Conflict Graphs for Parallel Stochastic Gradient Descent

Darshan Thaker*, Guneet Singh Dhillon*

***Abstract*— We present various methods for inducing a conflict graph in order to effectively parallelize Pegasos. Pegasos is a stochastic sub-gradient descent algorithm for solving the Support Vector Machine (SVM) optimization problem [3]. In particular, we introduce a binary tree-based conflict graph that matches convergence of a well-known parallel implementation of stochastic gradient descent, know as `HOGWILD!`, while speeding up training time. We measure these results on a real-world dataset in a binary classification setting. This allows us to run various experiments to compare iterations between Parallel Pegasos and traditional Pegasos in order to effectively measure classification accuracy.**

I. Introduction

A Support Vector Machine (SVM) is a popular classification algorithm. It learns a hyperplane that linearly separates a group of points and is also as wide as possible.

In a binary classification task, a halfspace created by the hyperplane corresponds to points in one label, whereas the other halfspace corresponds to points in the other label. A hyperplane is represented by the normal vector to the hyperplane, denoted as w. And the classification for a data point x is determined by its inner product with w. If the dot product is positive, x is labeled 1, and -1 otherwise.

Additionally, there might be several hyperplanes that would be able to divide the training data linearly with a hyperplane. However, in order for the SVM to be generalizable, the algorithm picks the hyperplane that is as wide as possible. Wideness of a hyperplane is determined by the minimum distance of the hyperplane to all the data points. In other words, the boundary separating the two classes of data is picked in a way that the minimum distance to a data point from both the classes is the same. So while predicting a classification for a data point, the absolute value of its inner product is desired to be above a certain threshold. This leads to the *hinge loss* function for a given point x and its label y (while ignoring the bias term):

$$\ell(w;(x,y)) = \max\{0, 1 - y\langle w, x\rangle\}$$

Here, $\langle u, v\rangle$ denotes an inner product between u and v. The width is proportional to $1/\|w\|_2$.

* These authors contributed equally to this report.

In order to maximize the width, the norm of w needs to be minimized. This can be incorporated as a regularization term, in addition to the hinge loss of each data point.

Formally, given a training set of $X = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, the optimization problem for an SVM is:

$$\min_w \frac{\lambda}{2}\|w\|_2^2 + \frac{1}{m}\sum_{(x,y)\in X} \ell(w;(x,y))$$

In the above problem, λ is the regularization parameter and $\ell(w;(x,y))$ is the hinge-loss function. The Pegasos algorithm solves the primal objective function above using a vanilla stochastic sub-gradient descent method.

II. Pegasos

In stochastic sub-gradient descent, at each iteration, the objective function considered is that for a particular data point picked for that iteration.

Initializing $w_0 = 0$, at iteration t, a random data point, (x_{i_t}, y_{i_t}), is picked, where index i_t is in the set $\{1, ..., m\}$. The objective function to be considered for this data point is:

$$\frac{\lambda}{2}\|w_t\|_2^2 + \frac{1}{m}\ell(w_t;(x_{i_t}, y_{i_t}))$$

Calculating the sub-gradient of the above objective function, we obtain:

$$\nabla_t = \lambda w_t - \mathbb{1}[y_{i_t}\langle w_t, x_{i_t}\rangle < 1] y_{i_t} x_{i_t}$$

where $\mathbb{1}[y_{i_t}\langle w_t, x_{i_t}\rangle < 1]$ takes the value 1 if the condition is met, and 0 otherwise.

The update for w is:

$$w_{t+1} = w_t - \eta_t \nabla_t$$

At the end of T iterations, w_{T+1} is the required normal to the hyperplane that defines the SVM. The pseudo-code for the algorithm is given in Algorithm 1.

Algorithm 1 Pegasos

```
Given: $X = \{(x_1, y_1), \dots, (x_m, y_m)\}$, $\lambda$, $T$
  Initialize: Set $w_0 = 0$
  for $t = 1, 2, ..., T$ do
      Choose $i_t \in \{1, ..., m\}$ uniformly at random
      Set $\eta_t = \frac{1}{\lambda t}$
      if $y_{i_t}\langle w_t, x_{i_t}\rangle < 1$ then
          Set $w_{t+1} \leftarrow (1 - \eta_t\lambda)w_t + \eta_t y_{i_t} x_{i_t}$
      else
          Set $w_{t+1} \leftarrow (1 - \eta_t\lambda)w_t$
      end if
  end for
  return $w_{T+1}$
```

A. *Mini-Batch*

Instead of considering just one data point at every iteration, k data points can be considered in order to give us a better estimate to the actual gradient of the objective function.

At iteration t, a set of k indices, A_t is chosen at random. The objective function to be considered for the data point with these indices becomes:

$$\frac{\lambda}{2}\|w_t\|_2^2 + \frac{1}{m}\sum_{i \in A_t} \ell(w_t; (x_i, y_i))$$

Calculating the sub-gradient of the above objective function, we obtain:

$$\nabla_t = \lambda w_t - \sum_{i \in A_t} \mathbb{1}[y_{i_t}\langle w_t, x_i\rangle < 1] y_i x_i$$

The update for w is:

$$w_{t+1} = w_t - \eta_t \nabla_t$$

At the end of T iterations, w_{T+1} is the required normal to the hyperplane that defines the SVM. The pseudo-code for the algorithm is given in Algorithm 2.

Algorithm 2 Mini-Batch Pegasos

```
Given: $X = \{(x_1, y_1), \dots, (x_m, y_m)\}$, $\lambda$, $T$, $k$
  Initialize: Set $w_0 = 0$
  for $t = 1, 2, ..., T$ do
      Choose $A_t \subset \{1, ..., m\}$ uniformly at random,
  where $|A_t| = k$
      Set $A_t^+ = \{i \in A_t : y_i\langle w_t, , x_i\rangle < 1\}$
      Set $\eta_t = \frac{1}{\lambda t}$
      Set $w_{t+1} \leftarrow (1 - \eta_t\lambda)w_t + \frac{\eta_t}{k}\sum_{i \in A_t^+} y_i x_i$
  end for
  return $w_{T+1}$
```

III. Parallelizing Stochastic Gradient Descent

Various approaches have been proposed to parallelize stochastic gradient descent (SGD) methods.This problem is inherently serial, since the parameter vector, or w, of one iteration depends directly upon on the parameter vector of the previous iteration.

Looking at how Mini-Batch works, it is easy to parallelize the algorithm by creating k threads in each iteration, where each thread gets a data point and it finds the update for the parameter vector. The updates for each iteration can be summed up once each thread is done to get the update for the parameter vector for that iteration. This though, is not a very big speed-up.

One notable approach for parallelizing SGD is the `HOGWILD!` approach [2], or the lock-free approach. In this algorithm, a thread operates on a shared-memory parameter vector *without* using any locks. This means that reads and writes can happen concurrently, and one thread may be modifying the parameter vector, while another thread reads an inconsistent vector for updates. However, the success of this method is reliant upon a measure of sparsity and randomness, ensuring that these inconsistent reads and writes do not affect the convergence drastically. In fact, when the input data is extremely sparse, this method performs very well and converges fast.

Another approach for parallelizing SGD, which is a more recent approach, is the `CYCLADES` algorithm [1]. In this algorithm, we focus on picking indices of the data points to use for updating the gradients. The key observation here is that inconsistent reads/writes occur when two data points have overlapping supports, or equivalently, these two data points have a 'conflict'. In this scenario, sparsity is helpful, since it is easier to find two data points that do not have overlapping supports. Suppose we wished to pick k indices of the data points to use for updating the gradients. The contribution of `CYCLADES` is to randomly sample k indices and model the conflicts between these k indices as a conflict graph. Thus, we connect two indices if there is a conflict between them. Then, a connected component of the conflict graph represents a set of indices that can safely be updated in parallel on a core. This gives us an estimate for the number of cores we need to perform a SGD mini-batch update with batch size k, and gives us a set of indices that can be allocated to each core.

The contribution of this report borrows ideas from the above algorithms, but ultimately uses a different approach to select non-conflicting indices to successfully parallelize stochastic sub-gradient descent in the context of SVMs. In particular, we note that `HOGWILD!` has no constraints on the indices that are picked, whereas `CYCLADES` imposes many constraints, so in theory, we could have many more connected components in a set of indices than cores available. Thus, the motivation behind our approach is that given a fixed number of cores, we would like a way to return a set of indices that can be updated on different threads.

A. Induced Conflict Graph Generation

We consider four techniques for generating sets of indices that may or may not have a conflict. Suppose we wish to pick a set of unique indices A where $|A| = k$. The conflict graph $G = (V, E)$ conceptually will have $|V| = k$ nodes and some number of edges between nodes, where an edge between two nodes denotes a conflict between the two nodes (i.e. an overlap in the features). At a high level, each method below induces a conflict graph and enforces some invariants on the number of edges $|E|$ in this graph. It is important to note that the conflict graph is never explicitly generated, but only used as a way to reason about various sets of indices and their effects on convergence rates. Each method is an iterative algorithm that starts off with a random index and builds up an induced conflict graph one index at a time.

1) This approach will enforce the invariant that every node will have an edge with at least one node. A simple approach to enforce this invariant is the algorithm that adds a new index to the set of indices if it conflicts with at least one of the previously generated indices. However, to reduce the lookup time of conflicts with all previous nodes, which requires $O(|V|)$ time, we only check conflicts with $O(\log(|V|))$ nodes.

   The goal of this is twofold. First, this approach introduces more stochasticity into the set of indices since we are only checking a small subset of previous indices for conflicts and discarding if there are no conflicts with these indices. Secondly, this approach improves lookup time for estimating the conflicts that a particular index has with previously generated indices. Note that it is possible for this method to discard more indices than necessary, if they do not happen to conflict with their ancestors. However, for large values of k, this method will, in expectation, find a set of 'valid' indices faster than an approach that scans all previously generated nodes.

   To implement this, it is beneficial to visualize a set of indices as a binary tree. We wish to enforce that every index conflicts with at least one of its ancestors, since the number of ancestors is logarithmic with respect to the total number of vertices. The iterative algorithm conceptually creates a binary tree in a level-order traversal and checks conflicts with all of the ancestors of a given node.
2) This approach enforces the invariant that every node must conflict with every other node. In particular, we build a complete conflict graph.
3) This approach uses a similar binary tree-based approach, however enforcing invariants on the number of non-edges in the graph, or the number of nodes that do not conflict. In this algorithm, we ensure that an index does not conflict with at least one of the ancestors in its binary tree representation.
4) This approach enforces the invariant that every node must not conflict with any other node. In particular, we build an empty conflict graph with no edges.

Observe that Methods 1 and 3 above attempt to improve sampling time without negatively affecting the convergence rate of Parallel Pegasos.

IV. Experiments

A. Dataset

We ran our evaluations and collected performance metrics on the CCAT data set. It is used for a classification task taken from the Reuters RCV1 collection.

Data Points	=	804414
Training Size	=	723973
Testing Size	=	80441
Features	=	47236
Sparsity	=	0.16%

B. Plots

We note three statistics on a single run of a method. These statistics are:

- The loss function value on the testing set for each iteration ($\lambda = 0$).
- The testing set percentage error rate for each iteration.
- The norm of the parameter vector for each iteration.

The different Parallel Pegasos methods vary based on the conflict graph generation. In particular, there are 4 Parallel Pegasos methods, corresponding directly to the 4 conflict graphs mentioned in Section III.

These three statistics were run on two values of k ($k = 2$ and $k = 10$), which represents the mini-batch

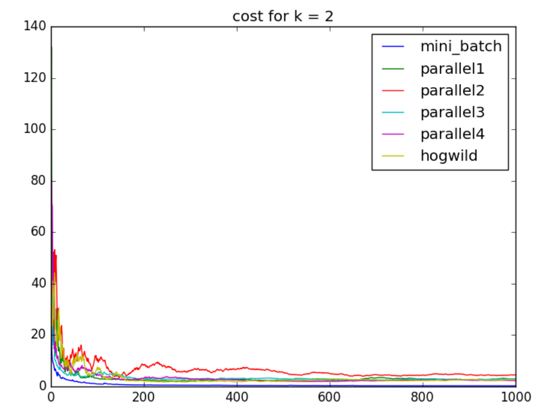

Fig. 1. Loss Function for $k = 2$

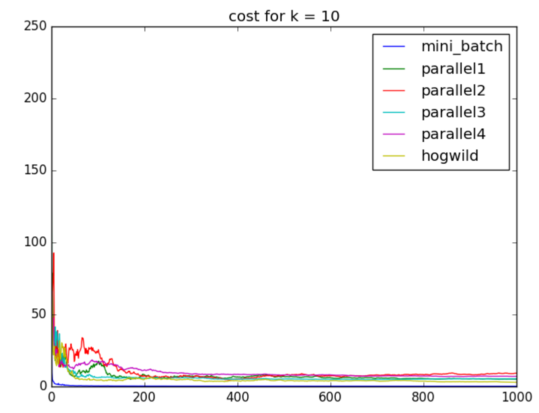

Fig. 4. Loss Function $k = 10$

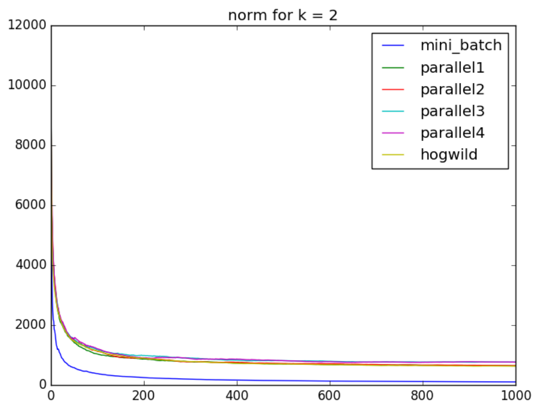

Fig. 2. Norm of Parameter Vector for $k = 2$

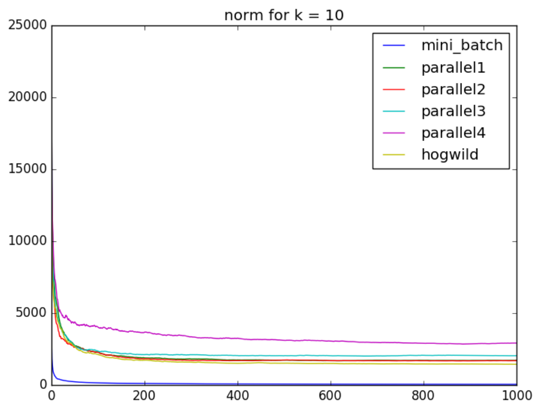

Fig. 5. Norm of Parameter Vector for $k = 10$

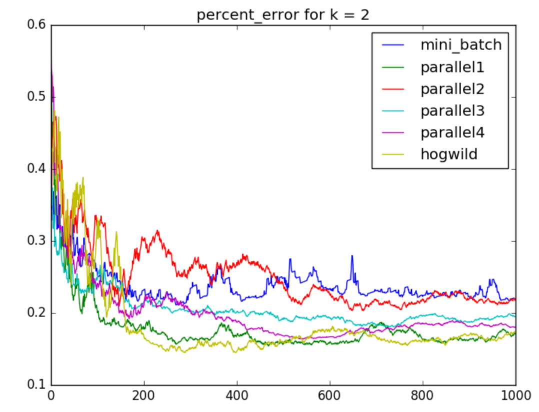

Fig. 3. Percent Error for $k = 2$

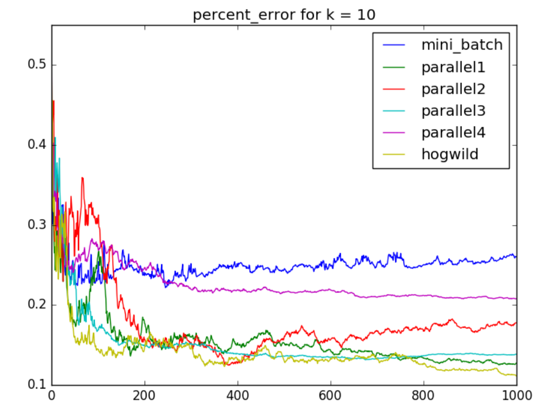

Fig. 6. Percent Error for $k = 10$

size or the number of indices to sample for a conflict graph. These runs were for 1000 iterations, and a λ value of 10^{-4}, borrowed from [3].

However, these figures do not capture the timing aspect of the various methods, which is described in Table I. Note that these timings include calculating the statistics presented in this report, per iteration.

TABLE I
ALGORITHM TIMINGS

k	2	10
Mini-Batch	641.188916922	541.698472977
Parallel1	437.151793003	439.413945913
Parallel2	455.587590933	441.605366945
Parallel3	486.384338856	453.869709969
Parallel4	462.699862957	6521.923002
HOGWILD!	565.418184042	601.219654083

Note that the Parallel Pegasos algorithms that use a binary-tree based algorithm are much faster than the mini-batch and HOGWILD approaches, while not sacrificing significant decreases in the percentage error rate, especially for the larger values of k. In fact, conflict graph 1 performs extremely well and almost matches the performance of the best method. When running conflict graph 4 for large values of k, Parallel Pegasos takes an extremely long amount of time, which is expected since generating nodes without any conflicts will take numerous iterations. This is another reason why the binary-tree based conflict graph performs well, since it does not take many iterations to keep a high convergence rate.

V. Conclusion and Future Work

Various methods were presented on how to get indices to parallelize Pegasos. Theoretically, inducing conflicts to parallelize should not work. However, we observed that inducing certain kinds of conflicts improved the algorithm. Some important questions to be asked here are why are such algorithms out-performing algorithms with no conflicts? Also, can there be any theoretical guarantees on such algorithms? We are not certain as to why these results so closely match performance of HOGWILD!.

In general, SVMs go hand-in-hand with Kernel methods, that help classify data that is not linearly separable in the dimension space that they are provided in. The introduction of a Kernel method only changes the objective function by substituting the inner product (Kernel trick). However, parallelizing the objective function then becomes very hard as the dimensions that the data points are raised to can be infinite (Gaussian Kernels are examples of that).

References

[1] X. Pan, M. Lam, S. Tu, D. Papailiopoulos, C. Zhang, M. I. Jordan, K. Ramchandran, C. Re, and B. Recht. Cyclades: Conflict-free asynchronous machine learning. *arXiv preprint arXiv:1605.09721*, 2016.

[2] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[3] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter. Pegasos: Primal estimated sub-gradient solver for svm. *Mathematical programming*, 127(1):3–30, 2011.