Performance Estimation of the Alternating Directions Method of Multipliers in a Distributed Environment

Johan Mathe - johmathe@stanford.edu

June 3, 2011

Goal

We will implement the ADMM global consensus algorithm with the MapReduce framework and compare the resulting performance with another distributed optimization technique. We will also investigate various caveats with this implementation.

Problem Definition

The supervised machine learning problem we are solving here is a classification problem, *i.e.* mapping a text to an ontology concept (*e.g.* classification of news articles into various categories - business, politics, *etc.*). We solve it by minimizing an L^1-regularized sum of loss functions, each one associated with a training example. This is a convex problem, and is defined in (1). m is the number of training examples, $l_k(x) \in \mathbf{R}$ the loss function for training example $k = 1, \dots, m$. $x \in \mathbf{R}^n$ is the feature vector we want to predict. $\lambda \in \mathbf{R}$ is the L^1 regularization parameter. Here we skip the details associated with the multinomial logistic regression loss function, as it is thoroughly studied in the literature [KCFH05]. We use L^1 regularization as a heuristic to obtain a sparse vector.

$$\underset{x}{\text{minimize}} \sum_{k=1}^{m} l_k(x) + \lambda \|x\|_1 \tag{1}$$

This problem is separable. We split it in N of subproblems in (2). $p = \frac{m}{N}$ is the number of training examples per subproblem. $x_i \in \mathbf{R}^n$ is the local (primal) optimization vector for subproblem i. In (3), we convert this problem to an ADMM global variable consensus problem with L^1 regularization. $z \in \mathbf{R}^n$ is the global consensus variable.

$$L_i(x_i) = \sum_{j=ip}^{(i+1)p} l_j(x_i) \quad i = 1, \dots, N \tag{2}$$

$$\begin{array}{ll} \underset{x_i,z}{\text{minimize}} & \sum_{i=1}^{N} L_i(x_i) + g(z) \\ \text{subject to} & x_i - z = 0 \;\; i = 1, \ldots, N \end{array} \tag{3}$$

Let's consider ADMM under its scaled form. Here we have $u_i \in \mathbf{R}^n$ a local dual vector. $S_\kappa(x)$ is the soft thresholding operator with threshold κ. We define $\forall i = 1, \ldots, N$ (in parallel) the ADMM iteration in (4).

$$\begin{array}{rcl} u_i & := & u_i + x_i - z \\ x_i & := & \underset{x_i}{\operatorname{argmin}} \left(L_i(x_i) + (\rho/2) \left\| x_i - z + u_i \right\|_2^2 \right) \\ z & := & S_{N\rho}(\bar{x} + \bar{u}) \end{array} \tag{4}$$

In order to check convergence, we compute primal and dual residual norms (see the stopping criterion defined in [BPC+11]).

Implementation

We used the MapReduce framework [DG04] for the distributed implementation of the problem. MapReduce lets the user define the Map and Reduce functions inspired from functional programming. It takes care of the scheduling operations, the input/outputs and the routing of the data between the mappers and the reducers. The Map and Reduce functions (seel algorithms 2 and 3) have to be idempotent and must be stateless. In order to work around this, we will be using the Bigtable distributed storage system [CDG+06] in order to keep subproblem states between one iteration and the next. Bigtable can be seen as a storage system allowing low latency access of a sparse matrix, with key-indexed rows and multiple columns. In our implementation, we will store one subproblem state per row. There will be only one column, and every cell will contain three vectors: $x_i \in \mathbf{R}^n$, $u_i \in \mathbf{R}^n$ and $\hat{z} \in \mathbf{R}^n$ (see algorithm 2 and 3 for their definition). The datasets representing the examples are stored in a distributed file system [GGL03]. We have a method checking convergence at each iteration, returning a boolean value: true if convergence is attained, false otherwise.

The general approach is inspired from [BPC+11]: a master program starts various iterations of MapReduce, and at each iteration, checks if convergence is reached by reading values from Bigtable (see algorithm 1 for details).

The Map and Reduce functions [DG04] take care of solving each ADMM subproblems. Most of the optimization work is done in the mapper function, as shown in algorithm 2. The reduce function takes care of the scatter-gather operation by receiving the primal and dual vectors from each mapper (respectively $x_i \in \mathbf{R}^n$, $u_i \in \mathbf{R}^n$), compute the update for $\hat{z}$, and write them to Bigtable.

A simplified view of the pipeline is given in figure 1.

Algorithm 1 Main control loop algorithm

```
1: converged := false
2: while not converged do
3:     RunMapReduceIteration()
4:     $x_i, u_i, \hat{z}$ := ReadFromBigtable()
5:     converged := HasConverged($x_i, u_i, \hat{z}$)
6: end while
```

Algorithm 2 Map function

```
1: function Map($D_i$)
2: $x_i, u_i, \hat{z}$ := ReadFromBigtable(i)
3: $z := S_{N\rho}((1/N)\hat{z})$
4: $u_i := u_i + x_i - z$
5: $x_i := \mathrm{argmin}_x(f_i(x) + (\rho/2)\|x - z + u_i\|_2^2)$
6: Emit(key CENTRAL, record $(x_i, u_i)$)
```

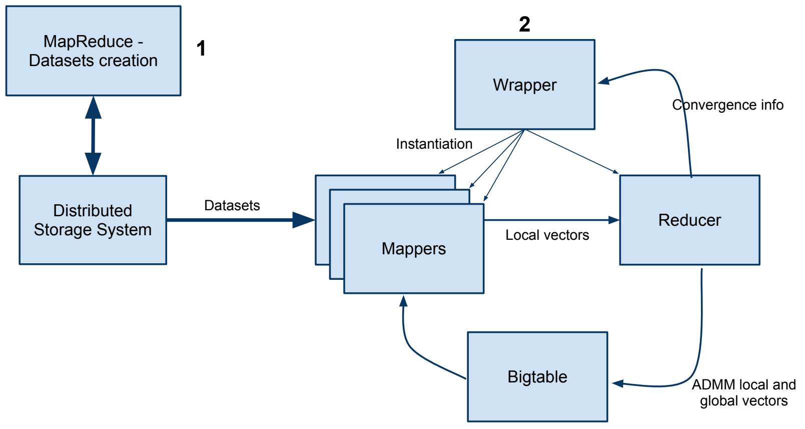
MapReduce - Datasets creation
1
2
Wrapper
Convergence info
Instantiation
Distributed Storage System
Datasets
Mappers
Local vectors
Reducer
Bigtable
ADMM local and global vectors

Figure 1: Data flows of our implementation.

Implementation Caveats

Training sets $D_i \in \mathbf{R}^{p \times n}$ have to be invariant at each iteration, therefore they need to be defined before the ADMM iterations start. Here since the input datasets are distributed, we need a MapReduce program to create them. Its implementation is straightforward: let the Map function be the identity. We then set the number of output shards to N, the number of datasets. Each reducer accumulates the values, and once all values are accumulated, we flush them with the reducer's id as dataset label. Given a good sharding function and enough examples, we get good size distribution.

Algorithm 3 Reduce function

```
1: function Reduce(key CENTRAL, records $(x_1, u_1), \ldots, (x_N, u_N)$)
2: $\hat{z} := \sum_{i=1}^{N} x_i + u_i$
3: for all $i$ such that $i < N$ do
4:     $S$ := SerializeToBigtableCell($x_i, u_i, \hat{z}$)
5:     WriteToBigtable($i$, $S$)
6: end for
```

Method	Success	CPU time	Memory	Net	Time	Shards	Iterations
ADMM	0.8309	46 h	57.34 GiB.h	29.9 GiB	92 min	200	44
Dist. Grad.	0.8407	14 h	28.33 GiB.h	31.95 GiB	17 min	50	50

Table 1: Results of our algorithm

Another caveat is that the original implementation advised in [BPC+11] does not scale because one single Bigtable row has a limited size. Our approach is then to write every subproblem information per Bigtable row, with label i as its key. This also leads to more efficient distribution of the problem data.

The biggest caveat we faced was that a basic warm start is not possible. Indeed, when starting to close to the optimal point, the region between the optimal point and the starting point is too flat, which leads to a lack of curvarture and ill-conditioned Hessian matrix (see Wolfe conditions in [ZNB+92]).

Results

We tested our algorithm with a 560MiB training Dataset (4M examples, 30K features). We split our problem into 200 subproblems. Each subproblem is a 20K $\times$ 30K datapoints size problem. Training vectors are spares (example density was 0.01%). We evaluate our result vector with a testing set T. We define success as the ratio of properly predicted examples over total number of examples in T (0: all predictions are wrong, 1: all predictions are correct).

We present here various results we obtained with our implementation. First, figure 2 shows convergence of our algorithm over time. We observe that our stopping criterion is met after 44 iterations (using stopping criterion from [BPC+11] with $\epsilon_{\text{abs}} = 10^{-4}$ and $\epsilon_{\text{rel}} = 10^{-2}$). Results in terms of CPU, memory, network usage are presented in table 1. Table 1 also compares our implementation with the distributed gradient algorithm, which has lot of similarities with our implementation (It uses a distributed implementation with MapReduce, as defined in [HGM10]). The classical distributed gradient implementation outperforms our implementation, in terms of memory, CPU time and wall clock time. We perform better for the network usage. Values stay in the same order of magnitude.

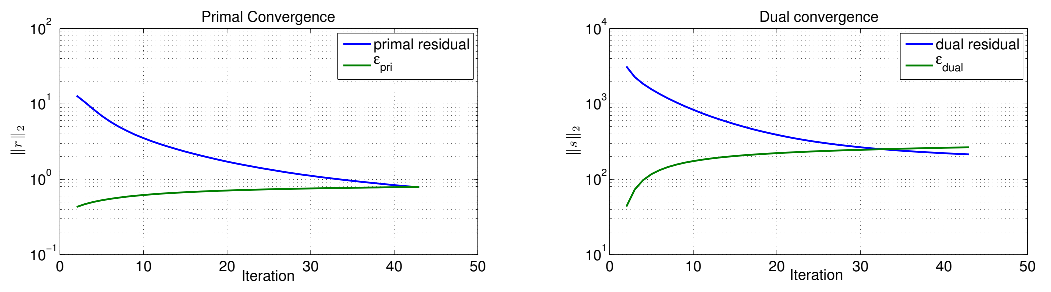

Figure 2: Convergence of our ADMM Implementation.

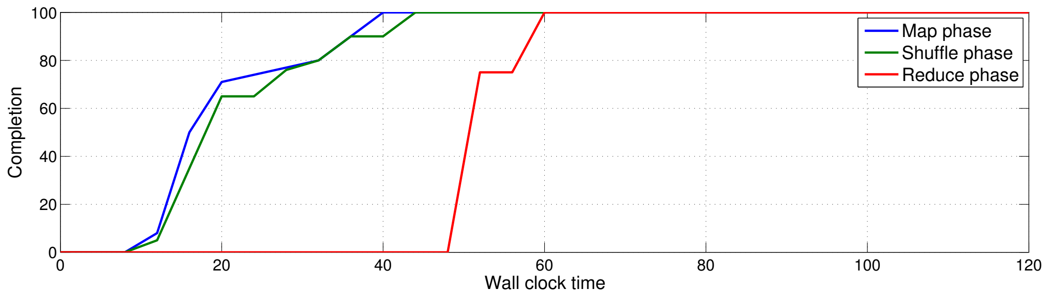

Figure 3: Phases of a single MapReduce iteration.

Figure 3 and table 2 show various results for a single iteration. Figure 3 shows the progression of the Map, Shuffle and Reduce [DG04] phases of a single iteration. It is interesting to notice that half of the wall clock time is spent in the Reduce phase, mostly I/O bounded by the Bigtable write operations. We will discuss ways to fix this in the next section.

In order to explore the effect of the values of ρ on the convergence of our algorithm, we tried various values on a logarithmic scale from 1 to 10^6. As seen on figure 4, small values of ρ ($< 10^2$) inhibit the effect of L^2 regularization producing erratic dual convergence. Small values of ρ also lead to poor primal residual convergence. Big values of ρ ($> 10^4$) lead to poor dual residual convergence. A good tradeoff here is to take $\rho = 5000$, this worked well with our experiment.

Improvements and Optimization

From the results we can see multiple possible optimizations. Here we will focus on two dimensions for the optimization: wall clock time and CPU usage. In order to reduce the wall

CPU time	Input data	Memory	Reduce Input	Time
3731 s	619 MiB	5161 GiB.s	31.2 MiB	122 s

Table 2: Footprint of a single MapReduce iteration.

clock time of the algorithm, we can reduce the time spent in each iteration. Figure 3 tells us that half of the time is spent in the reduce phase at each iteration. This is clearly I/O bounded: most of the time is spent in writing local variables to Bigtable. We can either use MapReduce combiners or use another MapReduce cycle to distribute this operation across multiple machines. The other MapReduce would write all results to Bigtable in the Map phase, whereas the reducer would just aggregate all data and write the averaged z vector.

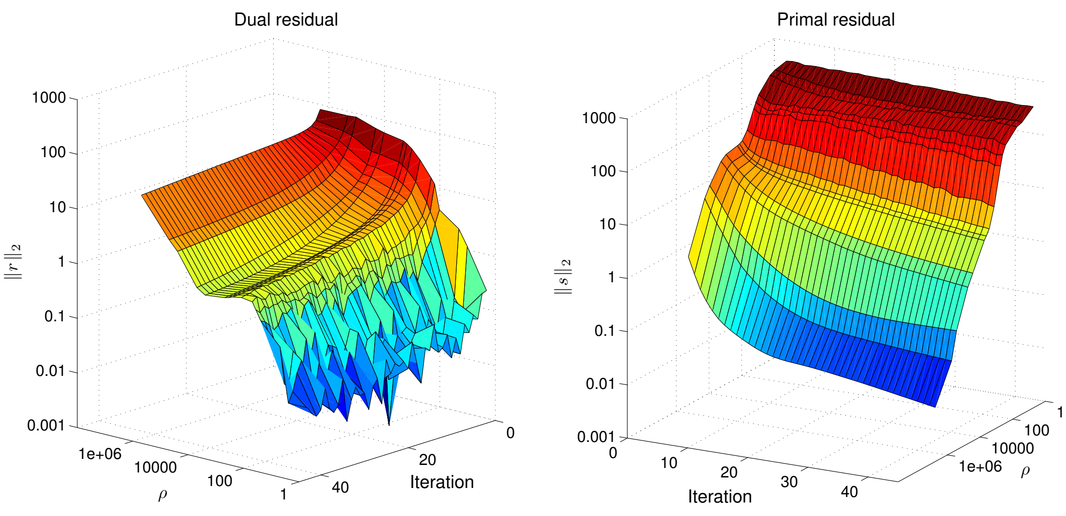

Figure 4: Dual and Primal residuals *vs* ρ

In order to reduce CPU usage, but also wall clock time, we need to fix the warm start issue defined in the caveats section. Indeed, as seen in [BPC+11], warm start can reduce the number of L-BFGS iterations by a factor of 5.

We are confident that these optimization could bring the total iteration phase from 122 seconds to approximately 40 seconds (10 for map and reduce initialization, 10 for the map phase and 20 for the reduce phase). This would bring the total time for our experiment from 92 minutes to approximately 34 minutes (taking into account the first iteration).

Conclusion

We successfully implemented the ADMM consensus problem for a multinomial L^1-regularized multinomial logistic regression problem with the MapReduce framework. We found that setting an appropriate ρ is primordial for a reasonable convergence time, and that setting ρ to a small value prevents L-BFGS to converge because of a lack of L^2 (Tikhonov) regularization. Global performance of our implementation is not as good as a direct distributed gradient method, but it is in the same order of magnitude. Further optimizations like warm start, the use of MapReduce combiners and adding another MapReduce cycle will greatly improve performance, both in terms of CPU time and wall clock time.

References

[BPC$^+$11] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 2011.

[CDG$^+$06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[HGM10] Keith Hall, Scott Gilpin, and Gideon Mann. MapReduce/Bigtable for distributed optimization. In *Advances in Neural Information Processing Systems Workshop on Learning on Cores, Clusters and Clouds*, 2010.

[KCFH05] Balaji Krishnapuram, Lawrence Carin, M?rio A.T. Figueiredo, and Alexander J. Hartemink. Sparse multinomial logistic regression: Fast algorithms and generalization bounds. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:957–968, 2005.

[ZNB$^+$92] X. Zou, I. M. Navon, M. Berger, K. H. Phua, T. Schlickii, and F. X. Le Dimet. Numerical experience with limited-memory quasi-newton methods and truncated newton methods. *SIAM J. Optimization*, 1992.