A Scalable and Parallel Implementation of the MultiBoost Library's AdaBoost.MH (Adaptive Boosting) Algorithm

Darshan Thaker (dbthaker@cs.utexas.edu)

Prabhat Nagarajan (prabhatn@cs.utexas.edu)

Abstract

This report explores a parallel implementation of the C++ MultiBoost open source library's AdaBoost.MH (Adaptive Boosting) algorithm (D. Benbouzid,2012), and the changes in performance as measured on the Texas Advanced Computing Center's Stampede supercomputer. This parallel implementation is based on the AdaBoost.PL Algorithm (Palit & Reddy,2012). The baseline is the AdaBoost.MH algorithm, and this algorithm's performance is contrasted against our AdaBoost.PL implementation. This report also explores the feasibility and effectiveness of other possibilities for optimization such as vectorization, cache optimizations, and GPU optimizations. The best performance increase we get is 2.7x speedup from a parallel implementation.

Keywords: Adaptive boosting; Performance; Parallelization; pthreads

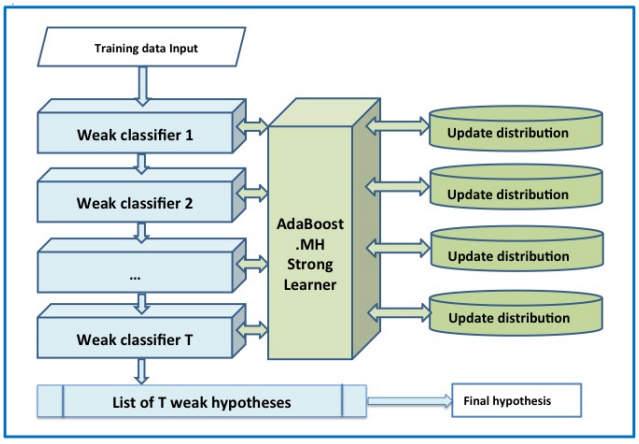
Background

The Adaptive Boosting algorithm (AdaBoost) is used for training a supervised machine learning classifier. A traditional machine learning classifier extracts hidden relationships between datapoints in a given dataset. Essentially, a classifier aims to place these datapoints into specific categories. An **example** in a dataset consists of a value for each of the x total features and an associated **label** y from a pool of labels (or **classes**). The classifier is trained on some subset of the entire dataset (known as the **training set**) and its performance is evaluated by testing it on the remaining part of the dataset (known as the **testing set**). A classifier will learn to create a **hypothesis function** from the training set, which is a function mapping given values for x features to a label y in the set of labels.

AdaBoost.MH Algorithm Intuition

The goal of boosting algorithms such as AdaBoost.MH is to maintain a list of **weak classifiers** and average these weak classifiers to get a final **strong classifier**. A *weak classifier* is defined as any supervised classifier that will generate a hypothesis function from the training data.
The intuition behind AdaBoost.MH is to maintain a **distribution of weights** such that every example in the training set will have a weight associated with it. This distribution will change after every iteration of AdaBoost.MH. Initially, this is an uniform distribution of weights across every training set example. Every iteration of AdaBoost.MH will run one weak classifier and generate a weak hypothesis function. The **training error rate** of this classifier is defined as the sum of the weights for the training examples on which the weak hypothesis function incorrectly classifies that input. In other words, this error rate is a measure of the number of examples in the training set that were misclassified. The distribution of weights over the training set is then modified according to this error rate so that the weights for training examples that were misclassified increase. Similarly, the weights for training examples that were classified correctly decreases. This essentially signals to the weak classifier of the next iteration of AdaBoost.MH to focus more on "learning" from the misclassified training examples and focus less on correctly classified examples. After T iterations of AdaBoost.MH, the strong hypothesis function is constructed as a weighted sum of the weak hypotheses functions. The weight of each hypothesis function is its error rate.
Figure 1 below summarizes and shows the high-level description of the AdaBoost.MH algorithm. Note that the way the diagram is represented shows the inherent serial nature of the algorithm.

Figure 1: Inherently Sequential AdaBoost.MH algorithm



As an aside, AdaBoost.MH is a multi-class version of the regular AdaBoost algorithm, which only works on binary classification problems.

AdaBoost.MH Algorithm Mathematics

We will introduce the AdaBoost.MH algorithm from a formal mathematical standpoint now for the mathematically inclined. However, for the purposes of clarity, the previous section is enough to gain intuition of the AdaBoost.MH algorithm.

Definitions:

$X_{i1}...X_{in}$ = n feature vectors for training set
y_i = True label for example i in training set
j = Number of training examples
l = set of labels for training set
$h_t(x) : X \to Y$ = Weak hypothesis function on iteration t
$H(x)$ = Final strong hypothesis function
$D_t(i,l)$ = Distribution of weights on iteration t
α_t = Learning rate based off the training error rate
ε_t = Training error rate
Z_t = normalization factor so D_{t+1} will be a distribution
$Y[l] = +1$ if $l \in Y$ and -1 if $l \notin Y$

Note that the α_t is a learning rate specific to the AdaBoost.MH algorithm. It merely serves to update the distribution of the next iteration. The choice of this learning rate can be proven from a statistical method, which is done in the initial AdaBoost paper (Schapire & Singer,n.d.).

Algorithm 1 AdaBoost.MH algorithm

Given: $(X_{1,1}...X_{1,n}, l_1)...(X_{j,1}...X_{j,n}, l_m)$
Initialize $D_1(i,1) = 1/m$
for $t = 1...T$ **do**
Train weak learner using distribution D_t
Get weak hypothesis h_t from this weak learner

$$\varepsilon_t = \sum_{i:h_t(x_i) \neq y_i} D_t(i)$$

$$\alpha_t = ln(\frac{1-\varepsilon_t}{\varepsilon_t})$$

Update:

$$D_{t+1} = \frac{D_t(i,l)exp(-\alpha_t Y_i[l]h_t(x_i,l))}{Z_t}$$

end for
Compute the final hypothesis function:

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

AdaBoost.PL Algorithm Intuition

This section aims to introduce the parallel version of the AdaBoost.MH algorithm. This algorithm is titled AdaBoost.PL. The algorithm divides up the training set to m worker threads, which generate $m*t$ different weak hypotheses functions, put into a matrix P. The rows of this matrix P are sorted with respect to the α of the weak hypotheses. To obtain the final hypothesis function, an extra merging step is taken. This merging step averages classifications down a column of P and takes a majority vote by running classifications for every example in the training set on each weak hypothesis. From a pedagogical perspective to understand this more deeply, it is useful to think about a hypothesis function as a hashmap that maps the dataset inputs to specific labels. In reality, however, note that the hypothesis function is a mathematical function with a domain and range, rather a finite hashmap. Every weak hypothesis function in the matrix P returns a value $\in (-1,1)$ as to whether there exists a mapping from an input to a label. If a majority of weak hypothesis functions in a column of P map a single input to a label, the final hypothesis will map that input to that label. Thus, every input in your testing set must run through all weak hypotheses functions in the matrix P to generate a label for that input.

Figure 2 describes the partitioning of data to AdaBoost.PL in the training phase, and Figure 3 describes the merging of weak hypotheses functions in the testing phase.

Figure 2: AdaBoost.PL algorithm

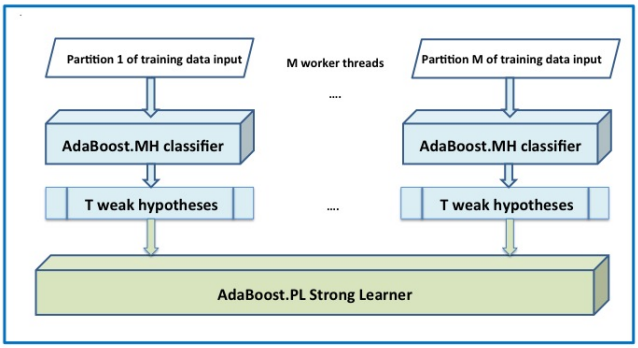
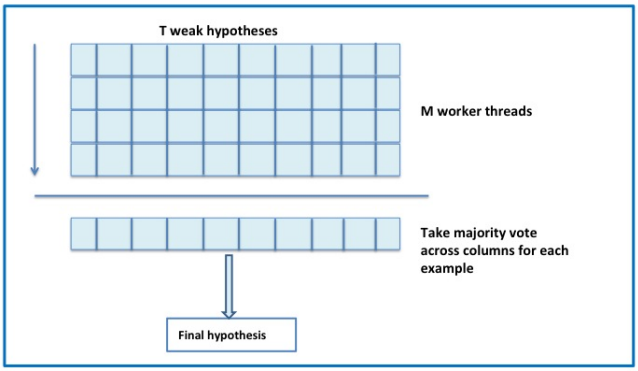


Figure 3: Merging step for matrix P in AdaBoost.PL



AdaBoost.PL Algorithm Mathematics

Definitions:

M = Number of worker threads
H^P = strong classifier outputted by the AdaBoost.MH function
$\alpha^{P(t)}$ = weight of weak classifier from the t^{th} iteration in AdaBoost.MH
H^{P*} = List of weak hypotheses functions in H^P sorted w.r.t. $\alpha^{P(t)}$
$h^{(t)=}$ hypothesis function generated by merging the t^{th} classifiers from each of the M H^{P*}'s.
α^t = the α corresponding to the hypothesis function $h^{(t)}$ as in the traditional AdaBoost.MH
H = final strong classifier (result of AdaBoost.PL)

Algorithm 2 AdaBoost.PL($D^1_{n^1},...,D^M_{n^M}$,T)

Input: The training sets of M workers ($D^1_{n^1},...,D^M_{n^M}$)
Number of boosting iterations (T)
Output: The Final Classifier (H)
Procedure:

1: **for** $p \leftarrow 1$ to M **do**
2: $H^P \leftarrow Adaboost.MH(D^P_{N^p},T)$
3: $H^{P^*} \leftarrow$ the weak classifiers in H^P sorted w..r.t. $\alpha^{P(t)}$
4: **end for**
5: **for** $t \leftarrow 1$ to T **do**
6: $h^{(t)} \leftarrow MERGE(h^{1*(t)},...,h^{M*(t)})$
7: $\alpha^t \leftarrow \frac{1}{M}\sum_{p=1}^{M}\alpha^{P*(t)}$
8: **end for**
9: return $H = \sum_{t=1}^{T}\alpha^t h^{(t)}$

The merged hypothesis function $h^{(t)}$ is called a *ternary classifier*, which can yield values of $+1$, -1, or 0. The function $h^{(t)}$ is created by taking a majority vote from the workers' weak classifiers. It is defined as follows:

$$h^{(r)} = \begin{cases} sign\left(\sum_{p=1}^{M} h^{p*(t)}(x)\right) & if \quad \sum_{p=1}^{M} h^{p*(t)}(x) \neq 0 \\ 0 & otherwise \end{cases}$$

Experimental Setup

The baseline optimizations and results are tested on a Stampede compute node in the Texas Advanced Computing Center (TACC). The following descriptions show the experimental setup:

1. Stampede Compute Node:
   - 16 Intel(R) Xeon(R) CPU E5-2680 - 2.70GHz processor
   - Intel Xeon Phi CoProcessor
   - Linux (CentOS 6.3 Distribution)
   - Run using gcc linked with the following libraries: multiboost, MultiBoostLib, Bzip2Lib, pthread, m, rt

Running adaptive boosting involves the choice of a weak learner (or base learner) for AdaBoost.MH and AdaBoost.PL. Our optimizations are concerned with optimizing the AdaBoost.MH algorithm itself, regardless of the choice of the weak learner. Therefore, we select a well known and relatively simple weak learner: SingleStumpLearner. This learner is a one-leaf decision tree. Selecting this weak learner corroborates the effectiveness of Adaboost.MH, because running a one-leaf decision tree on its own as a traditional classifier will give very little predictive power. This choice of weak learner provides a good metric for performance comparison between AdaBoost.PL and AdaBoost.MH because the weak classifier will perform poorly on its own.
We set the number of boosting iterations to be 200. We measure the performance of the baseline and parallel implementations for 3 datasets (see the following section, **Input Data**). For our baseline algorithm, AdaBoost.MH, we simply obtain timings for training and testing on the 3 datasets. For AdaBoost.PL, we obtain timings and plot graphs for the number of worker threads vs. speedup for training, testing, and total times.
Additionally, to evaluate the correctness of the parallel implementation vs. the serial implementation, we compare the two confusion matrices of the implementations. A confusion matrix describes a matrix where each column of the matrix represents the instances in a predicted class while each row represents the instances in an actual class. For instance, in a binary classification problem, the confusion matrix will show the number of false positives, false negatives, true positives, and true negatives. To compare two $y \times y$ confusion matrices, we first break up this confusion matrix into y 2×2 confusion matrices for each label. This means we find the true positives, true negatives, false positives, and false negatives with respect to each label. We then compare the following statistics for each of these matrices: Recall (true positive rate), Specificity (true negative rate), Precision (positive predictive value), Negative predictive value, Fall-out (false positive rate), False discovery rate, Miss rate (false negative rate), Accuracy, F1 score (harmonic mean of precision and recall), and Matthews correlation coefficient. Checking whether the difference of these coefficients is within a certain range (≈ 0.1) for the parallel and serial implementation will show correctness of the algorithm with a very high confidence level. A full proof of convergence for AdaBoost.PL can be found in the AdaBoost.PL paper.
One possible cause of experimental error in our timings is that the Adaboost.MH serializes throughout its *run* method. Consequently, it is extremely difficult to divorce serialization from the computation without modifying the entire library and introducing an I/O component to our measurements. Additionally, while the baseline calls the AdaBoost

run method once, our parallel implementation calls this method M times (where M is the number of worker threads). As a result, serialization creates a bottleneck since each worker thread serializes a weak hypothesis. As the number of features go up, this weak hypothesis is much larger, so the number of features becomes an I/O bottleneck. We have included the unserialization of the partitioned files and the serialization of the weak hypotheses in the timings. However, we have not included the partitioning of the files in the timings.

Input Data

UCI - Machine Learning Repository: Yeast Data Set:

- Contains data of protein localization sites in yeast bacteria
- General Goal: identify localization site of protein
- Number of Instances: 1484
- Number of Attributes/Features: 8
- Number of Labels: 10

UCI - Machine Learning Repository: Dexter Data Set:

- Data is a subset of the Reuter's text categorization benchmark
- General Goal: filter texts about corporate acquisitions
- Number of Instances: 2600
- Number of Attributes/Features: 20000
- Number of Labels: 2

UCI - Machine Learning Repository: Pendigits Data Set:

- Samples of handwritten digits from 44 writers
- General Goal: identify digit correctly
- Number of Instances: 10992
- Number of Attributes/Features: 16
- Number of Labels: 10

Baseline Performance

The following table shows the baseline performance of the original Multiboost AdaBoost.MH classifier. It is run on various input datasets and the training and testing time is recorded. The row titled "Total" is the total time (training + testing) spent by the AdaBoost.MH classifier.

	Yeast	Pendigits	Dexter
Training	138.93 ms	15418.76 ms	6615.89 ms
Testing	169.44 ms	1296.06 ms	48.82 ms
Total	1558.79 ms	16714.82 ms	6664.70 ms

As shown, the Yeast dataset has a relatively quick baseline training and testing time, due to its to its small number of instances and features. The Pendigits and Dexter datasets take significantly longer, given their increased number of examples and features. While it may seem that the Dexter dataset should take longer than than Pendigits, given that it has $20,000$ features, it should be noted that during training and testing, we iterate through the labels and the examples of the dataset, but not through its attributes. Adding attributes is scalable since features are stored using $C++$ vector types. Additionally, the number of labels influences the amount of total computation completed, since the original Adaboost loops through through the labels during both testing and training.

Planned/Predicted Optimizations

Parallelization of Training

As stated before, we plan on implementing the parallel implementation of AdaBoost described by the paper **Scalable and Parallel Boosting with MapReduce**. Our predicted optimizations are derived from the results that the authors of this paper achieved. Their empirical results on twelve datasets show a consistent linear speedup, with the workers vs. speedup graph having a consistent slope of 0.8. Additionally, the machine specifications are similar to ours, with their experiment running on Intel Xeons as well. The main difference between our implementation and the paper's is that the MultiBoost library has serialization, which should cause our speedups to be less than that of the authors. We estimate that the upper bound on our speedup, if we max at 16 threads, will be $16 * 0.8 = 14.4$, though a speedup of $12x$ to $13x$ is more likely. We perform this parallelization using *pthreads*.

Parallelization of Testing

After implementing the parallel AdaBoost, as we suspected, the testing time significantly increased. This is because there are $2M$ times as many computations in the AdaBoost.PL implementation than there are in the AdaBoost.MH implementation. This is because computing α_t requires summing up M values, and computing $h^{(t)}(x)$ also requires summing up M values, both of which are only one operation in AdaBoost.MH, thus resulting in $2M$ times the computation (see **AdaBoost.PL Algorithm Mathematics** section). We decided to parallelize these computations using *pthreads*. In theory, to have AdaBoost.PL's testing run in the same time as AdaBoost.MH's testing, we would need $2M$ threads executing in parallel (not accounting for other overhead). We hypothesize that as we reach $2M$ threads, this will not scale, and that our AdaBoost.PL testing time can at best match, but nor surpass, the testing time of AdaBoost.MH.

Other optimizations

We aim to reduce cache misses while implementing the parallel algorithm. These should be minor changes, so the overall speedup will not be greatly affected from these additions. The output of a gprof profiler shows that the main work being done is in the testing method of the parallel algorithm, which is a short method that already exploits locality in C++ vectors. The code uses C++ vectors, so this takes advantage

of vectorization to perform computations using vector types. GPU optimizations could not be explored because that required changing the codebase to support nvcc and modify the Cmakefiles completely.

Experimental Results

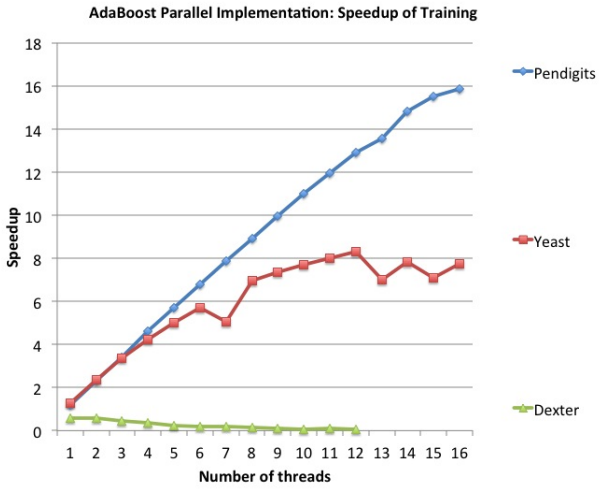
Confusion Matrix Performance Metrics

Between AdaBoost.MH and AdaBoost.PL, the comparison of the statistics mentioned in the **Experimental Setup** section fits within an interval of 0.12, with an average difference in percentages of ≈ 0.03. This shows that the AdaBoost.PL classifier performs similarly to the AdaBoost.MH classifier, which shows the algorithm's empirical correctness.

Speedup

The following graphs show the speedup of the training, testing, and (training + testing) times as the number of threads varies from 1 to 16:

Figure 4: Training Speedups



Evaluating Results

The first major result that can be seen in the graphs is the difference between the speedups for different datasets. This reiterates that different datasets cause different performance bottlenecks in the algorithm. For example, the dexter dataset performs poorly in terms of speedup on AdaBoost.PL as opposed to AdaBoost.MH for all thread values. This can be due to a variety of reasons. As mentioned before, serialization plays a bigger impact when the size of the dataset is very large. The dexter dataset contains 20000 features, but only 2600 examples. Thus, the parallel algorithm must load more data from each partition. This I/O operation of loading the file slows down the code much more than the work being done because the number of features is significantly higher than the number of examples.

Figure 5: Testing Speedups

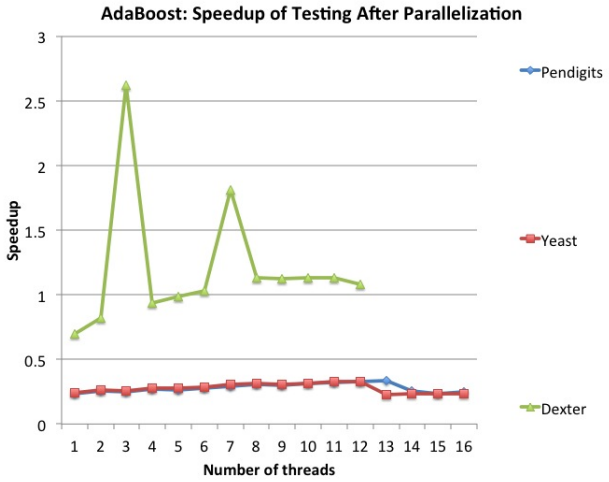
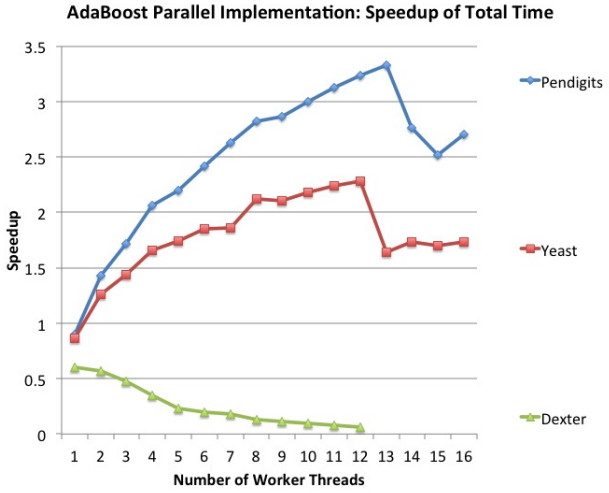


Figure 6: Total Speedups (Total = Testing + Training)



Examining the yeast and pendigits datasets shows the performance of running AdaBoost on a balanced dataset. The total time taken increases linearly as expected, but there are various performance bottlenecks in this code. For example, the biggest issue is false sharing in caches. Although each thread seems to be writing to different result vectors, the cache operates in line granularity. This would explain why performance decreases signicantly after around 12 threads. A hypothesis is that the difference in partition size from 12 to 13 threads causes false sharing among caches because a thread operates on a certain number of elements from this matrix. It then loads an entire cache line into the cache. Now, code for some threads is serialized because of the MESI protocol. A cache line is in the modified state, so the cache line will have to be invalidated so other threads don't read wrong memory. Even though the threads seem to be operating on independent data, false sharing causes inevitable serialization of the code.

The testing time for the dexter dataset performs much better than the other datasets. A possible explanation for this is that the number of labels in the dexter dataset is only 2, whereas the other datasets have 10 labels. The merge in the testing phase iterates through all these labels for each example. Since we are parallelizing this loop, it is expected that the testing phase speedup is high for the dexter dataset.

References

D. Benbouzid, N. C. F. C. B. K., R. Busa-Fekete. (2012, 3). Multiboost: A multi-purpose boosting package. *Journal of Machine Learning Research*(3), 549-553.

Dexter data set. (n.d.). https://archive.ics.uci.edu/ml/datasets/Yeast.

Freund, Y., & Schapire, R. (1999, 9). A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 771-780.

M. Abualkibash, A. M., A. ElSayed. (2013, 5). Highly scalable, parallel and distributed adaboost algorithm using light weight threads and web services on a network of multi-core machines. *International Journal of Distributed and Parallel Systems*, *4*(3), 771-780.

Palit, I., & Reddy, C. (2012, 10). Scalable and parallel boosting with mapreduce. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, *24*(10), 1904-1916.

Pendigits data set. (n.d.). https://archive.ics.uci.edu/ml/datasets/Pen-Based+Recognition+of+Handwritten+Digits.

Schapire, R. (n.d.). Explaining adaboost.

Schapire, R., & Singer, Y. (n.d.). Improved boosting algorithms using confidence-rated predictions.

Yeast data set. (n.d.). http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm.