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

Lecture 15 — April 03

Lecturer: Dimitris Papailiopoulos *Scribe: Yijing Zeng, Zhongkai Sun*

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

15.1 Serial Equivalence

Consider an asynchronous machine learning setting on a single multi-core machine. Let A_{serial} denote a serial algorithm and A_{parallel} denote a parallel algorithm. Then A_{parallel} is a serial equivalence of A_{serial} if

$$A_{\text{serial}}(S, \pi) = A_{\text{parallel}}(S, \pi), \tag{15.1}$$

for all dataset S and all data order π. Note that for any data order π, the data points can be repeated in arbitrary order.

From the theoretical perspective, there are two main advantages of proposing the serial equivalence:

1. We only need to "prove" the speedups of the parallel case over the serial case.
2. Convergence proofs of the parallel case is inherited directly from the serial case.

On the other hand, there are two main issues about the serial equivalence:

1. The notion of serial equivalence is very strict.
2. We cannot guarantee any speedups in the general case.

15.2 The Algorithmic Family of Stochastic-Updates

We consider a family of randomized algorithms that leverage Stochastic-Updates. The main algorithmic component of stochastic updates focuses on updating small subsets of a model variable $\mathbf{x}$, according to prefixed access patterns. Alg. 3 presents the pseudo code of it. In Alg. 3, S_i is a subset of the coordinates in $\mathbf{x}$, each function f_i operates on the subset S_i of coordinates, and u_i is a local update function that computes a vector with support on S_i using as input $\mathbf{x}_{S_i}$ and f_i.

There are several machine learning algorithms belong to the stochastic updates family, including SGD, SVRG/SAGA, Matrix Factorization, word2vec, K-means, Stochastic PCA, Graph Clustering, etc.

Algorithm 1 Stochastic Updates

Input: model $\mathbf{x}$, local loss function $f_i, i \in \{1, \ldots, n\}$, local update function $u_i, i \in \{1, \ldots, n\}$, total number of iterations T, the distribution of drawing samples D (from $\{1, \ldots, n\}$).
for t=1 to T **do**
  sample $i \sim D$.
  $\mathbf{x}_{S_i} = u_i(\mathbf{x}_{S_i}, f_i)$. //Update global model on S_i.
end for
Output: updated model $\mathbf{x}$.

15.3 Conflict Graph for Parallel Updates

A useful construct for parallel updates is the conflict graph between updates, which can be generated from the bipartite graph between the updates and the model variables. Fig. 15.3 illustrates the process of conflict graph construction.

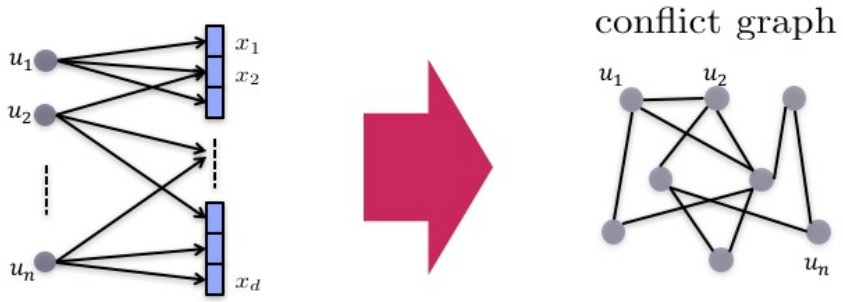

Figure 15.1. Conflict Graph Construction. In the left bipartite graph, an update using u_i is linked to variable x_j when u_i needs to read/write x_j. Then we can obtain the conflict graph on the right, where there is an edge between two updates if they overlap.

After conflict graph construction, the problem becomes to appropriate sampling and allocation of updates that can lead to near optimal parallelization. We leverage the following lemma by Krivelevich [].

Lemma: Let G be a graph on n vertices with max degree Δ. If we sample less than $B \leq (1-\epsilon)\frac{n}{\Delta}$ vertices (with or without replacement), then the largest connected component in the induced sub-graph has size $O(\frac{\log n}{\epsilon^2})$ with high probability.

Fig. 15.2 illustrates the conflict graph sampling and connected components computing process.

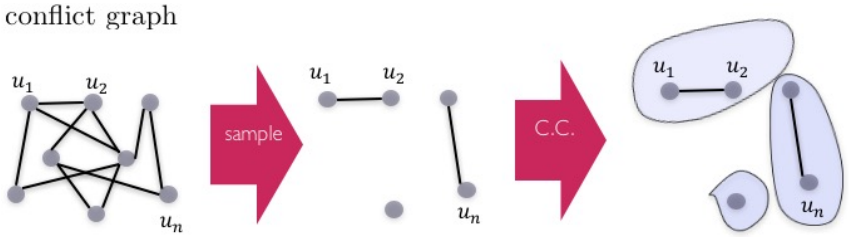

Figure 15.2. The process of sampling conflict graph and computing connected components.

15.4 Building a Parallelization Framework using the Lemma

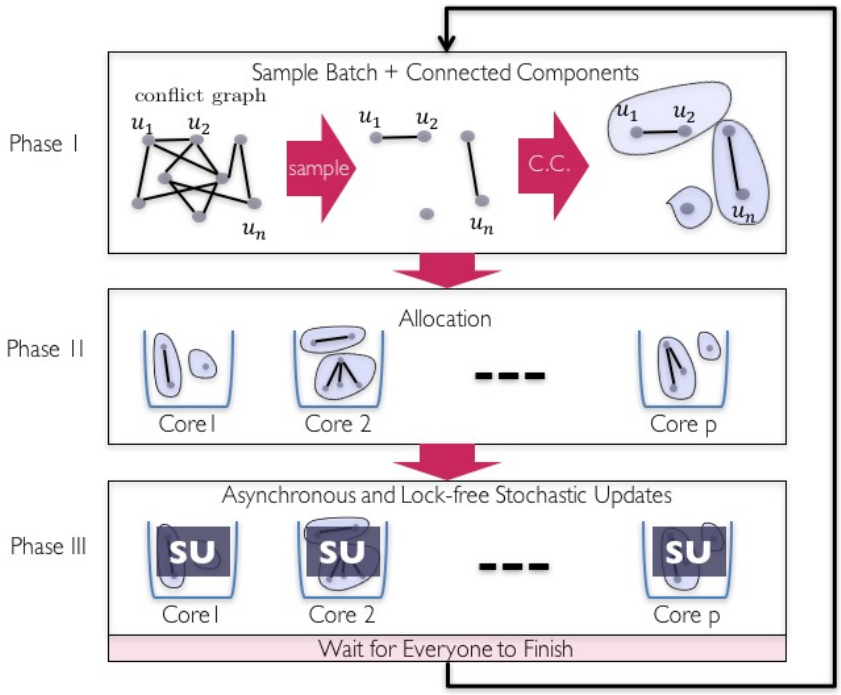

Figure 15.3. The parallel framework.

We build a parallelization framework called CYCLADES based on the above lemma. The framework is divided into three phases, which is shown in Fig. 15.3. Alg. 2 shows the detailed algorithm.

1. Phase I.

   Sample $B = (1-\epsilon)\frac{n}{\Delta}$ vertices. Compute the connected components. The number of connected components is on the order of $O(\frac{n}{\Delta \log n})$. For different connected components, there are no conflicts across them.

2. Phase II.

   Let the number of cores $p = O(\frac{n}{\Delta \log n})$. Allocation all connected components to different cores. This makes sure that we run the updates serially inside each connected component.

   There are different policies to allocate the connected components to different cores. We give two examples here. Policy 1: Random allocation. It is good when cores is far less than the batch size B. Policy II: Greedy min-weight allocation. It is 80% as good as the optimal, obtaining of which is NP-hard.

3. Phase III.

   Each core runs asynchronously and lock-free. There is no memory contention during writing or reading. In the end, wait every core to finish and start from Phase I again.

Algorithm 2 CYCLADES

Input: Conflict graph G, number of iterations T, batch size B.
Sample $n = T/B$ subgraphs $G^1, \ldots, G^n$ from G.
Compute connected components for sampled subgraphs in parallel.
for batch i=1 to B **do**
  Allocation of connected components to p cores.
  for each core in parallel **do**
    for each allocated connected component C **do**
      for each update j in order from C **do**
        $\mathbf{x}_{S_i} = u_i(\mathbf{x}_{S_i}, f_i)$.
      end for
    end for
  end for
end for
Output: updated model $\mathbf{x}$.

15.5 Speedups of CYCLADES

We only need to show that Phase I and Phase II are fast.

Theorem: Let us assume any given update-variable graph G with average and max degree $\bar{\Delta}_L$ and Δ_L such that $\frac{\bar{\Delta}_L}{\Delta_L} \leq \sqrt{n}$, and with induced max conflict degree Δ. Then, CYCLADES on $p = O(\frac{n}{\Delta \cdot \Delta_L})$ cores, cores, with batch size $B = (1-\epsilon)\frac{n}{\Delta}$ can execute $T = c \cdot n$ updates for any constant $c > 1$, selected uniformly at random with replacement, in time $O(\frac{E_u \cdot \kappa}{P} \cdot \log^2 n)$. Thus, the speedups is $\frac{P}{\log^2 n)}$.

Note that here we assume 1) max degree is not too large, 2) there are not too many cores, and 3) we sample according to the lemma. In addition, it achieves identical performance for different algorithms in the family of stochastic updates.

15.6 Fast Connected Components

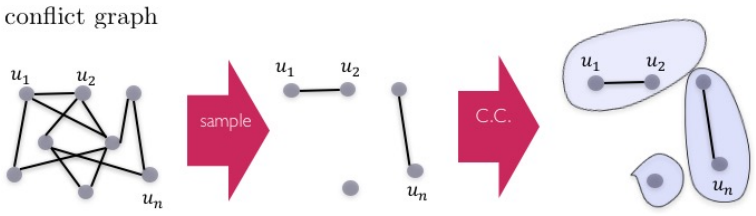

Figure 15.4. The process of sampling conflict graph and computing connected components.

If you have the conflict graph, then everything is fine. However, building the conflict requires n^2 time
Idea: Sample on the Bipartite, not on the Conflict Graphs.

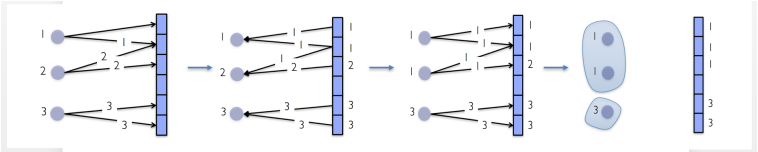

Figure 15.5. The processing of fast connected components.

Theorem 15.1. *Simple Message passing Idea:*
-Gradient send their IDS
-Coordinates Compute Min and Send Back
-Gradients Compute Min and Send Back
-Iterate till done

Cost $= O(\frac{E_u log^2 n}{P})$

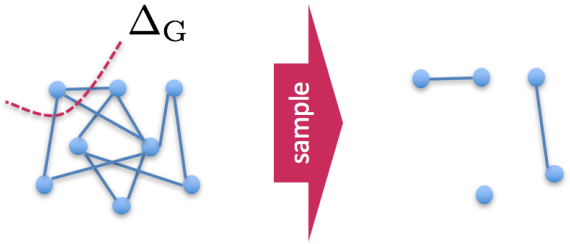

Figure 15.6. Fast Connected Components sampling.

Lemma 15.2. *Active each vertex with probability*

$$p = (1 - \epsilon)/\Delta \tag{15.2}$$

Then, the induced shatters, and the largest connected component has size $\frac{4}{\epsilon_2} logn$.

Algorithm 3 DFS with random coins

for Each vertex DFS wants to visit **do**
Flip a coin.
If 1 visit, if 0 don't visit and delete with its edges
end for

Comments:
-Assume a connected component of size k
-random coins flipped "associated" to that component $\leq k\Delta$
-Since a size k component means that the event "at least k coins are "ON" is in a set of $k\Delta$ coins"

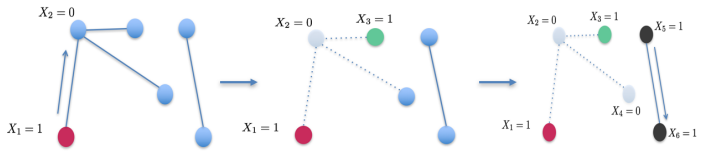

Figure 15.7. DFS with random coins

Lemma 15.3. *Consider a graph with a low-degree component + some high degree vertices Assume that there are $O(n^\delta)$ outlier vertices in the original graph C with degree at most Δ_o, and let the remaining vertices have degree at most Δ.Let the induced update-variable graph on these low degree vertices abide to the same graph assumptions. Let the batch size be bounded as*

$$B \leq min(1-\epsilon)\frac{n - O(n^\delta)}{\Delta}, O(\frac{n^{1-\delta}}{P}) \tag{15.3}$$

Then,the expected runtime will be $O(\frac{E_u k}{P} log^2 n)$

Figure 15.8. High degree vertices conflict sampling.

15.7 Experiments

Implementation in C++, experiments on Intel Xeon CPU E7-8870 v3 1TB RAM. CYCLADES v.s. Hogwild!

	Dataset	# datapoints	# features	Density (average number of features per datapoint)
SAGA	NH2010	48,838	48,838	4.8026
SVRG	DBLP	5,425,964	5,425,964	3.1880
L2-SGD	MovieLens	~10M	82,250	200
SGD	EN-Wiki	20,207,156	213,272	200

Figure 15.9. Description of experiment dataset.

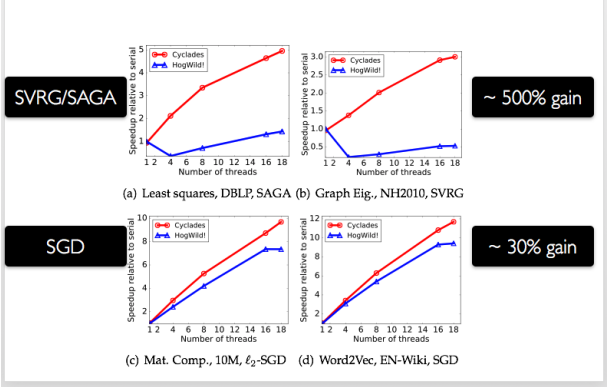

Figure 15.10. Result of Speedups.

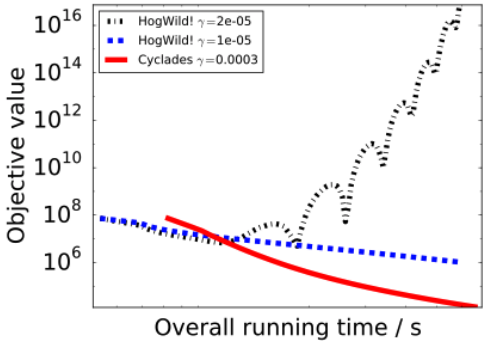

Figure 15.11. Comparison of Convergence.

15.8 Summary of CYCLADES

CYCLADES: a framework for Parallel Sparse ML algorithms.

1. Lock-free + (maximally)asynchronous.
2. No Conflicts.
3. Serializable
4. Black-box analysis