ABSTRACT

The rise of ‘big data’ and large-scale machine learning has created an increasing need for distributed optimization. Dual descent and the alternating direction method of multipliers (ADMM) are prominent distributed optimization techniques, due to their simplicity and effectiveness. We focus on implementing two other optimization techniques—dual descent and the alternating direction method of multipliers (ADMM), and implement distributed versions of both for quadratic functions, linear class SVMs, and logistic regression. Since dual descent and ADMM are algorithms that use multiple nodes and communicate between them, they lend themselves to some networking techniques. In particular, we utilized the User Datagram Protocol (UDP) to facilitate this nodal and communication structure, a novel use of UDP in this area of research. Ultimately, we provide a package that easily allows for optimization problems with Dual Descent or ADMM over the provided families of functions, and hope that it can be extended with other functions and/or optimization techniques in the future.

1. INTRODUCTION

With the size of datasets increasing rapidly, as well as the fact that with more data, more complex models can be fitted, it is important to be able to process those datasets in an efficient manner. Recently, a lot of attention has been focused in this area—companies in the adtech (advertising technology) sector, as well as many other sectors are extremely interested in being able to handle large, distributed datasets in a way that actually takes advantage of the structure of the problem, as opposed to using generic, existing tools that are not well-suited in their current form to distributed optimization.

Before fully discussing the details of the implementation, relevant definitions and high-level concepts with regards to the mathematics behind dual descent and ADMM will be briefly covered in Section 2. Previous work will be covered in Section 3, which will include an introduction to a distributed Coordinate Descent package that mirrors our implementation, as well as give credit to the theoretical ingenuity that led to the distributed version of dual descent and the alternating direction method of multipliers (ADMM). Note that we will only cover gradient methods in our distributed optimization library.

The main concern of this paper is to explain the motivation of as well as the implementation details of a distributed optimization framework that aims to have as little overhead as possible during execution, both in terms of wall time (real-world time measured by a clock on the wall) as well as extensibility. How we intend to achieve efficiency in both respects, as well as provide correct implementations, will be detailed in Section 5 as well as Section 6.

Preceding that, in Section 4, the system model will be given. The underlying infrastructure has been completed, pending beautification and refactoring, as well as the solving the symmetric positive definite quadratic case with dual descent.

In Section 7, we cover what remains to be implemented, namely extending our implementation to handle more families of functions, as well as using ADMM instead of dual descent.

2. BACKGROUND

In this paper, only two approaches to solving optimization problems will be outlined, for the sake of contrast. The first is gradient descent, which is the approach we used. Gradient descent uses information computed from the gradient at each iteration in order to descend in the direction of steepest change. The second approach is coordinate descent. With coordinate descent, the directions of descent are fixed from the outset, so each iteration is computationally less intensive than gradient descent. This generally takes many more iterations than gradient methods, which would mean much more communication overhead for a multi-agent (independent processors) distributed consensus (where each independent agent’s local solution converges to an agreed upon global one) optimization system.

Our package will deal with decomposable functions and solve them using gradient methods. This means that we will be optimizing over and solving functions of the form

\[
\text{minimize } \sum_{i=1}^{N} f_i(x). \quad (1)
\]

This is applicable in many machine learning scenarios, such as model combination. We will be focusing on this type of function specifically because of the fact that it is easily distributable.

The dual of a problem gives the lower bound on a solution of any convex optimization problem. In order to ‘solve’ a dual problem, we maximize the lower bound. Dual descent is where one finds the optimal solution of a problem in the dual domain, which is guaranteed to be less than or equal to the solution of the primal, if one exists. If strong duality holds, then the solution to the dual is equal to the solution
of the primal, so there is no duality gap.

The algorithms we will be applying, dual descent and ADMM, can both be distributed onto different computing nodes with different sets of data, as long as they have a method with which to communicate. Computing nodes can be different threads on the same computer, different computers in different locations, or computers in a cluster. In order to distribute decomposable functions, we project our variables into a higher dimensional space and find solutions for each node, under the constraint that they converge to the same solution. In order to accomplish this, the dual descent algorithm has two updates in each iteration, taken from Zargham et al. [5]. The algorithm we apply is called dual decomposition, simply because it is dual descent applied onto a function that can be decomposed into a sum of functions.

The primal update, given below in equation 2, computes the local optima.

$$x_i^t(\lambda^t) = \arg \min_{x_i} f_i(x_i^t) + x_i^t \left( \sum_{j \in n(i)} \lambda_{ij}^t - \lambda_{ji}^t \right)$$  \hspace{1cm} (2)

The dual update, given below in equation 3, ensures that all nodes converge to an agreement, where $\epsilon$ is the step size.

$$\lambda_{ij}^{t+1} = \lambda_{ij}^t - \epsilon(x_i^t - x_j^t)$$  \hspace{1cm} (3)

ADMM is another algorithm that can be used to solve the objective function given in equation 1 in a distributed manner.

$$\arg \min \hat{f}(x), \text{s.t. } Ax + Bz = 0$$ \hspace{1cm} (4)

In the above equation 4, we have defined $z$ to be the matrix representation of the network. We constrain each $x_i = z_{ij}$ and $x_j = z_{ji}$, represented by $\alpha_i$, and $\beta_j$, as seen in Boyd et al [1]. Boyd gives the primal variable update as:

$$x(k+1) = \arg \min_{x} \mathcal{L}(x, z(k), \lambda(k))$$ \hspace{1cm} (5)

Similarly, the $z$ updates are given by

$$z(k+1) = \arg \min_{z} \mathcal{L}(x(k), z, \lambda(k))$$ \hspace{1cm} (6)

The dual variable can be updated as in equation 7.

$$\lambda_i(k+1) = \lambda_i(k) + \epsilon \sum_{j \in n(i)} [x_i(k) - x_j(k)]$$ \hspace{1cm} (7)

We will not detail the convergence properties of the dual descent and ADMM algorithms, those can be found in [5, 1]. However, in order to demonstrate that an implementation successfully converges to the optimal solution, it is important to note that, given the formulation used to update the primal and dual variables, the solutions of each individual, independent node will converge to a global optima, given sufficient iterations and an appropriate step size.

Coordinate descent is, as mentioned before, different to gradient descent in that the direction of descent is fixed from the outset. A basis is chosen before proceeding, and then at each iteration, depending on the algorithm, a subset of those directions are chosen to perform line searches in. This allows for heavy parallelization and the use of randomization.

### 3. RELATED WORK

The algorithms themselves have already been produced, as seen in [5] and [1]. However, research so far has produced prototypes that are not in use in industry, and packages like CVX are not distributable. The final goal of our project, to reiterate, is to create a package that can be used for optimization in industry by those very familiar with convex optimization.

A very similar project is Hydra [4] and Hydra² [2], both by Peter Richtarik and Martin Takac. Rather than dual descent or ADMM, Richtarik and Takac used coordinate descent to solve the same three families of functions (squared loss, logistic loss, and hinge square loss) that we provide functionality for. The main difference is in the choice of their algorithm. In Hydra, a block coordinate descent algorithm was used, where only a certain number of variables were updated at each iteration, and each node was responsible for a projection (block, in this case synonymous with a subset) of the original variable vector. However, a large disadvantage of coordinate descent methods is the lack of convergence guarantees. The methods that will be used in this paper, the gradient-based descent methods dual descent and ADMM, have much stronger guarantees about finite-time convergence, which is the motivation behind implementing these algorithms in a package.

In contrast with our project, as well as the above project by Richtarik, an adtech company, Intent Media, designed a Hadoop implementation of the ADMM algorithm for logistic regression [3]. The Hadoop implementation also supports extensions to the functions it can handle. However, the motivation behind the Intent Media implementation was completely different. Rather than creating a solution with the minimum overhead in terms of efficiency, their goal was to craft a solution with the tools immediately available at the expense of efficiency. By using the Hadoop framework, their only choice was to aggregate all the primal variables in the reduce stage, and then at each iteration have one stage of the MapReduce repeat. The result is a large amount of overhead in each stage of the process.

### 4. SYSTEM MODEL

We present a a software package that can be used to run two distributed computational problems - Dual Descent and ADMM - in a distributed fashion. Larger instances of both problems generally take too much time to run on a single machine but are feasible on a cluster of machines. Running it on several, distributed machines is an important goal of our package, since similar software exists for running these types of problems on a single machine, whether single- or multi-threaded.

Dual Descent and ADMM both require a loss function to optimize, which is an important configuration parameter. This loss function is essentially the mathematical equation a user is trying to optimize, while the problem (Dual Descent or ADMM) is the method used to optimize it. The loss functions included are: Quadratic Loss (Least Squares), Logistic Loss (Logistic Regression), and Hinge Loss (Support Vector Machines). From here on, references to ‘function’ or ‘loss function’ refer to the these sorts of loss functions, while ‘problem’ or ‘method’ refer to a distributed computation problem such as Dual Descent.

The package is not limited in both of these areas. The fundamental structure of the package is conducive to adding new loss functions, and, although more complicated, adding more distributed computation problems. Though the package does not come with a vast array of options in terms
of loss functions or methods, users who need a specific loss function or version of an algorithm that is not available are able to program those specific pieces on their own, and then feed it into the package on their own with relative ease.

We anticipate two general use cases for our package. The first case involves users who wish to solve small instances of the distributed optimization problems, running it on a small cluster of around fifteen machines. This user is a researcher or developer who wants to experiment with the problem that they are trying to solve, perhaps trying different algorithms, parameters, or methods, and using our package to test out their hypotheses on a relatively small scale. This package will give this kind of user a means to easily distribute and compute solutions, and compare these answers to a metric that they wish to improve. This resembles the usage of Matlab and its various toolboxes, or SciPy for Python: both excellent tools for experimentation and validating proofs-of-concept, but mainly single-machine toolboxes. The key difference, however, between this package and the aforementioned ones is that this package is intended to be distributed from the start, whereas Matlab (for example) is generally used on a single machine. The primary advantage with this package is that it can be distributed easily across multiple physical machines, allowing for a wider variety of computation to be run easily. For this use case, a typical user could run a computation on a single machine using Matlab or SciPy on a small subset of their data, and tweak the parameters until they are satisfied; then, when this user wants to expand to the full set of data, this package is logically the next step in which they can run the full data set they may need to compute over, and test that the distributed result over the full set of data matches or does not match the trend they ran in the single-instance case. If it does not match, they can go back to the single-case or run this package on different parameters, perhaps on a different loss function. The essential point here is that this package fills a gap in the experimentation process, and allows for quick experimentation on a set of data that may be intractable or slow on a single machine.

A second use case is for going beyond the proof-of-concept stage and actually running a meaningful instance of one of the problems, in order to compute a real answer. This is the primary use case for our package; it is intended to make distributed optimization easier for Dual Descent and ADMM, which are generally more difficult than similar problems in this field, such as Coordinate Descent. We are designing the package with the target of industrial-strength usage in mind, and along the way, a package that can also be used for smaller-scale experimentation (as mentioned previously) will emerge. We have created the framework in such a way that it can still run on a large network of machines in a server room, and solve the given optimization problem with a given loss function. Though similar systems have been designed before, we believe that our approach in this matter is novel and more suitable for the algorithms we want to solve; for example, Dual Descent and ADMM. Both of these algorithms, when distributed, require a lot of communication, which makes using existing solutions such as MapReduce ineffective, as they are not tailored for multiple, small-sized updates. Our package wants to pursue a fundamentally different approach, by using the User Datagram Protocol, or UDP, which is used primarily in networking. The main benefit of UDP - the quick data transfer of a single packet of information - is also a large benefit to algorithms such as Dual Descent that rely on multiple, small updates over thousands of iterations. UDP is generally used as the underlying data transfer mechanism for real-time applications, such as audio and video streaming, voice over IP (VOIP), online multi-player video games, and other applications that require quick information. Similarly, UDP can also be used to handle the updates that are required for Dual Descent and ADMM, as they happen in real time, and do not contain a large amount of information. In both scenarios, large amounts of information are sent via small packets in short intervals. Though UDP has existed for decades, it has not been applied extensively to distributed algorithms as a communication mechanism (as we are applying it here).

Our system relies upon two essential components: single-node processing on each machine, and network communication of the results. Each node deals with these two parts separately, but both are required for any distributed algorithm: the single-node processing deals with the local piece of the problem on the current machine, while the communication outputs the result of the computation and inputs, as well as processes, the results of the other nodes’ computation so that it can perform the next step correctly. This communication component is performed via UDP.

In general, ADMM, Dual Desc, and similar algorithms work on a stepwise basis, computing a primal update, and then a dual update, and so on and so forth until an end condition is reached (for example, the value of the update does not change more than a small epsilon, indicating that the algorithm has converged to a single number within that epsilon). The actual updates that are computed is determined by the algorithm chosen by the user at the start of execution of the network. The code that runs the update runs decoupled from other parts of the system, including the other components on a single node; this allows for easier extensibility, and a clearer separation of concerns of the responsibilities of each node. Essentially, the update operates as a function that accepts a matrix as an input (or other form that constitutes the current ‘state’ that is important to this node), and outputs a vector of values.

Since the algorithm is distributed, at the end of each update, the node must communicate the update-vector it produces with other nodes to learn more information about the progress of the problem. Depending on the algorithm, a node may or may not need to receive the updated values in time for the next update, or may only require the updates from adjacent nodes rather than from the whole network. (These differing requirements depend on several factors, primarily concerning how synchronized or desynchronized the system’s communication is, and what convergence guarantees the algorithm itself has given a specific synchronization option). It takes several iterations to converge, and thus the communication happens often; thus, we try to minimize the cost of this communication. Additionally, we do not want to overload the network with messages, and thus must plan accordingly.

The following diagram summarizes this information. It represents both of the above procedures handled by two individual nodes connected together. Nodes 1 and 2 both have modules for computing the primal and dual updates, and a separate module for communication, as shown below. Each iteration can conceptually be broken up into four stages: (1)
1. Compute Primal
2. Update Primal
3. Compute Dual
4. Update Dual

Primal_1
Dual_1
Comm.1

Primal_2
Dual_2
Comm.2

Figure 1: An example of a two-node network (colored green and blue). Each node conceptually has three components, which are managed by a single instance of the optimization program.

computing the primal update, (2) communicating the primal update, (3) computing the dual update, and (4) communicating the dual update. These iterations are repeated until the algorithm converges.

5. SYSTEM IMPLEMENTATION

Our system is implemented using a variety of tools. We begin by describing the overall structure, and then elaborate on each particular piece of the structure.

There are three main pieces to the package, all of which currently run as command-line tools. These correspond to three distinct phases: a generation phase, an optimization phase, and a reporting phase. For the generation phase, there is a primary generator script, which takes in the choice of problem, the parameters of the problem/data, the loss function, and the desired network configuration. This script splits the problem up into initial pieces, and distributes a copy of the program and problem to each node, which is physically defined by a IP address and port number. The copy of the program is a compiled command-line binary will run on each node, which knows what piece of the data it is working on from an input file generated by the generator. This input file is then parsed by every node, and then they all begin in an initial state. A node consists of a computational program, an input file, an IP address, and a port number. This program is the same binary file for each nodes, and handles all three of the logical components in Figure 1.

This is the beginning of the optimization phase. Here, each node will compute iterations until reaching convergence. Each iteration consists of a primal update and dual update, both of which are run using a Haskell function with matrix operations, and then communicating the result of the update (usually a vector with double-precision values) with UDP packets to its neighbors. For this communication, each node is responsible for serializing and deserializing the primal and dual updates, respectively, and the computational program contains functions to perform these tasks and manage the sending and receiving of the updates.

Once it reaches convergence, we move to the reporting phase. The program then outputs the computational result as a simple text file. This text file describes a matrix: the first line are the row and column dimensions, and every line after that is a row of the matrix, where each entry is separated by a single space. This file is sent to the original generator and each file (one matrix file for each node) is put in a results directory.

A pictorial description of this process can be seen in Figure 2.

The input file has a certain specification. It begins with the description of the network, which is specified line-by-line in \((\text{node}, \text{IP:port})\) format. Then, following the IP addresses is a list of \((\text{node, node})\) pairs, indicating that the two nodes are connected. The entire network is essentially an undirected graph, and is stored using an adjacency list representation on the node itself. All nodes will be given a copy of this graph. The input file then describes the problem parameters. It will specify a number from a list of given problems (for example, 1 = Dual Descent, 2 = ADMM, etc.), followed by \(K\), the dimension of the problem. This is the dimension of the ‘answer’, which will be a \(K\times1\) vector. Following this will be a textual description of the input matrix, which represents the primary input to the problem and will be the main piece of the input that the computation is run on. The matrix is represented in the input file in the following format, where \(n\) and \(m\) are positive integers representing the dimensions of the matrix:

\[
\begin{array}{ccc}
n & m \\
\text{a11} & a12 & \ldots & a1m \\
\text{...} & \text{...} & \text{...} & \text{...} \\
an1 & an2 & \ldots & ann
\end{array}
\]

Each node will read in this input file, parse the contents into their appropriate representations in memory, and then initialize its state using the correct internal representations. From here, we use Haskell to manage the components of the overall system running on each node, and each node uses a socket to communicate with other nodes in the network using UDP packets. We make use of the hmatrix library, a library for efficient matrix and vector computation in Haskell. The primal and dual updates are also essentially run as Haskell functions using the same library. Since Haskell has a convenient representation of the Matrix and all of its parameters, it is simple and efficient to manage the data associated with it and manipulate it in memory.

These vectors then need to be transmitted to other nodes
Figure 2: An overview of the implementation, starting from top to bottom: the generation phase (green), the optimization phase (gray), and the reporting phase (yellow).
using UDP packets. First, however, the vector needs to be serialized into the UDP format, which Haskell is able to do relatively quickly and painlessly. Then, it will transmit the UDP packet using its outbound socket to other nodes on the network. It first needs to look up its neighbors in the adjacency list, then their associated IP addresses, and finally create a UDP packet containing the update and broadcast it to each of its neighboring nodes. The UDP packets are sent using Haskell functions that are essentially wrappers of the C socket library. We choose UDP because of its quick and efficient nature, and intend the serialization and deserialization costs to be fairly low, as these updates are communicated frequently.

While not part of the final implementation, it is important to analyze the correctness of the implemented algorithms and make sure they are working from a distributed sense. For these tests, we are utilizing Matlab and run simulations on small instances of each algorithm (with different network configurations, parameters, etc.) and analyzing the trends and values computed at each update step, to see if and how they are converging, and debug at that step. This decouples our testing environment of the algorithmic and theoretical properties of the algorithms we want to run from their actual implementation, and ideally, we want to create copies of the results from Matlab on our actual distributed implementation. Since Matlab has several plotting and analysis tools, it has a feedback-rich system that makes it convenient for testing purposes, so that we can eliminate problems in the algorithm and mathematical structure before we implemented them in our actual Haskell package which does not use Matlab.

6. RESULTS

Our system’s performed as expected. We include a graph of the quadratic program solved with dual decomposition and ADMM below in Figure 3.

As in the graphs, dual descent converged after around 1500 iterations, while ADMM took a third of the iterations, on average. We achieved similar results for logistic and hinge losses, although the graphs of their objective functions have been omitted since we do not have an optimal value that is calculable analytically.

The other results, given in Figure 4 give the real execution times of the different loss functions. The most surprising figure is the length of training an SVM. Since our implementation was in Haskell, it took too long to train with our own methods.

7. FUTURE WORK

Our current system is a framework to build upon. As such, extending this framework to handle more loss functions or optimization algorithms. One algorithm that would be interesting to extend our framework with would be the Network Newton [?], which was recently developed. More importantly, our framework currently only handles unconstrained optimization, so extending the current function library to handle more optimization problems would be very useful.

Another direction for future work would be to engineer a way in which the message passing within each iteration could be abstracted. Currently, the main body of the program is different depending on whether there are two exchanges of variables, or three. It would be an interesting task to leverage Haskell’s type system and features in order to refactor the code responsible for inter-nodal communication.

A third direction would be to develop a front end for the system. That would look like the CVX package for Matlab, where we allow the symbolic representation of a convex optimization problem, as well as the data. This would present a neat end-to-end solution for convex optimization problems in distributed settings.

A more tangential direction would be to port the framework to mobile devices. This could be an interesting application. Combined with encrypted messages, this could be used for something like calculating the best location from which to minimize the Euclidean distance to $N$ sensitive zones without aggregating the $N$ data points in one node.

8. ETHICS

This project primarily deals with the use of a distributed optimization package. As such, uses and misuses, whether intentional or unintentional, could lead to ethical concerns.

\begin{figure}[ht]
\centering
\includegraphics[width=0.8\textwidth]{objective_values.png}
\caption{The objective values over time for a quadratic program solved via Dual Decomposition and ADMM}
\end{figure}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Loss function & Time (Simulated) & Time (Distributed) \\
\hline
Quadratic & 0.6527 & 26.1079 & \text{500} \\
Logistic & 0.6685 & 35.4098 & \text{1000} \\
Hinge & 56 & - & \text{29} \\
\hline
\end{tabular}
\caption{A table of average run-times for the problems solved via ADMM. All problems were run on 11 nodes, with problems of size 11 by 11. The simulated times generated by running a Matlab simulation of the distributed algorithm. The distributed times were given by averaging each individual node. The real time for the hinge loss was not given because it was not run to completion.}
\end{table}
In particular, the way in which input and iteration data is managed by the package - through UDP - could be a potential issue, if it is leaked or exposed to a third party. This data could be private information and/or proprietary information. For example, consider a company with a contractual model, that collects data, either by itself or from other sources. Then, an employee loads this data into our package to compute an optimization internally, and finally offers suggestions to other entities based on this data, perhaps for a price.

There are two important stages which could cause vulnerabilities: the initial input data distribution, and the UDP primal and dual updates. The initial input data could contain sensitive information; in our example above, this is the step where the employee loads the data at the beginning of using the package. The initial input data is sent over SSH, which is relatively secure from attack, but is subject to vulnerabilities that SSH is; yet, SSH is generally considered secure. Additionally, the intention of the program is to be used on a ‘known’ cluster of machines, where the user creating an instance of a problem is aware of all the machines and the details of their connections between them. It is possible to even have them communicate entirely over ethernet and avoid connecting to the internet to communicate, provided the machines are all located in the same physical room.

The second vulnerability applies to the UDP primal and dual updates over UDP. The data could be leaked via sniffing the UDP packets, and this would compromise the economic value of the data, since the final iterations’ packets contain the ‘answer’ to the objective function, and an observer could steal this ‘answer’ by observing the final iterations. Yet, without large-scale computation, this is not sufficient to completely compromise the optimal solution, as an attacker would also need to know the initial conditions from the input data. This is because the optimization problem is computing a vector, $w^*$, that is then applied over the original data set. Technically, it is possible to infer the original data from just the dual and primal updates from the UDP packets, but this requires a large-scale computation to invert which goes much farther than simply sniffing.

Additionally, if the data were over some private information where the ‘end-result’ has been cleaned of sensitive data but not the intermediate steps (for example, advertisement data collection based on viewing history), the leaking of the intermediate data could cause concern for the data provider and the company.

These leaks are possible because of the package’s usage of UDP: the package can leverage the benefits of the networking protocol, but it is also subject to packet sniffing and familiar attacks on UDP. Nevertheless, there are some options for defense: for example, the communication packets can be encrypted in each iteration, and it is possible to use fast symmetric key encryption if one distributes it using a public/private key in the initial phase. Encrypting at each iteration would likely cause a decrease in running time performance, and it is not implemented as part of the current package, but it would solve the majority of the UDP-related issues.

9. CONCLUSION

We have created a framework for distributed consensus convex optimization. Although our choice of language was unconventional, it aided in the development process. However, since we relied on a linear algebra library written in Haskell, it slowed the execution time down. The overarching goal was achieved: an extensible framework for distributed optimization. The update functions themselves, as well as the linear algebra library, can easily be swapped out due to the modularity of the framework.

10. REFERENCES


