CS 224W Final Report: Community Detection for Distributed Optimization

Tri Dao trid@stanford.edu
Rolland He rhe@stanford.edu
Zhivko Zhechev zzhechev@stanford.edu

1 Introduction

Distributed optimization has a large range of applications that make use of massive amounts of data or require limited communication between computing nodes. Therefore, it is important to investigate efficient ways of performing distributed optimization. One of the more popular methods in this area is consensus optimization using the Alternating Direction Method of Multipliers (ADMM). Since a large family of AI and machine learning problems reduces to solving large-scale convex optimization problems, the ADMM algorithm plays a key role in tackling these problems. For example, many supervised learning models under the Empirical Risk Minimization paradigm such as Lasso, Group Lasso, Generalized Linear Model, Logistic Regression, and SVM require minimizing a convex objective function. Running a linear regression on billions of training examples naturally translates to solving a convex problem with billions of terms in the objective function, and applying a distributed method to perform this optimization can considerably reduce the amount of time needed for this task. Moreover, once the scale of the problem becomes too large, solving the problem on a single machine is not even feasible because of memory limitations. In this project, we propose and examine a method to split an arbitrary convex optimization problem into smaller subproblems that can be solved efficiently in a distributed manner using the ADMM algorithm.

2 Background and Problem Formulation

2.1 ADMM and Consensus Optimization

In the ADMM algorithm, as presented by Boyd et al. [2], the problem is assumed to split into separate additive components, or *subproblems*. These subproblems are only connected to each other via equality linear constraints. The general form of such a problem is called the *consensus form*:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{N} f_i(x_i) \\ \text{subject to} & x_i - z = 0, \quad i = 1, \ldots, N. \end{array} \tag{1}$$

Other constraints can be included by adding indicator function terms in the objective function. In particular, any constraint $x_i \in S_i$ where S_i is a convex set is equivalent to adding the term $\mathbf{1}_{S_i}(x_i)$ to the objective, with $\mathbf{1}_{S_i}(x) = 0$ if $x \in S_i$ and $\mathbf{1}_{S_i}(x) = +\infty$ otherwise. Therefore we can always assume that the optimization problem only has linear constraints $x_i - z = 0$ for $i = 1, \ldots, N$.

The terms $f_i(x_i)$ in the objective are separable, allowing each worker to minimize each subproblem separately. In the ADMM algorithm, the updates per iteration (for $i = 1, \ldots, N$) are:

$$x_i^{(k+1)} := \underset{x_i}{\arg\min} \left(f_i(x_i) + {y_i^k}^T (x_i - z^{(k)}) + (\rho/2)\|x_i - z^{(k)}\|_2^2 \right)$$

$$z^{(k+1)} := \frac{1}{N}\sum_{i=1}^{N} x_i^{(k+1)}$$
$$y_i^{(k+1)} := y_i^{(k)} + \rho(x_i^{(k+1)} - z^{(k+1)}).$$

Intuitively, in each iteration, each worker separately solves a local optimization problem, with parameter ρ, to compute its estimate $x_i^{(k+1)}$ of the public variable $z^{(k+1)}$. They then send these estimates to the master solver, who updates the public variable by averaging these estimates. The master solver then sends this public variable value back to each worker to update their local dual variables $y_i^{(k+1)}$. This procedure is repeated until convergence (i.e. until the public variable does not change very much across iterations and each worker's estimate approximately agrees with the value of the public variable).

2.2 Term Splitting and Problem Formulation

Any convex optimization problem can be transformed into the consensus form through *variable replication*. For example, to minimize $f(x) + g(x)$, we introduce variable z and constraint $z = x$ to reformulate the problem as: minimize $f(x) + g(z)$ subject to $x = z$.

In general, there is usually more than one way to split the terms. For problems that are highly structured or for small problems, term splitting can be done by hand. However, this is clearly not feasible for larger problems, which can have upwards of tens of thousands of variables. The rate of convergence depends on several factors, one of which is the splitting scheme of the original problem into subproblems. Term splitting is an important step that has not been addressed in [2], even though it can significantly affect the rate of convergence. For example, with a suboptimal split, the workers might share many variables, which will result in longer time for them to agree on the values of those variables. Another problem caused by suboptimal splitting is unequal allocation of work to each worker, since the time taken per iteration is the time taken by the slowest worker.

We aim to develop a method to automatically transform a general convex problem into the consensus form in (1). In particular, we will use some community detection heuristics to automatically split the terms in the objective, and consequently reduce the running time of the distributed optimization algorithm.

3 Modeling

3.1 Factor Graph

We now explain how we approached the splitting problem. The objective function is a sum of convex terms $T_1, T_2, \ldots, T_m$, each of which is a function of a subset of the variables $x_1, x_2, \ldots, x_n$. We want to partition the set of terms into disjoint groups, where the sum of the terms in each group will form a subproblem that will be assigned to a worker. Terms in different groups will in general share variables, so we will apply variable replication as explained in Section 2.2 to separate the groups. The ADMM algorithm needs to synchronize the values of these replicates; hence, it will converge in fewer iterations if the objective function terms can be split to reduce the number of shared variables between terms in different groups.

We model this problem using a factor graph: the nodes in the graph are the terms in the objective function (i.e. $T_1, T_2, \ldots, T_m$) and the factors in the graph will correspond to the variables $x_1, x_2, \ldots, x_n$. The factor corresponding to the variable x_i is a function of the terms $T_1, T_2, \ldots, T_m$ in which x_i occurs. Figure 1 shows an example of such factor graphs.

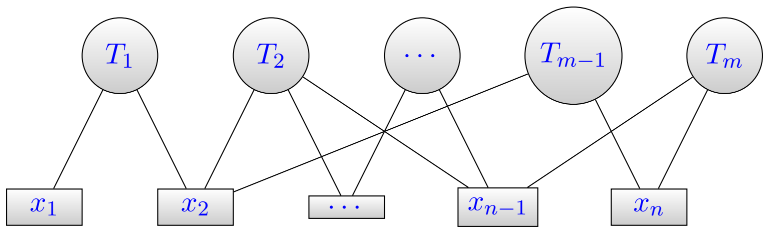

Figure 1: An example factor graph.

Recall that we want a partitioning of the terms $T_1, T_2, \ldots, T_m$ in which few variables appear across different groups. Therefore, we want the weight associated with the factor corresponding to x_i to be large when all terms in which x_i appears are in the same group, and small when x_i appears in terms which fall in different groups. The goal is then to maximize the product of the weights associated with all the factors, which will coincide with minimizing the number of shared variables between different groups.

Let us formalize this model. Suppose we want to partition the set of terms into N clusters (corresponding to the number of machines or the number of CPU cores). For any variable x_i, the corresponding factor is a function of the terms in which x_i appears. Let those terms be $T_{i_1}, T_{i_2}, \ldots, T_{i_l}$. We define the weight of factor to be

$$w(T_{i_1}, T_{i_2}, \ldots, T_{i_l}) = \frac{1}{\text{Number of distinct groups that contain } T_{i_1}, T_{i_2}, \ldots, T_{i_l}}.$$

Every term T_j in our factor graph takes value in $\{1, 2, \ldots, N\}$, which specifies the group in the partitioning we assign the term T_j to. The weight of the factor at x_i is then inversely proportional to the number of distinct values in the set $\{T_{i_1}, \ldots, T_{i_l}\}$.

3.2 Graph Partitioning

Next, we show how we heuristically convert the problem of maximizing the product of weights in our factor graph (which is NP-hard) into a graph partitioning problem, which is an instance of a community detection problem. Recall that in the graph partitioning problem, we are given an undirected graph and we try to partition the nodes into a specified number of clusters so that we minimize the number of edges cut.

We construct the graph for the graph partitioning problem given our factor graph in the following way: the nodes in the new graph are still going to be the terms $T_1, T_2, \ldots, T_m$, which were the terms in our objective function. Next, for every factor f defined on the set of nodes $(T_{s_1}, T_{s_2}, \ldots, T_{s_r})$, we construct a clique of the nodes $T_{s_1}, T_{s_2}, \ldots, T_{s_r}$ (i.e. we draw every possible edge). We let each edge have the same cost $\frac{1}{\binom{r}{2}} \frac{1}{w(T_{s_1}, T_{s_2}, \ldots, T_{s_r})}$. Note that in principal we can have a multigraph, since two terms can share more than one variable. However, we can always collapse all the edges between two nodes into a single edge, with an edge weight equal to the sum of all the original edge weights, to produce a normal weighted graph with at most one edge between any two nodes.

This defines a (weighted) undirected graph between the nodes $T_1, T_2, \ldots, T_m$. Our task is then to partition the nodes into N clusters, so that the sum of the weights of the cut edges is as small as possible. Note that we set the cost of an edge to be inversely proportional to the factor value $w(T_{s_1}, T_{s_2}, \ldots, T_{s_r})$ because the goal of the graph partitioning problem is to minimize the cost of the edges cut, while the goal of the variable assignment in a factor graph is to maximize the product of the weights. Moreover, we choose to take the sum of the costs rather than the product for simplicity.

We have thus explained how the original problem of splitting the terms in the objective can be transformed into a graph partitioning problem. The final graph is simply the graph with nodes representing the terms in the objective function and with an edge between two nodes if and only if the terms corresponding to the nodes share a variable.

3.3 Simple Example

To illustrate the problem in concrete detail, consider the following simple example: suppose we have the optimization problem

$$\text{minimize} \quad f(a,b) + g(b,c,d) + h(c,d),$$

where f, g, h are convex functions, and we want to split them into 2 subproblems. This can easily be done by hand. There are a few reasonable splits, with f and g in one partition and h in the other, or f in one partition and g and h in the other. However, let us examine how to proceed using the framework of factor graphs and graph partitioning. We can create the bipartite factor graph by drawing nodes for f, g, h on one side and factors a, b, c, d on the other, followed by connecting terms that contain a particular variable:

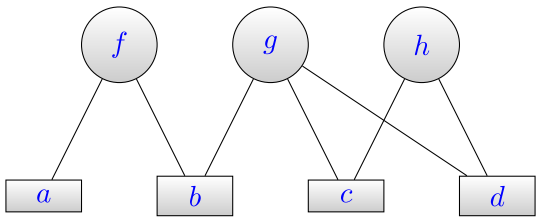

Figure 2: Factor graph for the example.

As explained in Section 3.2, we can transform this into a graph partitioning problem. In this graph, we note that f and g share 1 variable, whereas g and h share 2. Thus, there would be 1 edge between the nodes representing f and g and 2 edges between the nodes representing g and h. Since a multigraph can be represented as a weighted graph, we can draw the graph as follows:

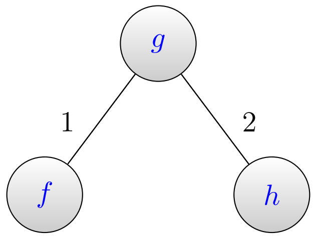

Figure 3: Graph of the terms f, g, h for the example.

In the above graph, we want to split the graph in a way that minimizes the total cost of edges cut. Since there are only 2 possible splits in this case (either a cut on the edge connecting f and g or on the edge connecting g and h), we can easily see that the cut between f and g is optimal. This cut represents a partitioning that puts f into a subproblem and combines both g, h into the other subproblem.

After deciding on the cut, we replicate the variable b (the only shared variable, which corresponds to the only edge cut) to make the objective separable. The problem becomes

$$\begin{array}{ll} \text{minimize} & [f(a, b_1)] + [g(b_2, c, d) + h(c, d)] \\ \text{subject to} & b_1 = b_2. \end{array}$$

This form is suitable for the consensus solver that will make use of two workers (running on two different CPU cores or two different machines), one to solve the subproblem involving $f(a, b_1)$ and the other to solve the subproblem involving $g(b_2, c, d) + h(c, d)$. The consensus solver will combine the results and continually update the parameters until convergence to solve the original problem.

4 Implementations

4.1 Term Splitting via Community Detection

We implement this automatic term splitting scheme atop CVXPY [8], a Python-embedded modeling language for convex optimization problems. From the abstract syntax tree of the convex problem expressed in CVXPY, we form a graph of terms of the objective function and the shared variables between them. We represent the terms of the objective function in the convex optimization problem as vertices and the shared variables as edges. We then partition this graph into N pieces of roughly equal size, with the minimum number of edges cut between pieces.

We have also implemented a graph partitioning scheme based on different community detection algorithms, namely graph coarsening, spectral clustering, the Girvan-Newman algorithm, and the Clauset-Newman-Moore algorithm. We used the implementation of the graph coarsening algorithm provided by METIS [14], and the implementation of the Clauset-Newman-Moore algorithm provided by Snap.py [15]. Our splitting scheme based on these algorithms handles both constrained and unconstrained problems. We describe the details of these methods in Section 5.

4.2 Consensus Solver

We have improved the consensus solver by Dao and Wang [7] that can solve any `cvxpy.Problem` instance after it has been split into subproblems. All the necessary information (number of shared variables, number of iterations, time taken per iteration) is stored in a `Multiproblem` object. The solver uses the ADMM algorithm to solve these subproblems on a multi-core machine and is used to test the effectiveness of a splitting scheme.

5 Community Detection Algorithms

5.1 Graph Coarsening

Graph coarsening is a multi-level graph partitioning method that takes in a weighted graph and returns a partitioning of the nodes that attempts to minimize the sum of edge weights cut [14]. For the purposes of our method, we have defined the edge weights as the number of shared variables between 2 nodes; when 2 terms share many variables, and therefore require a lot of communication, we want to avoid cutting the corresponding edge if possible. Graph coarsening has 3 main phases:

1. *Coarsening phase.* A sequence of successively smaller graphs $G_1, G_2, \ldots G_n$ are constructed from the original graph G_0. The graph G_{i+1} is computed by finding the maximal matching M_i of G_i and collapsing the adjacent vertices of the matching together (a maximal matching of a graph G is the set of edges with maximum total weight such that each edge is in G and no 2 edges share a vertex). The edges of the new collapsed vertex will be the union of all the

edges incident to the original 2 nodes u, v, excluding edge (u, v). If an edge in G_i is incident to both u and v, then the edge weight of the corresponding edge in G_{i+1} will be the sum of the original 2 edge weights.

2. *Partitioning phase.* In the partitioning phase, we seek to find the minimum N-cut of the coarse graph (where N is the number of clusters). The edges and edge weights reflect the edges of the original graph G_0, so this partition serves as a crude approximation to a minimum N-cut for G_0.
3. *Uncoarsening phase.* The partition is projected back through $G_{n-1}, G_{n-2}, \ldots, G_1$, where at each iteration, the uncollapsed nodes are assigned to whichever partition their corresponding collapsed nodes were assigned to in the previous iteration. In addition, since the original partition was a crude approximation to the minimum N-cut for G_0, the partition is further refined at each iteration by finding a perturbation of the partitions that decreases the total sum of the cut edge weights.

The algorithms that implement each of these phases is discussed in more detail in [14].

5.2 Spectral Clustering

We can approximate the sparsest cut of a graph (into two pieces) through the second eigenvalue of its Laplacian matrix, which can be computed efficiently (in $O(n^3)$ time, where n is the number of terms, which can be much smaller than the problem size). For the general case of splitting to N pieces, we use the approach presented by Ng et al. [16]. We look at the N largest eigenvalues of the Laplacian matrix and run the K-means algorithm on those to split the nodes into N clusters.

Given a simple graph $G = (V, E)$ with n vertices $1, \ldots, n$, its Laplacian is $L = D - A$, where D is the degree matrix diag(deg(1), . . . , deg(n)) and A is the adjacency matrix of the graph. The Laplacian has the quadratic form

$$x^T L x = \sum_{(i,j)\in E} (x_i - x_j)^2.$$

Thus L is a positive semidefinite matrix, and its smallest eigenvalue is 0, with the eigenvector $v_1 = \begin{bmatrix} 1 & \ldots & 1 \end{bmatrix}^T$. Assuming that G is connected, we can use the following algorithm to partition G into N clusters:

1. Compute the Laplacian matrix L of the graph
2. Compute the N largest eigenvectors of the Laplacian matrix $v_1, v_2, \ldots, v_N$ and construct matrix $V = \begin{bmatrix} v_1 & \ldots & v_N \end{bmatrix}$.
3. Normalize the rows of V such that each row has norm 1. Call this new matrix V'.
4. Treat each row of V' as a point in $\mathbf{R}^N$ and cluster these n points into N clusters using the K-means algorithm.
5. Assign the i-th point to cluster j if row i in V' is assigned to cluster j.

5.3 Girvan-Newman Algorithm

The Girvan-Newman (GN) algorithm [11] is a common method to perform community detection using edge betweenness. Edge betweenness is defined as the number of shortest paths between any 2 vertices that pass through the edge, normalized by the total number of shortest paths. The algorithm is as follows:

1. Compute the betweenness for all edges in the network.
2. Remove the edge with the highest betweenness.
3. Repeat the first 2 steps until some predetermined stopping criteria is satisfied.

In our implementation, we stop when the graph splits into two connected components.

5.4 Clauset-Newman-Moore Algorithm

The Clauset-Newman-Moore (CNM) algorithm [5] attempts to maximize the modularity of a graph using a greedy approach. The algorithm is as follows:

1. Start by setting each individual vertex to be a cluster.
2. Join the 2 clusters that provide the greatest increase in modularity.
3. Repeat step 2 until no joining of clusters increases the modularity.

6 Results and Analysis

We test the effectiveness of our splitting schema by comparing them to two random splits based on the number of iterations and the time taken for the ADMM algorithm to complete. Here, we analyze the results for three convex optimization problems and report the results for seven other problems.

6.1 Network Lasso

Hallac et al. [12] have recently developed a framework which solves a generic convex optimization problem defined on a graph, which can be applied to a variety of problems. More concretely, the convex problem under investigation can be formulated in the following way: given a graph $G = (V, E)$, weights w_{jk} for each edge $(j, k) \in E$, and a parameter λ,

$$\text{minimize} \quad \textstyle\sum_{i \in V} f_i(x_i) + \lambda \sum_{(j,k) \in E} w_{jk} \|x_j - x_k\|_2. \tag{2}$$

Here we have a variable $x_i \in \mathbf{R}^p$ and a convex function $f_i : \mathbf{R}^p \to \mathbf{R} \cup \{\infty\}$ for every node $i \in V$. Intuitively, we want to minimize the sum $\sum_{i \in V} f_i(x_i)$ but we also prefer the variables on adjacent nodes to be relatively close (or possibly even equal) to each other. Adding the second term $\lambda \sum_{(j,k) \in E} w_{jk} \|x_j - x_k\|_2$ to the objective function in (2) acts as a regularizer which penalizes differences between variables at adjacent nodes. The hyperparameter λ determines how aggressively we want to force the adjacent variables to be equal.

Let us briefly summarize a machine learning example presented in the paper which motivates the consideration of the convex problem above. Suppose we are interested in predicting the prices of houses in some geographic region. We are given information about the geographical location of each house (latitude/longitude) as well as a few other features such as number of rooms, number of bathrooms, house area, etc. A standard machine learning algorithm here would be a linear regression which puts weights on each of these features. However, one could imagine that two houses with very similar feature vectors may have very different prices because, for example, one of them is situated near a good school whereas the other one is not. Even though the linear regression has access to the latitude and longitude for each house, it might not have access to the quality of the nearest school. However, if we cast the problem in the convex form (2), the solver will cluster the houses in neighborhoods as a side effect of solving the problem. When the algorithm converges, the models for houses in the same neighborhood will be similar. As a result we not only solve our regression problem, but we implicitly get a clustering of the houses.

In this section, we consider a simplified network lasso problem whose graph is a dumbbell graph. This is a somewhat contrived example to show the extent to which a good splitting scheme can speed up the computation of the optimal point. The functions f_i in (2) are chosen to have the form $\|A_i x_i - b_i\|^2$ for some matrices A_i's and vectors b_i's. From the formulation of the problem in (2), we see that only neighboring nodes share variables, so the graph representation of the convex problem defined in Section 3.2 is isomorphic to the graph on which the problem is defined in the first place. In particular, the graph representation of the convex problem is a dumbbell graph.

Suppose we have $n = 10$ nodes in the graph, separated into two groups of 5 nodes each. The (dumbbell) graph representing the problem is shown in Figure 4. The nodes and edges represent the terms in the objective and the shared variables between those terms, respectively. Black edges represent variables that are shared between the two subproblems (obtained after partitioning), while red and blue edges represent variables internal to the first and the second subproblems. Comparing the cut produced by our variable splitting scheme, say by the spectral clustering method (called the *spectral cut*), and a random cut, we can clearly see that the spectral cut results in much fewer shared variables between the 2 partitioned subproblems.

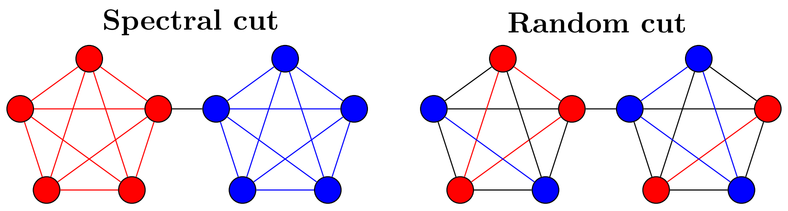

Figure 4: Graphs representing sum of squares problem, with spectral cut and random cut.

We plot the convergence of the objective value produced by running ADMM on problems obtained from different splitting schema in Figure 5. We use several different methods for splitting including random splitting as a baseline.

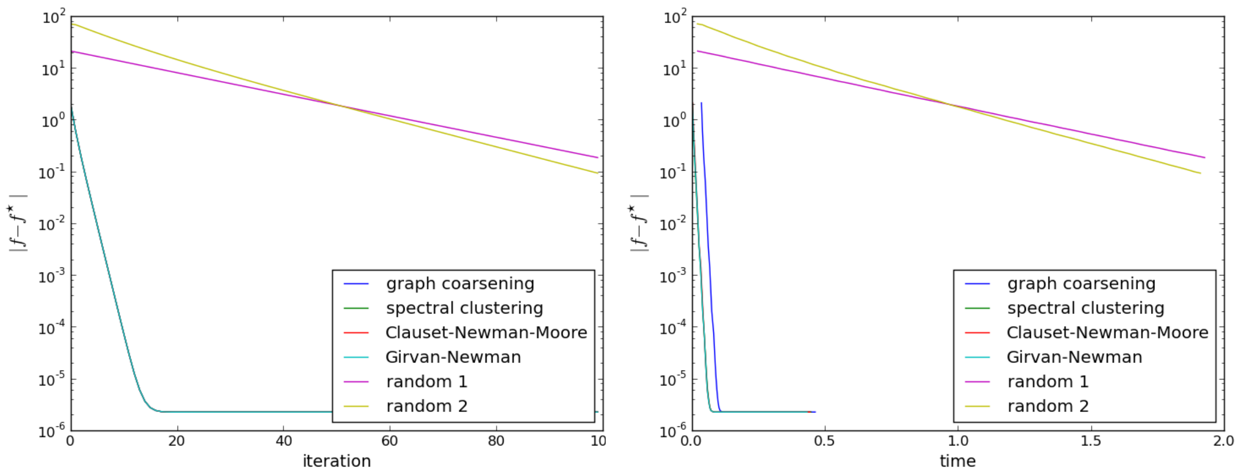

Figure 5: Number of iterations and time (in seconds) comparison between different splits.

Note that in this simplified example, all of the splitting techniques we used (graph coarsening, spectral clustering, Clauset-Newman-Moore, Girvan-Newman) produce the same (optimal) splitting since the good split is quite obvious by design and is easily found by all the splitting techniques. As a result, their lines coincide on the figure. The random splits, however, do not perform as well, since they generally produce suboptimal splits. Similarly, in terms of running time, all of our heuristics for the splitting run in virtually the same amount of time (with some small difference due to noise during the execution), whereas the random ones are again suboptimal. This example shows that a good splitting scheme can achieve a speedup of up of up to 5-10 times compared to random splitting.

6.2 Routing

The previous example was engineered just to show that good splitting techniques can noticeably improve the performance of the solver, but the example itself is not necessarily practical. Here, we consider a standard flow routing problem [1], a more realistic example in which splitting is important for performance. In this flow control problem, we have a routing matrix $R \in \mathbf{R}^{m\times n}$, where $R_{ij} = \mathbf{1}\{\text{flow } j \text{ passes over link } i\}$, and a capacity vector $c \in \mathbf{R}^m$. The utility optimization problem to be solved is:

$$\begin{array}{ll} \text{maximize} & U(f) = \sum_{i=1}^{n} U_j(f_j) \\ \text{subject to} & Rf \leq c, \quad f \geq 0. \end{array}$$

We take U to be the concave function $U(x) = \sqrt{x}$. Intuitively, each link (i.e. edge) has some capacity that the flows have to respect and each flow of commodity produces some concave utility function proportional to the amount of commodity routed through that flow. We aim to maximize the total utility of all the flows.

We compare the plots of the convergence of the objective function value versus iteration and versus execution time (in seconds) for a few splitting methods in Figure 6. Since this is a constrained problem, each iteration of ADMM does not generally produce a feasible solution, and so the objective value might increase beyond $f^\star$. Thus, we plot the absolute value of $f - f^\star$ in order to get a better picture of the convergence. Note that the values $|f - f^\star|$ appear to fluctuate as the iterations alternate between feasible and infeasible solutions to the original problem.

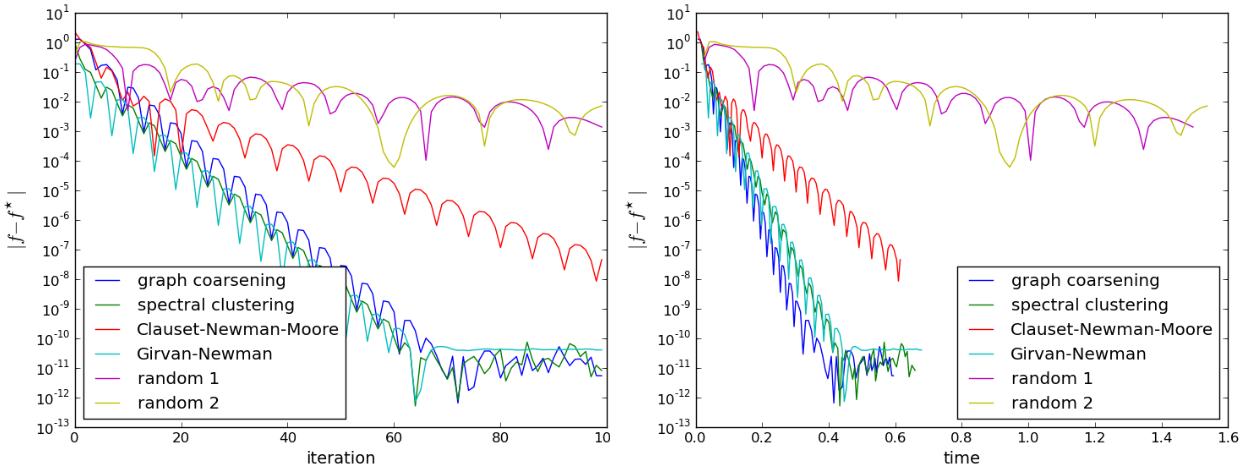

Figure 6: Objective function error values $|f - f^\star|$ versus iterations and time for the routing problem.

In both plots, we observe the same pattern: graph coarsening, spectral clustering, and Girvan-Newman perform very similarly to each other, and they do better than Clauser-Newman-Moore, which in turn outperforms the random splits. Note that the speed up in terms of execution time is much more impressive than the speed up in terms of number of iterations. This is because if the subproblems are not well-split, one might take much longer to solve than the other, and hence each iteration of ADMM will take longer to run.

6.3 Group Lasso

Group Lasso [10] is another problem that arises in practice which we considered in our experiments. The problem definition is as follows: we have L different predictors $\beta_1, \beta_2, \ldots, \beta_L$, L matrices

$X_1, X_2, \dots, X_L$ which store the data of the problem, and a response vector y. Let $p_1, \dots, p_L$ be the sizes of the groups respectively. Then the problem is:

$$\text{minimize} \quad \|y - \textstyle\sum_{l=1}^{L} X_l \beta_l\|_2^2 + \lambda \sum_{l=1}^{L} \sqrt{p_l} \|\beta_l\|_2,$$

with variables $\beta_1, \dots, \beta_L$. Note that we can rewrite this problem equivalently as:

$$\text{minimize} \quad \textstyle\sum_{i=1}^{n} \left(y_i - \sum_{l=1}^{L} X_l^{(i)} \beta_l \right)^2 + \lambda \sum_{l=1}^{L} \sqrt{p_l} \|\beta_l\|_2.$$

where $X_l^{(i)}$ denotes the i-th row of the matrix X_l. In this form, we have more terms in the objective function, which allows the distributed algorithm to parallelize more computations. For this reason, we used the latter formulation of the problem in our experiments. We ran our solver using all the splitting techniques we have discussed on a problem with $L = 4$ groups and where each p_l a random integer between 1 and 5, and the X_l are matrices of size $60 \times p_l$ (so there are 60 observations in total) filled with random numbers drawn from a standard Gaussian distribution. We also set $\lambda = 1$. The convergence plots (in terms of number of iterations and time in seconds) for different splitting techniques are given in Figure 7.

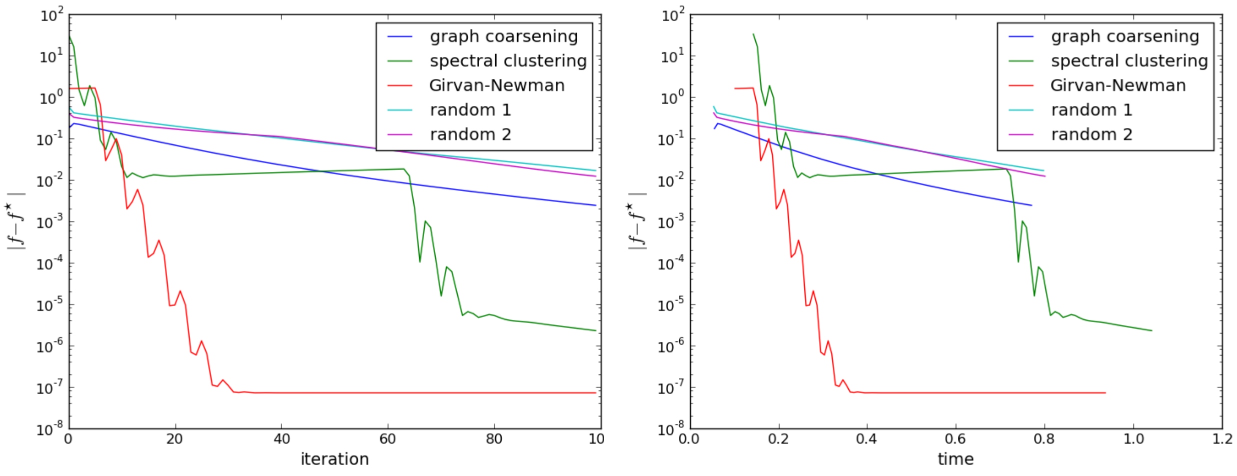

Figure 7: Objective function error values $|f - f^\star|$ versus iterations and time for the routing problem.

We excluded the Clauset-Newman-Moore splitting method from the plot because it did not produce a split of the graph with the desired number of partitions. Among the other methods, we observe that Girvan-Newman performs the best both in terms of number of iterations and running time. The spectral clustering comes second in both metrics, followed by the graph coarsening. The two random splits which serve as a baseline perform the worst both in terms of the number of iterations and in terms of the running time.

6.4 Other convex problems

We measured the performance of our splitting methods on a wide variety of convex optimization problems. We include here the descriptions of some problems besides the previous three. These problems appear in the context of statistical estimation, machine learning, and finance, as presented in [9]. The data for each problem is randomly generated.

1. *Basis pursuit.* This problem [4] seeks the smallest vector in the ℓ_1-norm sense that satisfies a set of under-determined linear equality constraints. The objective has the effect of finding a sparse solution. It can be stated as

$$\begin{array}{ll} \text{minimize} & \|x\|_1 \\ \text{subject to} & Ax = b. \end{array}$$

2. *Lasso.* This problem [17] seeks to perform linear regression under the assumption that the solution is sparse. The ℓ_1 penalty in the objective encourages sparsity. The problem can be stated as

$$\text{minimize} \quad \|Ax - b\|_2^2 + \lambda \|x\|_1.$$

3. *Logistic regression.* This problem [13] fits a probability distribution to a binary class label. It can be stated as

$$\text{minimize} \quad \textstyle\sum_{i=1}^m \left(\log(1 + \exp(x^T a_i)) - b_i x^T a_i\right),$$

   where $b_i \in \{0, 1\}$ is the class label of the ith sample and a_i^T is the ith row of A.
4. *Linear program.* This problem [1] minimizes a linear function subject to linear inequality constraints:

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & Ax \leq b. \end{array}$$

5. *Non-negative least squares.* This problem [3] seeks a minimizer of the least squares problem subject to the solution vector being non-negative. This comes up in applications where the solution represents real-world quantities. The problem can be stated as

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_2 \\ \text{subject to} & x \geq 0. \end{array}$$

6. *Portfolio optimization.* Portfolio optimization or optimal asset allocation seeks to maximize the risk adjusted return of a portfolio. A commonly used model is the k-factor risk model [6], which assumes that the return covariance matrix can be expressed as the sum of a diagonal matrix plus a rank k matrix.

$$\begin{array}{ll} \text{maximize} & \mu^T x - \gamma x^T (FF^T + D)x \\ \text{subject to} & x \geq 0, \quad \mathbf{1}^T x = 1. \end{array}$$

   where $F \in \mathbf{R}^{n \times k}$ and D is diagonal.
   The constraint $x \geq 0$ indicates that the portfolio is limited to long positions only. We can also have a variant of the problem with a limit on the leverage $\|x\|_1$ and the risk instead:

$$\begin{array}{ll} \text{maximize} & \mu^T x - \gamma x^T (FF^T + D)x \\ \text{subject to} & \mathbf{1}^T x = 1, \quad \|x\|_1 \leq L_{\max}, \quad x^T (FF^T + D)x \leq R_{\max}. \end{array}$$

We include in Table 1 the results for the graph coarsening and the spectral clustering methods. The speedup is measured against two random splits of the problem. The reason we did not include the CNM method is that the implementation by SNAP does not always partition the graph into a specified number of parts. The GN algorithm on the other hand often produces splits in which one part is much smaller in size compared to the other one (depending on the structure of the graph), which does not match the behavior we want. Hence we do not include the result for the GN algorithm either.

	Graph coarsening		Spectral clustering	
Problem	Iteration speedup	Time speedup	Iteration Speedup	Time speedup
Basis pursuit	0.63–1.41	0.61–1.42	1.76–3.93	1.49–3.47
Group Lasso	1.01–1.03	1.22–1.48	6.97–7.11	5.69–6.86
Lasso	1.12–1.28	1.11–1.32	1.19–1.44	1.09–1.3
Least squares	1.12–1.28	1.14–1.24	1.42–1.62	1.2–1.32
Logistic regression	0.84–0.88	0.93–0.93	1.15–1.21	1.01–1.01
LP	1.31–1.82	1.13–1.45	0.97–1.35	0.91–1.17
Network lasso (dumbbell)	15.93–16.8	35.99–55.37	15.93–16.8	37.44–57.59
Routing	2.02–2.13	3.49–4.41	38.5–40.5	34.12–43.06
Portfolio (long only)	1.03–1.32	0.93–1.37	1.07–1.37	0.99–1.46
Portfolio (leverage limit)	0.66–1.08	0.85–1.58	1–1.62	1.36–2.51

Table 1: Speedup achieved by the graph coarsening and spectral clustering methods.

The heuristic algorithms for splitting the different graphs we are considering generally work better than the baseline random split algorithm. Furthermore, for most problems we examined, spectral clustering performs better than graph coarsening. These two methods (especially spectral clustering) significantly improve the solving time compared to the random splits in problems defined on graphs (such as the routing problem and network lasso) or problems with sparse graph representations (such as group lasso).

However, our splitting method does not perform very well in problems with dense structures, such as logistic regression or least squares, because the graph representation of the problem is a complete (or nearly complete) graph. Each term in the objective represents a data point, which contains the same weight vector. In such cases, an alternative approach is to split the terms in the objective based on the properties of the data points instead of based on the graph structure of the problem. If we can develop a good heuristic for how to partition the data points, the local weight vectors obtained from each worker solving each subproblem corresponding to the different data point groups will be similar; thus, we gain statistical power from the large amount of data. Exploring such heuristics is an interesting direction for future work.

7 Conclusion

We have designed and implemented a method to transform an arbitrary convex problem into the consensus form by splitting the terms in the objective. This method reduces the number of iterations and time required to solve the convex problem in a distributed manner using the ADMM algorithm.

We modeled the splitting task using a factor graph and as a graph partitioning problem, and constructed the graph representing the convex problem with terms in the objective function as vertices and shared variables as edges. We then implemented different splitting heuristics based on various community detection algorithms: graph coarsening, spectral clustering, the Girvan-Newman algorithm, and the Clauset-Newman-Moore algorithm. We tested the effectiveness of these heuristics by measuring the number of iterations and time required for the consensus solver to solve a variety of convex optimization problems.

For dense problems, the graph structure is close to that of a complete graph, and so these splitting heuristics do not result in any substantial speedup. Our heuristics are most effective for problems with sparse structures or problems defined on graphs, since we can exploit such structures to produce a good split that reduces the solving time by 5 to 30 times compared to a random split.

Note

Both our CS 221 and CS 224W projects are on this topic of distributed convex optimization. In our CS 221 project, we focus on modeling the splitting task using factor graph and on applying unsupervised learning approaches to perform problem splitting. For our CS 224W project, we mainly use community detection to determine an optimal partitioning of terms.

Acknowledgment

We thank Professor Stephen Boyd for introducing us to convex optimization and ADMM and Professor Percy Liang for his insightful suggestions. We also thank Yushi Wang for his help in writing the consensus solver.

References

[1] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[2] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2011.

[3] D. Chen and R. J. Plemmons. Nonnegativity constraints in numerical analysis. In A. Bultheel and R. J. Plemons, editors, *The Birth of Numerical Analysis*, pages 109–140. World Scientific, 2009.

[4] S. S. Chen, D. L. Donoho, and M. A. Saunders. Atomic decomposition by basis pursuit. *SIAM Journal on Scientific Computing*, 20(1):33–61, 1998.

[5] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70, 2004.

[6] G. Connor and R. A. Korajczyk. The arbitrage pricing theory and multifactor models of asset returns. *Handbooks in Operations Research and Management Science*, 9, 1993.

[7] Tri Dao and Yushi Wang. Consensus optimization with automatic variable splitting. *Convex Optimization II (EE 364B) Class project*, 2015.

[8] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research.* URL `http://stanford.edu/~boyd/papers/pdf/cvxpy_paper.pdf`. To appear.

[9] Christopher Fougner and Stephen Boyd. Parameter selection and pre-conditioning for a graph form solver. *arXiv preprint arXiv:1503.08366*, 2015.

[10] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. A note on the group lasso and a sparse group lasso. *arXiv preprint arXiv:1001.0736*, 2010.

[11] M. Girvan and M.E.J. Newman. Community structure in social and biological networks. *Proc. Natl. Acad. Sci.*, 99:8271–8276, 2002.

[12] David Hallac, Jure Leskovec, and Stephen Boyd. Network lasso: Clustering and optimization in large graphs. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 387–396, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3664-2. doi: 10.1145/2783258.2783313. URL `http://doi.acm.org/10.1145/2783258.2783313`.

[13] T. Hastie, R. Tibshirani, and T. Friedman. *The elements of statistical learning.* Springer, 2009.

[14] George Karypis and Vipin Kumar. Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[15] Jure Leskovec and Rok Sosič. Snap.py: SNAP for Python, a general purpose network analysis and graph mining tool in Python. `http://snap.stanford.edu/snappy`, June 2014.

[16] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, pages 849–856. MIT Press, 2001.

[17] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society*, pages 267–288, 1996.