The Alternating Direction Method of Multipliers

Customizable software solver package

Peter Sutor, Jr.

Project Advisor: Professor Tom Goldstein

April 27, 2016
The Dual Problem

- Consider the following problem (*primal problem*):
  \[ \min_x f(x) \text{ subject to } Ax = b. \]

- Important components of this problem:
  1. The Lagrangian: \( L(x, y) = f(x) + y^T(Ax - b) \)
     - We refer to the original \( x \) variable as the *primal variable* and the \( y \) variable as the *dual variable*.
  2. Dual function: \( g(y) = \inf_x (L(x, y)) \)
     - New function made purely out of the dual variable.
     - Gives a lower bound on the objective value.
  3. Dual problem: \( \max_{y \geq 0} g(y) \)
     - The problem of finding the best lower bound.

- End goal: recover \( x^* = \arg \min_x (L(x, y^*)) \), where \( x^* \) and \( y^* \) are corresponding optimizers.
The Alternating Direction Method of Multipliers (ADMM)

- Robustness of the Method of Multipliers.
- Supports Dual Decomposition → parallel x-updates possible.
- Problem form: (where \( f \) and \( g \) are both convex)
  \[
  \min (f(x) + g(z)) \text{ subject to } Ax + Bz = c,
  \]
- Objective is separable into two sets of variables.
- ADMM defines a special Augmented Lagrangian to enable decomposition: \( (r = Ax + Bz - c, \; u = y/\rho) \)

\[
L_\rho(x, z, y) = f(x) + g(z) + y^T(r) + \frac{\rho}{2}||r||_2^2
\]

\[
= f(x) + g(z) + (\rho/2)||r + u||_2^2 - \text{const}
\]

\[
= L_\rho(x, z, u)
\]
ADMM Algorithm

- Repeat for \( k = 0 \) to specified \( n \), or until convergence:
  1. \( x^{(k+1)} := \arg \min_x (L_\rho(x, z^{(k)}, u^{(k)})) \)
  2. \( z^{(k+1)} := \arg \min_z (L_\rho(x^{(k+1)}, z, u^{(k)})) \)
  3. \( u^{(k+1)} := u^{(k)} + (Ax^{(k+1)} + Bz^{(k+1)} - c) \)

- Recall the \textit{proximal operator}: (with \( v = Bz^{(k)} - c + u^{(k)} \))

\[
\text{prox}_{f,\rho}(v) := \arg \min_x (f(x) + (\rho/2)\|Ax + v\|_2^2)
\]

- If \( g(z) = \lambda \|z\|_1 \), then \( \text{prox}_{g,\rho}(v) \) is computed by soft-thresholding: (with \( v = Ax^{(k+1)} - c + u^{(k)} \))

\[
z_i^{(k+1)} := \text{sign}(v_i)(|v_i| - \lambda)_+
\]
In this project...

- Our goal is to make ADMM easier to use in practice.
- Maximizing ADMM’s potential means tweaking parameters such as step size $\rho$, starting values for $x$ and $z$, efficient proximal operators, etc., for specific problem.
- Want a comprehensive library for general ADMM use.
  - Generalized ADMM functionality (with customizable options).
  - Ready to go optimized functions for problems ADMM is most used for (with customizable options).
  - High performance computing capabilities (MPI).
  - Implementations in Python and Matlab.
Prior Progress

1. Created a fully customizable general ADMM function:
   - Convergence checking of proximal operators.
   - Multiple types of stopping conditions.
   - Over/under relaxation.
   - Complete run-time information.
   - Accelerated and Fast ADMM

2. Created library of solvers for problems ADMM is used for:
   - **Constrained Convex Optimization**: Linear and Quadratic Programming.
   - **$l_1$ Norm Problems**: Least Absolute Deviations, Huber Fitting, and Basis Pursuit.
   - **$l_1$ Regularization**: Linear SVMs, LASSO, TVM, Sparse Inverse Covariance Selection, Logistic Regression.
Prior Progress (continued)

3. Testing and validation software for ADMM and solvers:
   - **For ADMM**: general solver (simple quadratic model) to test on.
   - **For solvers**: tester functions. Set up random problems and solve them, knowing the “correct” solution.
   - **Batch tester** to run solvers over a problem size scaling function.

4. Adaptive Step Sizes:
   - Tried several interpolation + 1D least squares methods:
     1. On Ye and Huan’s \( w = [x^T, z^T, u^T]^T \) values.
     2. On Esser’s \( \phi \) and \( \psi \) based residual.
   - Step sizes tended to explode.
Further Progress

- **File organization and setup routines:**
  - With solvers, testers, and other files, about 30 programs.
  - Organized into subfolders containing solvers, testers, examples.
  - Nifty routine to automatically setup paths no matter what file is run.

- **Code Restructuring:**
  - Streamlined solver code.
  - Added different algorithms do some solvers.
  - Prepped all code for parallel implementation.

- Implemented local, parallel capabilities into ADMM to use all cores efficiently.
Decomposition In ADMM

- Suppose function $f$ is *separable* in $x = (x_1, \cdots, x_n)^T$; then:

$$f(x) = f_{n_1}(x_{n_1}) + \cdots + f_{n_m}(x_{n_m}), x = (x_{n_1}, \cdots, x_{n_m}), \sum_{i=1}^{m} n_i = n$$

- Can decompose the proximal operator for $f$.

- Thus, our $x$-minimization step in ADMM is split into $m$ separate minimizations that can be carried out in parallel:

$$x_{i}^{(k+1)} := \text{prox}_{f_{n_i},\rho}(x_{n_i}, z, u)$$
Parallel Updates

Can use this observation to parallelize $x$-updates in ADMM.

No reason this can’t be done for $g$ as well!
- We often use simple $z$-updates (soft-thresholding, projections)
- Can be updated component-wise, or block component-wise.

For the $u$-update: $u^{(k+1)} := u^{(k)} + (Ax^{(k+1)} + Bz^{(k+1)} − c)$
- Can compute $\hat{x} = Ax^{(k+1)}$ and $\hat{z} = Bz^{(k+1)}$ by similar parallel computation.
- Clearly can update $u^{(k+1)}$ component-wise:

$$u_i^{(k+1)} := u_i^{(k+1)} + (\hat{x}_i + \hat{u}_i − c_i)$$

Note that update chunks $n_i$ can differ between $x$, $z$, and $u$. 
Implementing Parallel Updates

- Interpret user provided proximal operators for $f$ or $g$ as component proximal operators:
  - Normally given function $\text{prox}_f(x,z,u,rho)$.
  - Change to $\text{prox}_f(x,z,u,i)$ (Vector variables passed by reference)

- Instead of looping over $i$, distribute workload to workers (processors) in each update.

- User provides slices $(n_1, \cdots, n_m)$ for every update they wish to parallelize as acknowledgement to perform Parallel ADMM.

- In MATLAB, all this is easy to do for local parallel processes:
  - Use parfor loop over $i$ on each parallel update ($x$ or $z$).
  - Most matrix/vector operations already distributed among workers.
Example: LASSO Problem

- Standard LASSO formulation: \( \min_x (1/2 \| Dx - s \|_2^2 + \lambda \| x \|_1) \)
- ADMM form: \( \min (f(x) + g(z)) \) subject to \( x - z = 0 \), where \( f(x) = 1/2 \| Dx - s \|_2^2 \) and \( g(z) = \lambda \| z \|_1 \).
- \( L_\rho(x, z, u) = f(x) + g(z) + (\rho/2) \| x - z + u \|_2^2 - \text{const}(u) \)
- Proximal operator for \( f \) is \( x \) such that:

\[
\nabla_x (L_\rho(x, z, u)) = D^T(Dx - s) + \rho(x - z + u) := 0
\]

- Update step: \( x := (D^T D + \rho I)^{-1}(D^T s + \rho(z - u)) \)
- Update for \( z \) can be parallel (soft-thresholding). What about the \( x \) update?
Parallel LASSO

- Slice up rows of $D$ and $s$ into $i$ chunks $\{D_{ni}\}$ and $\{s_{ni}\}$:
  \[
x_{ni} := (D_{ni}^T D_{ni} + \rho I)^{-1}(D_{ni}^T s_{ni} + \rho(z_{ni} - u_{ni}))
  \]

- Consensus update $z = S_{\lambda/(\rho N)}(\bar{x} + \bar{u})$.

- In both serial and parallel LASSO, cache Cholesky factorizations ($X = Y^T Y$) of matrix to invert and solve the system for updating.

- Parallel preprocessing:
  - You need to factor and store each chunk’s decomposition.
  - Solution: Add parameter to options struct, `options.preprocess`, a function handle to local preprocessing function in user’s program.
LASSO Serial vs. Parallel: $2^{11}$ Rows, $2^3$ Columns

**Serial**
- True objective value: 1.0466
- ADMM's objective value for $x$: 1.0431
- ADMM's $(x,z)$ objective value: 1.0431
- Relative error in $x$ objectives: 0.0051898
- Number of iteration steps: 12
- ADMM Runtime: 0.13514 seconds.
- Solver Runtime: 0.70649 seconds.

**Parallel**
- True objective value: 1.0466
- ADMM's objective value for $x$: 1.0464
- ADMM's $(x,z)$ objective value: 1.0459
- Relative error in $x$ objectives: 0.0021078
- Number of iteration steps: 1
- ADMM Runtime: 0.03516 seconds.
- Solver Runtime: 0.85629 seconds.
LASSO Serial vs. Parallel Thorough Test

Run-time comparison of regular LASSO vs. parallel LASSO.
Transpose Reduction

- Want more efficient parallel $x$-update for skinny matrix $D$, which is typical.
- Note: $\frac{1}{2} ||Dx - s||^2_2 = \frac{1}{2}x^T(D^TD)x - x^TD^Ts + \frac{1}{2}||s||^2_2$
- Now, a central server needs only $D^TD$ and $D^Tb$. For tall, large $D$, $D^TD$ has much fewer entries.
- Note that: $D^TD = \sum_i D_i^TD_i$ and $D^Tb = \sum_i D_i^Tb_i$.
- Now each server need only compute local components and aggregate on a central server.
- Once $D^TD$ and $D^Tb$ are computed, solve with ADMM.
Problem statement: \( \min_z (g(z)) \) subject to \( z = Dx \).

ADMM form: \( \min(f(x) + g(z)) \) subject to \( Dx - z = 0 \), where \( f(x) = 0 \), and \( g(z) \) is the same.

Define the pseudoinverse of \( D \) as \( D^+ = (D^T D)^{-1} D^T \)

As \( f(x) = 0 \), proximal operator for \( f \) is simply \( x \) such that:

\[
\nabla_x \left( \frac{\rho}{2} \| Dx - z + u \|_2^2 \right) = D^T (Dx - z + u) := 0
\]

which is simply \( x = (D^T D)^{-1} D^T (z - u) = D^+ (z - u) \).

Cache \( D^+ \). For separable function \( g \), can parallelize \( z \) update.

Can we parallelize \( x \) update?
Unwrapped ADMM With Transpose Reduction

- Abuse Transpose Reduction: slice $D$ into $D = [D_1^T \cdots D_N^T]^T$.
- Then update $x = D^+(z - u) = W \sum_i D_i^T (z_i - u_i)$, where $W = (\sum_i D_i^T D_i)^{-1}$.
- Note that for skinny $D$, $W$ is very small and linear system solve much cheaper!

In distributed setting, can:

1. Store $D_i^T D_i$, $D_i^T$, $z_i$ and $u_i$ on each machine.
2. Have central server compute and cache $W$.
3. Central server adds up $d_i = D_i^T (z_i - u_i)$ into sum $d$ and computes $Wd$.

In local, parallel settings, can:

1. Do everything distributed does, but locally.
2. Compute summations in parallel.
Example: Linear SVMs

- General Form: \( \min \left( \frac{1}{2} \|x\|^2 + C h(Dx) \right) \), \( C \) a regularization parameter. \( D \) is training data, with \( \ell \) the training labels.
- “Hinge loss” function: \( h(z) = \sum_{k=1}^{M} \max(1 - \ell_k z_k, 0) \).
- Unwrapped ADMM can solve this problem, even for 0-1 loss.
- For hinge loss: \( z^{k+1} = Dx + u + \ell \max(\min(1 - v, C/\rho), 0) \)
- For 0-1 loss: \( z^{k+1} = \ell \mathbb{I}(v \geq 1 \text{ or } v < (1 - \sqrt{2C/\rho})) \)
- Here, \( v = \ell (Dx + u) \)
- Can use both parallel and serial Unwrapped ADMM, as \( z \) update is a component-wise computation.
- Perform parallel sums and preprocessing using options.preprocess.
Solution: Serial vs. Parallel SVM for $2^9$ Rows
Serial vs. Parallel: Thorough Test

Runtime Comparison of Regular Unwrapped ADMM vs. Parallel With Transpose Reduction

- Red line: Two-core Parallel ADMM
- Blue line: Two-core Regular ADMM
Serial vs. Parallel: Thorough Test
Final Stretch

- Need to finish code restructuring on about 4 solvers.
- Need to add parallel versions a few more solvers.
- Test and validate everything using testers.
- Documentation.
- Write final report.
High Performance Computing

- We have an efficient parallel implementation of ADMM. Can take full advantage of all cores on a machine.
- Would like a distributed version:
  - Parallel ADMM allows for distributed computing.
  - Distribute to many machines, then use all cores with clever parfor usage.
  - Optimize my solvers for big data.
- Looking into MatlabMPI to do this.
  - Distributed computing via MPI-like behavior.
  - Potential to completely automate ADMM usage for big data.
Adaptive Step-Sizes

- Previous attempts at adaptive step-sizes had issue of blowup of stepsizes.
- Restarting at detection of blowups negates this. Tends to improve convergence regardless of starting stepsize.
- Drawback: no theoretical support for this.
- **Future work:** adaptive stepsizes with strong theoretical support and better results.
A Better Software Library

- Create general solvers and group more specific ones under them:
  - Streamlines code.
  - General solvers more useful for users.

- More solvers:
  - Need solvers for consensus and sharing problems.
  - Need more distributed solvers for big data.

- More user friendliness, examples, and documentation. Want this to be a base for future ADMM research.
References


Thank you! Any questions?