Parallelizing AdaBoost on Multi Core Machines

Vibhu Jawa
vjawa1@jhu.edu
Praateek Mahajan
praateek@jhu.edu

***Abstract*— AdaBoost, short for Adaptive Boosting, is a type of boosting algorithm which combines several weak classifiers to create one strong classifier. AdaBoosts fundamental nature doesn't allow for parallelizing finding the weak classifiers, we present a way which helps achieve nearly $22.14x$ times the speedup compared to a serial implementaiton. In this paper, we develop a parallel AdaBoost algorithm that exploits the multiple cores in a CPU via light weight threads. We propose different algorithms for different types of datasets and machines.**

I. INTRODUCTION AND BACKGROUND

Boosting [1] refers to a general and provably effective method of producing a very accurate prediction rule by combining rough and moderately inaccurate rules. Throughout the paper we say that our algorithm takes as input a training set $(x_1,y_1)\ldots(x_n,y_n)$ where each x_i belongs to $\mathbb{R}^m$, and each label y_i is in a label set $\{-1,+1\}$. The algorithm walks $1,2\ldots T$ steps to find T weak classifiers. Each successive weak classifier $t+1$, learns from the mistakes of classifier t. AdaBoost enforces this sort of learning, by maintaining a set of weights over the training set. The weights of the training set on round t are referred as D_t, where D_1 is set as $\{1/n\}^n$.

Once a weak classifier is found for round t, the weights D_t are updated, and then the classifier $t+1$ uses the updated D_{t+1} weights to make a classification.

More formally this is given as :

1) Initialize $D_1 = 1/n$ where D_t is described above.
2) For each of successive $t = 2,3\ldots T$
   a) Find weak classifier $h_t(x_i)$ which minimizes the following

$$num\_errors = \sum_{i=1}^{N} D_t(i)\mathbb{I}.(h_t(x_i) \neq y_i)$$

   (*the number of misclassifications*)

   b) Set $\varepsilon_t = num\_errors/N$
   c) Set $\alpha_t = \frac{1}{2}\ln\frac{1-\varepsilon_t}{\varepsilon_t}$
   d) Update weight of every example as per it's classification i.e

   i)

$$D_{t+1}(i) = \frac{D_t(i)}{Z}\exp\left(-\alpha_t h_t(x_i) y_i\right)$$

   ii) where Z is the normalization constant

$$Z = \sum_{i=1}^{N} D_t(i)\exp\left(-\alpha_t h_t(x_i) y_i\right)$$

3) Then we combine all weak classifiers using

$$H(x_i) = \sum_{t=1}^{T} sign(\alpha_t h_t(x_i))$$

Practically, AdaBoost has many advantages. It performs well without hyperparameter tuning (except for the number of rounds/weak classifiers) and requires no prior knowledge about the weak learner. Thus it can be flexibly combined with any method for finding weak hypotheses.

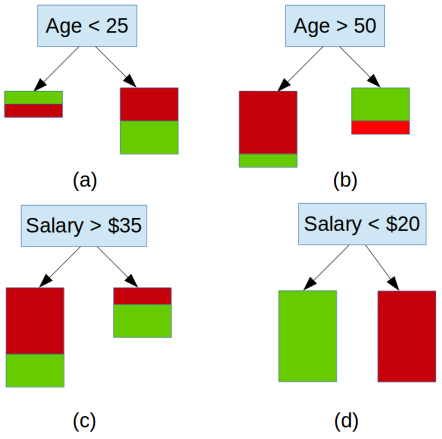

Fig. 1. Here green and red denote $+1,-1$ classes respectively. Different stumps on the different feature and different values provides different results. We show that all $(a)(b)$ and (c) provide sub-optimal splits, whereas (d) yields the best split.

II. OUR SETUP

We create our experiment setup only for the task of binary classification and continuous feature vectors. In our experiment, we use decision stumps (step $2.a$) as weak classifiers. A decision stump makes a prediction based on the value of just a single input feature. For continuous features, usually, decision stumps select a threshold feature value, and then split the data into two leaves for values below and above the threshold.

We show in Fig. 1 that to select the best split, we need to iterate over all the features, m, and also all the n values over which we need to perform the split. For every split, we compute the *quality* of the split by iterating over all examples, and seeing how many values less than the *split value* are correctly/incorrectly labeled. The best split that yields the least number of incorrect results, gives us our best split and helps us achieve step $2.a$

```
#pragma omp parallel for
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            dst[j][i] = src[i][j];
        }
    }
```

Listing 1. Transpose using OpenMP

A. *Taking a transpose : $X \rightarrow X^T$*

Most of the datasets available are in $n*m$ dimensional. But for our work, we need to split values across features, and find the best split. Since in C++, data is stored in row major form, we take a transpose of our matrix X. This makes sense as we don't iterate over examples but over features of all examples thus the feature values need to be contiguous than the examples itself.

We experiment with different transposing solutions available online, but learned that in C++11, vector of a vector (2D) are not contiguous in nature and thus tiling does not make sense. When we compiles our final code using $-O3$ flag, it resulted in a much faster performance than the parallelized code given in Listing. 1.

B. *Selecting the split values*

User	**1**	**2**	**3**
Age	10	20	40

TABLE I
WE WOULD SPLIT AT 15, 25, 35. THIS HELPS PREVENT OVERFITTING TO THE TRAINING DATA, THUS ENFORCING MAX-MARGIN PRINCIPLE.

Every time while selecting h_t we iterate over all features and then try different split values. We select split values by **enforcing a max-margin principle** as described in Table II-B. To get the split values, we initially sort our data for the respective feature once, and then compute midpoints between every two data points. We also check if we don't have the same repetitive mid-value again. This also helps **save only unique values** for every feature, thus reducing the time to find best the best h_t.

C. *Adaboost in for loops*

```
for t in range(1,T):
    for ft in features:
        for split_val in split_values(ft):
            for x_i,y_i in X,Y:
                if x_i[ft] < split_val & y_i != -1:
                    error += weight_t,i
                if x_i[ft] > split_val & y_i != 1:
                    error += weight_t,i
            save error for the split_value on ft
        save least error over all split_value on ft
    h_t = least error over all features
    update_weight(weight_i) ∀x_i
    ...
```

Listing 2. Python example

We see that loop on *Line 1*, is not iteration independent, i.e the second iteration depends on updates of first iteration. This is shown in step 2.d.i, in Section I.

However to find the best feature split value, for h_t, we have three for loops *(Line 2, 3, 4)*, which are independent of the iteration. In the coming sections we describe the results and experiments we tried for each of these lines.

III. DIFFERENT PARALLEL APPROACHES

We use the following terms to explain the approaches :

- n : number of examples
- m : number of features
- s_i : split values of feature i
- t : number of threads/cores available

A. *Finding Error For A Given Split Value*

Here we parallelize *Line* 4 in Listing 2. We parallelize using an `openmp for` directive along with `reduction` to `sum` up the error. Here each thread gets a set of x_i, y_i, which sums over the weights. We observe that this yields in the worst performance. This can be attributed to excessive startup cost. Since the threads are initiated and synchronized $\sum_{i=1}^{m} s_i \approx m*n$ times.

We observed negligible speedup after parallelizing this.

B. *Finding Best Split Value for a given feature*

Here we parallelize *Line* 3 in Listing 2. We parallelize using an `openmp for` directive along with `reduction` to find the `minimum` error across all split values. We observe the following

1) Good to use, if $t > m$.
   Every thread will be utilized, in finding the best split value for the given feature. Compared to Section III-C.
2) Good to use, if our $s_i > t \forall i$
   If split values for a given feature are less than the number of threads, then maximum parallel efficiency will not by utilised. For example on a feature like Gender, with only two possible values "Male" and "Female", a 32 core machine would not be able to spawn enough resources to make the most out of it.
3) This results in some extra overhead since, threads are spawned m times.

C. *Finding Best Feature for given t*

Here we parallelize *Line* 2 in Listing 2. We parallelize using an `openmp for` directive along with `reduction` to find the `minimum` error across all features. We observe the following

1) Good to use, if $t < m$.
   Otherwise, several threads would be unitilized. For example on a 32 core machine, and a dataset with only 10 features, such an approach would not redeem the most of the possible parallel efficiency possible.
2) Good to use, if our $s_i \approx n$.
   This means that values for a feature are unique, therefore the split values are distinct. If that was not the case, then it would result in skew, as for some i, j, $size(s_i) << size(s_j)$, which means that the thread

TABLE II

THREAD 1 WOULD JUST PERFORM ONE ITERATIONS TO PRINT 24 AND WILL WAIT FOR THREAD 2 TO FINISH, WHICH WILL HAVE TO ITERATE FOR 5 TIMESTEPS.

Age (thread 1)	**18**	**30**	**30**	30	30
Weight (thread 2)	184	129	158	193	140

responsible for i would get over much faster than the others. In such a case, it's best to use dynamic scheduling.

.

IV. OUR MOST OPTIMAL MODEL

While figuring out which procedure to parallelize, we realized that the sequential part of the code is still very high, (parallel efficiency of 0.93 , for a speed up of 10.46x with 32 cores). This sequential part prevented the sub-linear speedup that we were expecting. To further optimize the code, we extend our approach in III-C, to parallelize two more functions.

1) **Finding Feature Split Values**
   As explained in Section II-B, we find the midpoints of all out feature values. Since this loop is independent this can easily be exploited using `openmp for`. However if data has a lot of same values for a feature, then this can result in severe skew as shown in Table II. We avoid such a task by adding `dynamic scheduling`.
2) **Updating Weights**
   We also need to update weight based on its classification by the current weak classifier for each example We also parallelized this part of the code and is given in listing. This also help us make gains in the performance. 3.

```
weight_sum = 0
#pragma omp parallel for reduction (+:weights_sum)
for i in range(0,len(X[i]):
    if prediction for X[i] is incorrect:
        w[i] = w[i] * exp(a_t * l[i])
    else:
        w[i] = w[i] * exp(-1 * a_t * l[i]);
    weight_sum = 0 = weight_sum + w[i]
#pragma omp parallel for
for i in range(0,len(X[i]):
    w[i] = w[i] / weight_sum
```

Listing 3. Weight Update Code

V. CONCLUSIONS

We achieved a 22.14x speed up in our implementation. The normalized time taken by our best implementation is presented in Figure IV. With our final implementation we achieved parallel efficiency of 0.983 (with 32 cores), thus the code is highly parallized.

We also show cased the speed up across the implementations in figure IV

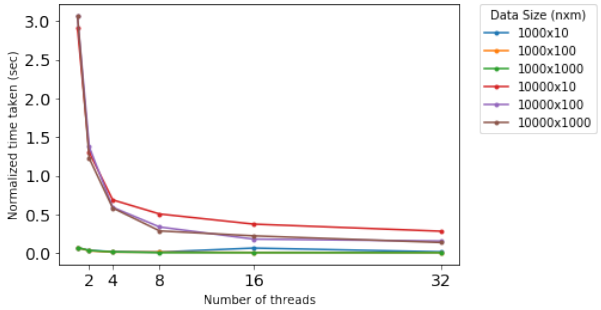

Fig. 2. Time Taken for each data size matrix normalized by number of operations

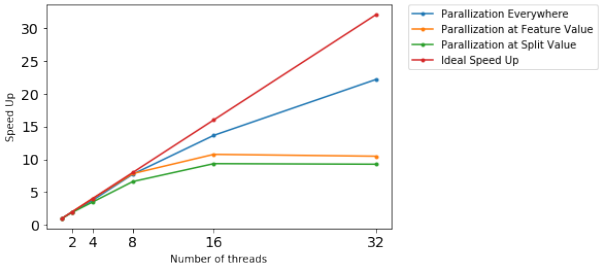

Fig. 3. Speed up of different implementations

WORK DISTRIBUTION AND GITHUB LINK

GitHub Link: https://github.com/praateekmahajan/parallel-adaboost

Programming

1. Vibhu:
   - Initial serial implementation of the code in C++
   - Parallel Transpose experimentation
   - Weight update and finding feature split val parallization.
2. Praateek:
   - Serial implementation in Python, (Helped in debugging the C++ code and sanity check its accuracy)
   - Parallized and ran experiments explained in section 3.

Writeup Sections:

1. Vibhu:
   - Our Most optimal Method
   - Conclusion
2. Praateek:
   - Introduction
   - Set Up
   - Different Parallel Approaches

ACKNOWLEDGMENT

We would like to thank Professor Randall Burns for this opportunity, and AWS educate for providing us credits for the project.

REFERENCES

[1] Freund, Y., Schapire, R. and Abe, N., 1999. A short introduction to boosting. Journal-Japanese Society For Artificial Intelligence, 14(771-780), p.1612.

[2] Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." IEEE computational science and engineering 5, no. 1 (1998): 46-55.