Large-scale Linear Optimization through Machine Learning

JungAh Hong
20143729

Jaeheung Surh
20111047

1. INTRODUCTION

Recent advances in cloud and distributed computing enabled Big Data processing. Today, processing big data generally falls under the category of statistical inferencing, e.g., predicting congestion with traffic data in the road network. The bottleneck is mainly the system I/O capacity for reading and writing data. The computation applied to individual items is relatively simple compared to the amount of data to process.

However, big data processing will ultimately require large scale optimizations in the future. For example, one might want to optimize the traffic light system of an entire city using predictions. Unfortunately, the traditional optimization algorithms are fundamentally hard to parallelize. These algorithms are hard to scale and cannot run on existing cloud environment.

In recent years, several abstractions using belief propagation (BP), a popular class of distributed machine learning algorithm, to parallelize linear optimizations have been proposed [2, 6]. Several well known optimization problems - the matching problem [4] and min-cost network flow problem [3] - were relaxed to a Linear Program (LP) then solved through BP in certain cases.

The goal of this research is to develop a practical system that runs on modern cloud environments, such as Amazon EC2, for parallel, distributed optimizations on maximum weight matching (MWM) problems. We plan to enhance recent theoretical developments on BP for large-scale linear optimizations and and develop a cloud-based software to solve MWM problems using the power of large-scale cloud computing.

2. BACKGROUND

2.1 Belief Propagation

Belief Propagation is a message passing algorithm for performing inference on graphical models. It is common used in artificial intelligence and information theory and has demonstrated empirical success in numerous applications.

On the graph, messages are associated on each edge with each direction and updated until convergence. Each node has a **belief**, which is calculated from incoming messages. Those beliefs are used as decision variable of nodes and edges.

2.2 Simplified Max-product for weighted matching [6]

Let $m_{i\to j}$ denote the messsage at time t, node i and edge $einE_i$. For every pair of neighbors i and j, let $e = (i, j)$ be the edge connecting the two, and define

$$a^t_{i\to j} = log(\frac{m^t_{i\to e}[0]}{m^t_{i\to e}[1]})$$

- **(INIT)** Set t=0 and initialize each $a^0_{i\to j} = 0$
- **(ITER)** Iteratively compute new messages until convergence as follows: ($y_+ = \max(0, y)$)

$$a^{t+1}_{i\to j} = max_{k\in N(i)-j}(w_{ik} - a^t_{k\to i})_+$$

- **(ESTIM)** Upon convergence, output estimate $\hat{x}_{(i,j)}$ is, respectively, $>$, $<$, or $= w_{ij}$.

It guarantees convergence when the extreme points of the matching LP polytope are integral. In other words, if LP doesn't have any fractional optima.

2.3 Hungarian algorithm [8]

The Hungarian algorithm is a combinatorial optimization algorithm that solves the assignment problem in polynomial time. The time complexity of the original algorithm was $O(n^4)$, modified to achieve $O(n^3)$ running time.

If we formulate the problem using a bipartite graph, $G = (S, T; E)$ with n vertices on one side (S) and same number of vertices on the other side (T), and each edge has a nonnegative weight $w(i, j)$, the way to find a perfect matching with minimum cost is as below. (Edges with 0 weight are added to make the graph complete in advance)

Let $y : (S \cup T) \mapsto \Re$ a **potential** if $y(i) + y(j) \leq c(i, j)$ for each $i \in S, j \in T$. The value of potential y is $\sum^{y(v)}_{v\in S\cup T}$. The Hungarian method finds a perfect matching of tight edges: an edge ij is called tight for a potential y if $y(i) + y(j) = c(i, j)$. Let G_y denote the

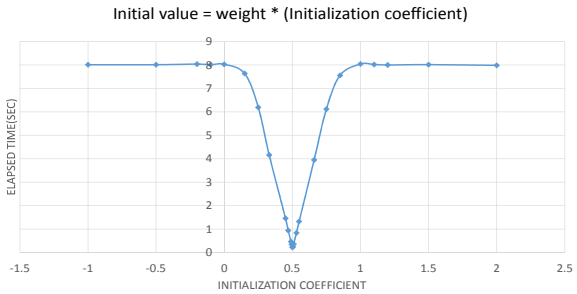

Figure 1: Convergence time gets shorter as the coefficient gets closer to 0.5

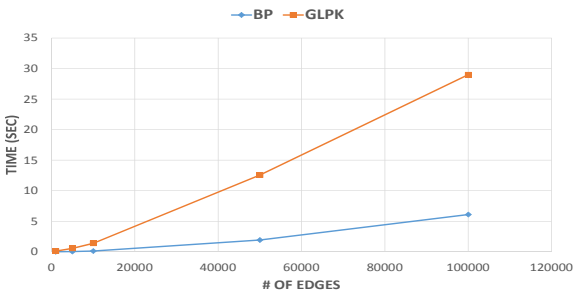

Figure 2: Convergence time of random dataset

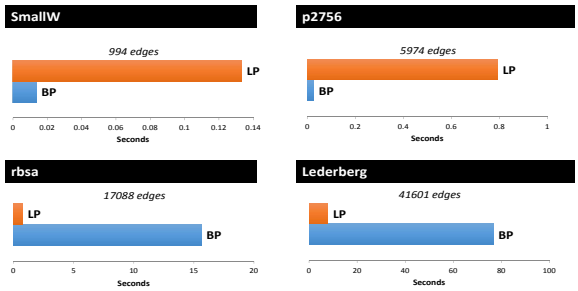

Figure 3: Convergence time of realistic dataset

subgraph of tight edges. The cost of a perfect matching in G_y equals the value of y.

2.4 Blossom algorithm [7]

The blossom algorithm is an algorithm for constructing maximum matchings on graphs. The matching is constructed by iteratively improving an initial empty matching along augmenting paths in the graph. Micali and Vazirani [5] showed an algorithm that constructs maximum matching in $O(|E||V|^{1/2})$ time.

3. DESIGN

We had two major challenges: to speed up the algorithm and to make BP converge when it has multiple optima.

3.1 Dataset

Two different datasets are used.

Random dataset is generated randomly with 1% of all possible edges.

Edge weight are set to the integers between 1 and $\frac{|S|*|U|}{100}$ if S is one subset of nodes U is the other subset.

Realistic dataset is from "The University of Florida Sparse Matrix Collection [1]". The distribution of edge weights are mostly biased.

We first tested our implementation with random dataset then moved on to the realistic dataset.

3.2 Initialization

For speedup, we initialized messages with respect to edge weights. The larger the weight, the more likely the edge to be selected. Therefore, by assigning proper initial values, it is possible to accelerate convergence. Figure 3 shows the relationship between convergence time and initialization coefficient. The convergence was fastest when the initial messages were set to the half of the edge weights.

3.3 Noise

The algorithm converges only when it doesn't have any fractional optima. However, sometimes the data has multiple optima, especially the realistic data. Biased weight distribution leads to a lot of fractional optima.

To enforce convergence by making the solution unique, we added a weight noise to the current weight and removed it after convergence. This method would not affect to the solution only if the sum of all the noises is less than 1.

To avoid each noise being too small, the noise was only added to **unstable edges**, the edges that the decision keeps changing after several iterations. Here, the noise was given after first 100 iterations and the edges were marked unstable after 20 iterations.

Let n the number of unstable edges. The noise is a random number between 0 and 1/(n+1).

Figure 2 shows the convergence time of random dataset when the noise was given after 100 iterations. BP was up to 12.54 times faster than GLPK.

Realistic dataset has much more fractional optima then the random dataset. Most of the random dataset reached convergence after giving random noise once, however, the realistic dataset barely did. We tried adding another noise and run after removing previous noise until convergence. The noise was added after every 100 iterations.

The result varies with the input data. Since GLPK is multi threaded, the performance of BP is still comparable to that of LP.

However, repeating the same step is inefficient until it gets the best noise by generating random noise sets. In figure 4, we enforced matching after every 100 iter-

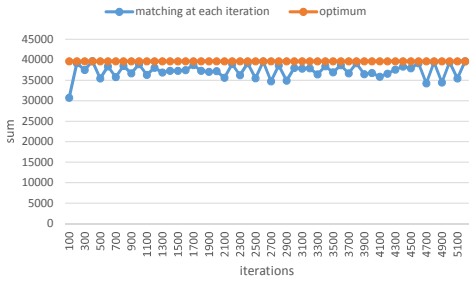

Figure 4: Weight sum of current matching stays near optimum

ations before adding noise and plotted current weight sum with respect to the number of iterations. An edge is randomly selected when a node is connected to multiple edges. The result shows that BP gets near optimum quickly and there is not much difference after once the noise is given.

3.4 Multi-threading

Using the fact that BP is a graph-parallel algorithm, a multithreaded algorithm can be devised. Two methods were explored: synchronous and asynchronous. The single-threaded version of the algorithm sequentially updates the belief messages for each node and then sequentially passes the messages through via the edges, which is considered to be one iteration of the algorithm. The general concept of the single-threaded version is to prepare the messages then pass them when all nodes have prepared their messages.

In the synchronous multithreaded version, the algorithm tries to mimic this behavior but to reduce the computation time by dividing up the graph and doing iterations concurrently, while also synchronizing the start of message preparation and delivery. First, the graph is divided up into groups with equal number of nodes and edges. Then, for each iteration, a group is assigned to a thread to prepare their messages and pass them. The threads must synchronize between each iteration and also between the message preparation stage and the passing stage to ensure that all nodes have prepared their messages before passing them.

In the asynchronous version, however, the idea of an iteration is slightly loosened. The synchronization step in between the preparation and passing stage is discarded and multiple iterations are done at once. Instead of having to wait for each iteration to start, the threads continually prepares and passes messages until they are stopped by the main process or until the algorithm converges.

The performance of each algorithm was tested by testing multiple randomly generated bipartite graphs of varying sizes and configurations. Five different graph node sizes were chosen for which five different graphs with different connection configurations were tested twenty times each. The performance was measured by time alone, but the number of iterations was also recorded for further analysis.

3.4.1 Synchronous

Three functions were added to the single-threaded version of the BP solver. Two functions were newly added - the makeThread and processThread function - and a preexisting function was modified - the oneIteration function.

The makeThread function makes multiple slave threads, each one running the processThread function, and returns when they are ready. It is this function that randomly divides up the graph into groups and assigns it to each slave thread. It also assigns the necessary shared variable addresses between the threads and the condition variables to wait on. This function must be called before the main BP solver loop to initialize the slave threads.

The oneIteration function is modified to wake the slave threads for one iteration of the BP algorithm. It waits for the slave threads to signal the end of their iteration and then returns. The processThread function is the designated method for each thread. One the thread is initialized, it updates the number of threads ready and waits on the wakeup condition specified by the makeThread function. If the desired number of threads are made - 12 in our implementation - then the last thread to go into the waiting state signals the main thread of the initialization before going into its dormant state. The slave threads are only woken up by the oneIteration function signaling for the thread wakeup and iterate procedure. The slave threads first prepare the node messages for their assigned node groups and then wait for all the threads to finish. Once all the threads have finished preparing their messages, they start passing the messages to their respective edges of their assigned groups. Once the threads all finish, the last thread to finish signals the main thread of the job completion and all the threads wait to be woken up again.

As figure 5 shows, the average time for the synchronous algorithm is higher than that of the single-threaded version. The most likely explanation for this is the additional overhead time for the multithreaded version's framework. The main thread has to wake each slave thread individually and each slave thread has to wait for other threads to finish their work for it to move on. These small inconveniences add may add up to a huge bottleneck.

To further investigate the source of the long runtime, we plotted the average time it takes per iteration, as follows:

Figure 6 suggests that the wait time between slave

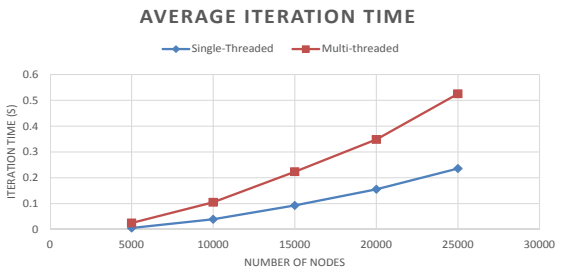

Figure 5: Weight sum of current matching stays near optimum

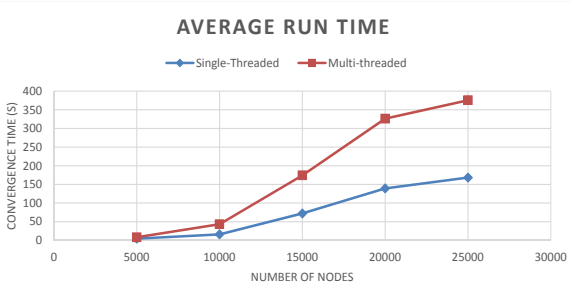

Figure 6: Weight sum of current matching stays near optimum

threads to finish is causing more of the time delay than the wake-up procedure, as the wake-up should only make a constant time difference for each iteration.

3.4.2 asynchronous

In the asynchronous version, the three functions in from the synchronous version are modified and another new function is added - the Iterate function.

The makeThread function is kept the same except for the shared variables it must additionally assign to each slave thread. The oneIteration function is reverted back to its single-threaded algorithm. This is only used to check whether the algorithm has converged or not in an incremental fashion, as the main function that handles iterations does multiple iterations at once, making it inconvenient for convergence checking.

The Iteration function works similar to the oneIteration function from the synchronous version, but it takes in an extra integer as an input. This integer value determines how many iterations the algorithm will approximately iterate before returning. This number is set to 20 in our implementation.

The processThread function works quite similarly to that of the synchronous version with some changes. One difference is that this version does not wait in between the message preparation and passing phases of the algorithm, they simply keep iterating through them without waiting until they reach a certain number of iterations. Another difference is that the slave threads do multiple iterations, instead of one, to signal the master thread to notify that they are done. The makeThread function randomly designates one of the slave threads as the counter that counts the amount of iterations it has done. When the counter thread reaches the desired number of iterations, it signals all the other slave threads to stop and wait after finishing their current iteration, regardless of how many iterations they have done. Then, when the slave threads all finish, they wait to be woken up again and signal the master thread that the job is done.

When testing a graph with 5000 nodes and 62500 edges, the algorithm converged after 57088 iterations and spent 523.820 seconds, which is 60 times the average iteration and a hundredfold the time it took for the single-threaded version. We speculate that the main problem lies in the data race of the nodes and the edges, as an edge might be stuck with a stale value for a long time until it is updated in the asynchronous version. This also explains why the algorithm fails to converge at times.

4. CONCLUSIONS

We idntified the limitation in existing optimization algorithms that they're structurally not parallelizable. We used BP, a message passing machine learnng algorithm has parallel structure, to solve the MWM problem in distributed environment.

We had two major challenges: speedup and multiple optima. For speedup, we initialized messages to half of their edge weights. And for multiple optima, we are using noise temporally to make the routine converge.

The algorithm worked pretty well on the ramdom dataset, but still have some problems with realistic dataset. We leave further approaches to future work.

5. FUTURE WORK

We saw that just giving noise repeatedly is not effective. Here we suggest two further approaches.

- Gradual noise

  Start with a large noise and decrease it gradually until the sum of all the noises is less than 1. For now we're adding noise to unstable edges. The more edges being marked as unstable, the smaller each noise gets, which doesn't make much difference when it is applied. To deal with this problem, by adding a large noise, we move the routine to some state, then by removing previous noise and assigning smaller noise, we get the right solution.

- BP-Hungarian algorithm

  As figure 4 shows BP is good at brining the state near to the optima. By applying Hungarian algorithm from the state BP made, we can get the advantages from both. Hungarian algorithm doesn't have the convergence issue with multiple optima.

6. REFERENCES

[1] The university of florida sparse matrix collection.

[2] M. C. Andrew E. Gelfand, Jinwoo Shin. Belief propagation for linear programming. *ISIT*, pages 2249–2253, 2013.

[3] Y. W. D. Gamarnik, D. Shah. "belief propagation for min-cost network flow: convergence & correctness". *SODA*, pages 279–292, 2010.

[4] M. S. M. bayati, D. shah. Max-product for maximum weight matching: Convergence, correctness, and lp duality. *IEEE Transactions on Information Theory*, 54(3):1241–1251, 2008.

[5] V. V. Micali Silvio. An $O(V^{1/2}E)$ algorithm for finding maximum matching in general graphs. *21st Annual Symposium on Foundations of Computer Science*, pages 17–27, 1980.

[6] A. W. Sujay Sanghavi, Dmitry Malioutov. Belief propagation and lp relaxation for weighted matching in general graphs. *IEEE Transactions on Information Theory*, 57(4), 2011.

[7] Wikipedia. Blossom algorithm.

[8] Wikipedia. Hungarian algorithm.