CME 323: Distributed Algorithms and Optimization, Spring 2016
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 17, 5/23/2016. Scribed by Xin Jin and Yi-Chun Chen.

Lecture outline:

- Covariance matrices and all-pairs similarity
- Computing $A^\mathsf{T}A$
- General convex optimization, where "general" means gradient doesn't necessarily exist

What can be considered a good distributed algorithm? For PRAM models, a good parallel algorithm usually has the same work as the sequential version, and has small, say logarithmic, depth. However, for distributed algorithms, having communication costs that are independent of some input dimensions is often a more important consideration.

17.1 Computing $A^\mathsf{T}A$

A is an $m \times n$ matrix:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

We assume A is tall and skinny, i.e, $m \gg n$. Examples values would be $m = 10^{12}$ and $n \in \{10^4, 10^6\}$. Moreover, A has sparse rows, each of which has at most L nonzeros. In general, A is stored across hundreds of machines and cannot be streamed through a single machine. In particular, even two consecutive rows may not be on the same machine.

An example of A in real applications would be a Netflix matrix: A lot of users and only a few movies. Rows are all sparse, but some column can be very dense, e.g., the column for movie *Godfather*.

Our task is to compute $A^\mathsf{T}A$ which is $n \times n$, considerably smaller than A. $A^\mathsf{T}A$ is in general dense; each entry is simply a dot product of a pair of columns of A.

Algorithm 17.1 NaiveMapper (r_i)

for all pairs (a_{ij}, a_{ik}) in r_i **do**
  Emit $((j,k) \to a_{ij}a_{ik})$
end for

We first describe an easy but naive algorithm: since each entry of the product matrix is of the form $\sum_i a_{ik}a_{ij}$, we can emit all entry products between column k and j to one reducer and let

Algorithm 17.2 NaiveReducer $((i,j), \langle v_1, \ldots, v_R \rangle)$

output $c_i^\mathsf{T} c_j \to \sum_{i=1}^R v_i$

that reducer do the summation. See (17.1) and (17.2) for the pseudocodes of the Mapper and the Reducer. Easy to see that the shuffle size is $O(mL^2)$ and the largest reduce-key is $O(m)$. Since m is usually very large (e.g., 10^{12}), this algorithm would not work well. It turns out that we can bring down both complexities via clever sampling.

We need the following notion of similarity of two vectors

Definition 17.1 (Cosine Similarity) *The* cosine similarity *between two columns c_i and c_j is defined as*

$$\cos(i,j) = \frac{c_i^\mathsf{T} c_j}{\|c_i\| \|c_j\|}.$$

Algorithm 17.3 DIMSUMMapper (r_i)

for all pairs (a_{ij}, a_{ik}) in r_i **do**

With probability $\min\left\{1, \frac{\gamma}{\|c_j\| \|c_k\|}\right\}$, emit $((j,k) \to a_{ij} a_{ik})$

end for

Algorithm 17.4 DIMSUMReducer $((i,j), \langle v_1 \ldots, v_R \rangle)$

if $\frac{\gamma}{\|c_i\| \|c_j\|} > 1$ **then**

output $b_{ij} \to \frac{1}{\|c_j\| \|c_j\|} \sum_{i=1}^R v_i$

else

output $b_{ij} \to \frac{1}{\gamma} \sum_{i=1}^R v_i$

end if

The *Dimension Independent Matrix Square using MapReduce* (DIMSUM) algorithm is described in (17.3) and (17.4). The DIMSUM algorithm outputs the cosine similarities (in fact probabilistic estimates of the cosine similarities). Also note that you need to compute the norms of columns beforehand (which requires all-to-all communication). To see why the outputs are estimates of the similarities, define $\tilde{v}_i$ to be v_i if v_i is emitted in (17.3) and 0 if not, and R the number of nonzero products for c_i and c_j (v_i and R are the same as in (17.1) and (17.2), not as in (17.3) and (17.4)). Then the expectation of an output from (17.4) is

$$E\left[\frac{1}{\gamma} \sum_{i=1}^R \tilde{v}_i\right] = \frac{1}{\gamma} \mathbb{P}\left(\tilde{v}_i = v_i\right) \sum_{i=1}^R v_i = \frac{1}{\|c_i\| \|c_j\|} \sum_{i=1}^R v_i.$$

We shall prove that the shuffle size is $O(nL\gamma)$ and largest reduce-key is $O(\gamma)$. To simplify the proof, we assume $A_{ij} \in \{0,1\}$, and thus $\|c_i\|_2 = \sqrt{\#(c_i)}$, where $\#(c_i)$ denotes the number of nonzeros

in c_i. Also we let $\#(c_i, c_j)$ denote the number of co-occurrences of nonzeros in c_i and c_j, i.e., the number of k's for which $c_{ik}c_{jk} \neq 0$.

Theorem 17.2 *For* $\{0,1\}$ *matrices, the expected shuffle size of* `DIMSUMMapper` *is* $O(nL\gamma)$.

Proof: The expected contribution from each pair of columns will constitute the shuffle size:

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i+1}^{n}\sum_{k=1}^{\#(c_i,c_j)} \mathbb{P}\left(\text{DIMSUMEmit}(c_i,c_j)\right) &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \#(c_i,c_j)\mathbb{P}\left(\text{DIMSUMEmit}(c_i,c_j)\right) \\
&\leq \sum_{i=1}^{n}\sum_{j=i+1}^{n} \gamma\frac{\#(c_i,c_j)}{\sqrt{\#(c_i)}\sqrt{\#(c_j)}} \\
&\leq \frac{\gamma}{2}\sum_{i=1}^{n}\sum_{j=i+1}^{n} \#(c_i,c_j)\left(\frac{1}{\#(c_i)}+\frac{1}{\#(c_j)}\right) \quad \text{(by AM-GM)} \\
&\leq \gamma\sum_{i=1}^{n}\frac{1}{\#(c_i)}\sum_{j=1}^{n}\#(c_i,c_j) \\
&\leq \gamma\sum_{i=1}^{n}\frac{1}{\#(c_i)}L\#(c_i) \\
&= \gamma Ln.
\end{aligned}$$

■

On the other hand, the expected reduce-key size is at most

$$\min\left\{\#(c_j),\#(c_k)\right\} \times \frac{\gamma}{\|c_j\|\|c_k\|} \leq \gamma,$$

which implies the largest reduce-key is $O(\gamma)$.

The key improvement of the DIMSUM algorithm over the naive one is that the complexities now are independent of the dimension m. This is achieved by sampling high magnitude columns with low probabilities. The parameter γ is a knob that can be used to preserve similarities and singular values:

- With a low setting of γ, i.e., $\gamma = \Omega(\log(n)/s)$, preserve similar entires of $A^{\mathsf{T}}A$
- With a high setting of γ, i.e., $\gamma = \Omega(n/\epsilon^2)$, preserve singular values of $A^{\mathsf{T}}A$

Full proofs of the preservation properties can be found in [1].

17.2 Gradient descent

$$F(x) = \sum_{i=1}^{n} F_i(x) \tag{1}$$

$$\nabla F(x) = \sum_{i=1}^{n} \nabla F_i(x)$$

$\nabla F_i(x) \in \mathbb{R}^d$, and the sum is computed via Reduce. The communication cost is independent of n. For general convex optimization problems, you may not have gradients (requires differentiability of objectives) or even subgradients (requires continuity of objectives).

ADMM Introduce x_i, for which each machine gets one x_i. Instead of (1), we solve the following problem

$$\begin{aligned} \min_{x_i, z} \quad & \sum_{i=1}^{P} F_i(x_i) \\ \text{such that} \quad & \forall 1 \leq i \leq p, x_i - z = 0 \end{aligned}$$

The alternative formulation can be numerically solved by the following iterative algorithm:

$$\begin{aligned} x_i^{k+1} &= \operatorname*{argmin}_{x_i} \left\{ F_i(x_i) + (y_i^k)^\mathsf{T} \left(x_i - \bar{x}^k \right) + \frac{p}{2} \|x_i - \bar{x}^k\|_2^2 \right\} \\ y^{k+1} &= y_i^k + P \left(x_i^{k+1} - \bar{x}^{k+1} \right) \\ \bar{x}^{k+1} &= \frac{1}{P} \sum_{i=1}^{P} x_i^k \end{aligned}$$

Only requires all-to-one communication of size d. For details, see [2].

If you have strongly convex functions, you can do parallel SGD:

1. Shuffle data
2. Solve locally on each machine
3. Average results

References

[1] Zadeh, Reza Bosagh, and Gunnar Carlsson. "Dimension independent matrix square using mapreduce." arXiv preprint arXiv:1304.1467 (2013).

[2] Boyd, Stephen, et al. "Distributed optimization and statistical learning via the alternating direction method of multipliers." Foundations and Trends in Machine Learning 3.1 (2011): 1-122.