ECE 901: Large-scale Machine Learning and Optimization **Fall 2016**

Lecture 18 — 11/10

Lecturer: Dimitris Papailiopoulos *Scribe: Vamsi K. Ithapu*

The focus of this lecture is to bridge the gap between inherently serial algorithms and parallel algorithms. Specifically, we discuss the necessary tools needed to evaluate the goodness of parallelizing a given serial algorithm. We will present the parallel versions of few standard, typically serial, optimization problems encountered in machine learning. As always stochastic gradients would be the focus.

18.1 From Serial to Parallel

We have the following notation.

$\boldsymbol{A^s(\cdot)}$: The *output* of serial algorithm.

$\boldsymbol{A^p(\cdot)}$: The *output* of parallel algorithm.

Output? This is a very vague term. It could correspond to the final estimate from the algorithm, or it could simply represent the test-set error on a hold-out dataset, or a more involved function of the estimates from last few hundred or so iterations.
We choose apriori what this *output* is, and move on!
Speed-up: Conditioned on this quantification of the serial and output algorithms, the speed-up corresponds to the following,

$$\mathcal{S} = \frac{time(A^p(\cdot))}{time(A^s(\cdot))}$$

One would then want to minimize $\mathcal{S}$ – an ideal parallel scheme would achieve the best possible $\mathcal{S}$ with *high probability*, or in other words, the worst case $\mathcal{S}$ is as large as possible. $time(\cdot)$ here represents the CPU time i.e, the above expression can be written as

$$\mathcal{S} = \frac{iter(A^p(\cdot)) * time-per-parallel-iter}{iter(A^s(\cdot)) * time-per-serial-iter}$$

Traditionally, the time per serial or parallel iteration is proportional to the time taken by a small, but similar, set of computations on a given computational course (a work station or computer). From the perspective of optimization algorithm design and modeling, this is difficult to keep track of. So, typically, one would instead use the following expression for the speed-up,

$$\mathcal{S} \approx \frac{iter(A^p(\cdot))}{iter(A^s(\cdot))}$$

Hence, if we can *fix* the outputs of the serial and parallel algorithms, then the speed-up is given by the ratio of the number of iterations. Taking cues from the world of databases, an aspect called **serial equivalence** will allow us to do precisely this.

18.1.1 Serial equivalence

Definition: Given two different algorithms A^1 and A^2, they are referred to as algorithmically equivalent whenever $A^1(s) = A^2(s)$ for all the inputs s.
Note that this algorithmic equivalence is a statement for a specific s i.e., this is not an 'in expectation' or 'worst-case' statement. In this sense, this algorithmic equivalence is more powerful a statement than the average or asymptotic analysis.
Definition: Given a set of inputs s, and a serial algorithm A^s and parallel algorithm A^p, and a sequence π where each element of π comes from $[1, N]$. These two algorithms are equivalent whenever $A^s(s, \pi) = A^p(s, \pi)$

OK. Why the fuss about this serial equivalence?
We list down the Pros and Cons of using this serial equivalence, and in turn provide reasonable evidence for why fixing serial equivalence and then asking about the speed-up would make sense.

Pros	**Cons**
All serial tricks/heuristics follow to parallel;	Too much to ask – unclear how to guarantee in general;
Reproducibility – the estimates are in some sense robust across multiple runs;	Need memory locks and synchronization;
Better than the worst case;	Generally leads to slow parallel algorithms;
	Speed-up guarantees are not trivial;

18.2 A prototypical Case

Let us consider our prototypical stochastic gradients to discuss more details.
Goal: Build a parallel framework for stochastic gradient descent with

- *serially equivalence*, and
- *provable speed-up guarantees*

Stochastic gradients is an inherently serial algorithm, i.e., **it has memory**.
Recall the 1 batch-size setting and the update,

$$\min \sum_{i=1}^{N} f^i(x) \qquad x_{k+1} \leftarrow x_k - \gamma \nabla_x f^j(x) \qquad j \sim Unif[1, N] \tag{18.1}$$

The update at $k + 1^{th}$ iteration depends on the choices of the js used in the previous k iterations. Although there are guarantees about the algorithm's robustness to the choice of j across multiple repetitions of stochastic gradients, nevertheless this inherently serial nature is a hurdle for deriving any sensible convergence or speed-up guarantees.

18.2.1 Sparsity to the rescue? Or may be not?

Consider 2 different *parallel* cores which are available for performing computations independently. Running parallel stochastic gradients entails to performing the gradient updates on each of the two cores independently. Consider two consecutive iterations. The "inherently serial" nature between these two iterations can be de-coupled by simply *forcing* the set of parameters that are *read and updated* in these two iterations to be different. Once de-coupled, the two iterations can, in principle, be performed in tandem i.e., in parallel. If instead of forcing this characteristic, the loss function naturally lends itself to allow for this kind of de-coupling, then the main bottleneck for parallelizing stochastic gradients is avoided totally. More formally, consider the following structure for the loss function on which stochastic gradients operates

$$\min \sum_{i=1}^{N} f^i(x) \qquad f^i(x) = \ell(x^T a_i) \qquad \nabla_x f^i(x) = \nabla \ell(x^T a_i) a_i \tag{18.2}$$

Iff $a_i \forall i$ is sparse, then one can choose $f^i(x)$ and $f^{\hat{i}}(x)$ that are used in two consecutive serial iterations such that the supports of a_i and $a_{\hat{i}}$ are mutually disjoint. In this setting, a direct speed-up of 2 would be obtained conditioned on the fact that the serial and parallel updates are shown to have the serial equivalence.
We point out that there are many machine learning problems that naturally take up this form for the loss function. These include support vector machines, linear and logistic regression including matrix completion and factorization.