associates, Inc., 2012.