Block Splitting for Large-Scale Distributed Learning

Neal Parikh and Stephen Boyd
Department of Computer Science and Information Systems Laboratory, Department of Electrical Engineering, Stanford University

Distributed optimization

- we want to decompose machine learning problems by both features (variables) and training examples, in order to solve large distributed problems
- consider the problem form

\[
\begin{align*}
\min & \quad f(y) + r(x) \\
\text{subject to} & \quad y = Ax
\end{align*}
\]

here \( f \) is a loss function and \( r \) is a regularizer
- consider partitioning \( Ax \) into \( M \) blocks as

\[
\begin{bmatrix}
A_{11} & A_{12} & \cdots & A_{1N} \\
A_{21} & A_{22} & \cdots & A_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
A_{M1} & A_{M2} & \cdots & A_{MN}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_M
\end{bmatrix}
\]

where \( A_{ij} \in \mathbb{R}^{n_i \times n_j} \)
- goal: handle each \( A_{ij} \) on a separate machine or process, and never transfer the \( A_{ij} \), over the network

Alternating direction method of multipliers

- consider generic constrained convex program

\[
\begin{align*}
\min & \quad f(z) \\
\text{subject to} & \quad z \in C
\end{align*}
\]

- ADMM, aka Douglas-Rachford splitting:

\[
\begin{align*}
\tilde{z}^{k+1/2} & := \text{prox}(y^k - \tilde{z}^k) & \text{// prox} \\
\tilde{z}^{k+1} & := \Pi_C(\tilde{z}^{k+1/2} + \tilde{z}^k) & \text{// projection} \\
\tilde{z}^{k+1} & := \tilde{z}^k + \tilde{z}^{k+1/2} - \tilde{z}^{k+1/2} & \text{// dual update}
\end{align*}
\]

Problem transformation

- assuming \( l \) and \( r \) are block separable, problem becomes

\[
\begin{align*}
\min & \quad \sum_{i=1}^M l_i(y_i) + \sum_{j=1}^N r_j(x_j) \\
\text{subject to} & \quad y_i = \sum_{j=1}^M A_{ij} x_j, \quad i = 1, \ldots, M
\end{align*}
\]

- introduce additional variables:

\[
\begin{align*}
\min & \quad \sum_{i=1}^M l_i(y_i) + \sum_{j=1}^N r_j(x_j) + \sum_{i=1}^M \sum_{j=1}^N I_{ij}(y_i, x_j) \\
\text{subject to} & \quad x_j = \sum_{i=1}^M x_{ij}, \\
y_i = \sum_{j=1}^N y_{ij}
\end{align*}
\]

Block splitting algorithm

\[
\begin{align*}
y^{k+1/2} & := \text{prox}_{1/\lambda} (y^k - \tilde{z}^{k+1/2}) & \text{// prox} \\
x^{k+1} & := \Pi_C(y^{k+1/2} + \tilde{z}^k) & \text{// projection} \\
(y^{k+1}, x^{k+1}) & := \text{exch}(y^{k+1/2}, x^{k+1/2})
\end{align*}
\]

where \( \text{exch}(c, \{x^N_j\}) \) is given by

\[
y^{k+1} = c + (c - \sum_{j=1}^N c_j)/(N+1), \quad x^{k+1} = c - (c - \sum_{j=1}^N c_j)/(N+1)
\]

and \( \text{avg} \) is elementwise averaging

Graph projection

- main work in the algorithm is often projecting onto the graph of the linear operator \( x \mapsto Ax \)
- projection onto the graph of \( A_{ij} \) is denoted \( \Pi_{ij} \)
- evaluating \( \Pi_{ij}(c, d) \) involves solving a quasi-definite linear system

\[
\begin{bmatrix}
I & A^T \\
A & I
\end{bmatrix}
\begin{bmatrix}
x \\
y
\end{bmatrix}
= \begin{bmatrix}
d + A^T c \\
0
\end{bmatrix}
\]

- if \( A \) is dense, this system can be solved via the updates

\[
\begin{align*}
y := (I + AA^T)^{-1}(Ad + AA^Tc) \\
x := d + AA^T(c - y)
\end{align*}
\]

depending on whether \( A \) is fat or skinny (respectively)
- otherwise, we can solve the quasi-definite system directly using a permuted LDL^T factorization (or an iterative or hybrid method)
- this is the only operation that touches the data, and it does not involve \( l_i \) or \( r_j \), so this operation can be reused for any problem using the same data
- key for implementation: factorization caching (cache the factorization of the coefficient matrix in the linear system being solved, then reuse across all subsequent solves)

Numerical example

- lasso:

\[
\begin{align*}
\text{minimize} & \quad (1/2)||Ax - b||_2^2 + \lambda||x||_1 \\
\text{in standard form, set} & \quad l = (1/2)||z||_2^2, r_j = \lambda \cdot ||x||_1 \\
\text{prox operators:} & \quad \text{elementwise scaling, soft thresholding}
\end{align*}
\]

- algorithm:

\[
\begin{align*}
y^{k+1/2} & := \text{proj}_{y^k} (y^k - \tilde{z}^{k+1/2}) \\
x^{k+1} & := \Pi_C(y^{k+1/2} + \tilde{z}^k) \\
(y^{k+1}, x^{k+1}) & := \text{exch}(y^{k+1/2}, x^{k+1/2})
\end{align*}
\]

- examples with dense \( A_{ij} \in \mathbb{R}^{10000 \times 10000} \)
  - distributed solver written in C using MPI and GSL (ATLAS)
- computation times (all times in seconds)

<table>
<thead>
<tr>
<th>( M \times N )</th>
<th>4 \times 2</th>
<th>8 \times 5</th>
<th>8 \times 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>nonzero entries</td>
<td>120MM</td>
<td>600MM</td>
<td>1.2B</td>
</tr>
<tr>
<td># cores</td>
<td>8</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>factorization time</td>
<td>0.05-0.15</td>
<td>0.05-0.15</td>
<td>0.05-0.15</td>
</tr>
<tr>
<td># iterations</td>
<td>90</td>
<td>230</td>
<td>490</td>
</tr>
<tr>
<td>main loop time</td>
<td>10</td>
<td>27</td>
<td>60</td>
</tr>
<tr>
<td>total time</td>
<td>28</td>
<td>50</td>
<td>80</td>
</tr>
</tbody>
</table>

Conclusions and extensions

- evaluating \( \Pi_{ij} \) is often the majority of the work
- large speedup via factorization caching
- all communication implementable via ‘allreduce’ operations
  - MPI, Vowpal Wabbit (Hadoop-compatible), etc.
- prox operations (nonlinearities) and data handled separately
  - e.g., can fit multiple models simultaneously, easily compute entire regularization path of models, do model comparison