Parallel Boosted Regression Trees for Web Search Ranking

Stephen Tyree
swtyree@wustl.edu

Kilian Q. Weinberger
kilian@wustl.edu

Kunal Agrawal
kunal@wustl.edu

Department of Computer Science & Engineering
Washington University in St. Louis
St. Louis, MO 63130

ABSTRACT

Gradient Boosted Regression Trees (GBRT) are the current state-of-the-art learning paradigm for machine learned web-search ranking — a domain notorious for very large data sets. In this paper, we propose a novel method for parallelizing the training of GBRT. Our technique parallelizes the construction of the individual regression trees and operates using the master-worker paradigm as follows. The data are partitioned among the workers. At each iteration, the worker summarizes its data-partition using histograms. The master processor uses these to build one layer of a regression tree, and then sends this layer to the workers, allowing the workers to build histograms for the next layer. Our algorithm carefully orchestrates overlap between communication and computation to achieve good performance.

Since this approach is based on data partitioning, and requires a small amount of communication, it generalizes to distributed and shared memory machines, as well as clouds. We present experimental results on both shared memory machines and clusters for two large scale web search ranking data sets. We demonstrate that the loss in accuracy induced due to the histogram approximation in the regression tree creation can be compensated for through slightly deeper trees. As a result, we see no significant loss in accuracy on the Yahoo data sets and a very small reduction in accuracy for the Microsoft LETOR data. In addition, on shared memory machines, we obtain *almost perfect linear speed-up* with up to about 48 cores on the large data sets. On distributed memory machines, we get a speedup of 25 with 32 processors. Due to data partitioning our approach can scale to even larger data sets, on which one can reasonably expect even higher speedups.

Categories and Subject Descriptors

D. Software [**D.0 General**]

General Terms

Algorithms

Keywords

Machine Learning, Ranking, Web Search, Distributed Computing, Parallel Computing, Boosting, Boosted Regression Trees

1. INTRODUCTION

The last two decades have witnessed the rise of the World Wide Web from an initial experiment at CERN to a global phenomenon. In this time, web search engines have come to play an important role in how users access the Internet and have seen tremendous advances. A crucial part of a search engine is the ranking function, which orders the retrieved documents according to decreasing relevance to the query.

Recently, web search ranking has been recognized as a supervised machine learning problem [7, 34], where each query-document pair is represented by a high-dimensional feature vector and its label indicates the document's degree of relevance to the query. In recent years, fueled by the publication of real-world data sets from large corporate search engines [21, 9], machine learned web search ranking has become one of the great success stories of machine learning.

Researchers around the world have explored many different learning paradigms for web-search ranking data, including neural networks [7], support vector machines [18], random forests [4, 22] and gradient boosted regression trees [34]. Among these various approaches, gradient boosted regression trees (GBRT) arguably define the current state-of-the-art: In the Yahoo Labs *Learning to Rank Challenge 2010* [9], the largest web-search ranking competition to date, *all* eight winning teams (out of a total of 1055) used approaches that incorporated GBRT. GBRT works by building an ensemble of regression trees, typically of limited depth. During each iteration a new tree is added to the ensemble, minimizing a pre-defined cost function.

Due to the increasing amount of available data and the ubiquity of multicores and clouds, there is increasing interest in parallelizing machine learning algorithms. Most prior work on parallelizing boosting [19, 32] is agnostic to the choice of the weak learner. In this paper, we take the opposite approach and parallelize the construction of individual weak learners, i.e. the depth-limited regression trees. Our algorithm is inspired by Ben-Haim and Yom-Tov's work on parallel construction of decision trees for classification [2].

In our approach, the algorithm works step by step constructing one layer of the regression tree at a time. One processor is designated the master processor and the others are the workers. The data are partitioned among the workers. At each step, the workers compress their portion of the data into small histograms and send these histograms to the master. The master approximates the splits using these histograms and computes the next layer in the regression tree. It then communicates this layer to the workers, which allows them to compute the histograms for the construction of subsequent layers. The construction stops when a predefined depth is reached.

This master-worker approach with bounded communication has several advantages. First, it can be generalized to multicores, shared memory machines, clusters and clouds with relatively little effort. Second, the data are partitioned among workers and each worker only accesses its own partition of the data. As the size of the data sets increases, it will become impossible for a single worker to hold all of the data in its memory. Therefore, data partitioning is imperative for performance, especially on clusters and clouds.

While adapted from [2], where they use a similar approach for classification trees, this approach is a natural fit for gradient boosting for two reasons. First, the communication between processors at each step is proportional to the number of leaves in the current layer. Therefore, it grows exponentially with the tree depth, which can be a significant drawback for full decision trees. However, since regression trees used for boosting are typically very small, this approach works well. Second, the small inaccuracies caused by the histogram approximations can be compensated for through marginally deeper trees (which are still much too small for inter-processor communication to have a noticeable effect) or a relatively small increase in the number of boosting iterations.

In this paper we make several novel contributions: 1. We adapt the histogram algorithms from Ben-Haim and Yom-Tov [2] from their original classification settings to squared-loss regression; 2. We incorporate this histogram-based feature compression into gradient boosted regression trees and derive a novel parallel boosting algorithm; 3. We demonstrate on real-world web-search ranking data that our parallelized framework leads to linear speed-up with increasing number of processors. Our experiments show no significant decrease in accuracy for the Yahoo Learning to rank competition (2010). In particular, on Set 1 (the larger data set), our parallel algorithm, within a matter of hours, achieves Expected Reciprocal Rank results that are within 1.4% of the best known results [6]. On the Microsoft LETOR data set, we see a small decrease in accuracy, but the speedups are even more impressive.

To our knowledge, we are the first paper to explicitly parallelize CART [5] tree construction for the purpose of gradient boosting. In addition, most previous work on tree parallelization, which parallelizes tree construction by features or by sub-trees [27], shows speedup of about 4 or 8. In contrast, our approach provides speed-up on limited depth trees of up to 42 on shared memory machines and up to 25 on distributed memory machines. Moreover, our approach is scalable to larger data sets and we expect even better speedups as the data set size increases.

This paper is organized as follows. In section 2, we introduce necessary notation and explain the general setup of machine learned web search ranking. Section 3 reviews gradient boosted regression trees and the greedy CART algorithm for tree construction. Section 4 shows that histograms can be used to approximate the exact splits of regression trees and introduces our parallel version of gradient boosted regression trees. Section 5 provides the empirical results for speedup and accuracy of our algorithm on several real-world web search ranking data sets. We relate our contribution to previous work in section 6 and conclude in section 7.

2. WEB SEARCH RANKING

Web ranking data consists of a set of web documents and queries. Each query-document pair is represented with a set of features which are generated using properties of both the query and the document. In addition, each pair is labeled, indicating how relevant the document is to the query. Using this data, the goal is to learn a regressor so that given a new query we can return the most relevant documents in decreasing order of relevance. In this paper, our algorithmic setup is not affected by the number of queries. Therefore, to simplify notation, we assume that all documents belong to a single query throughout the following sections. However, the techniques work for sets with multiple queries, and in fact the data we use for experiments does contain many queries.

More formally, we assume the data are in the form of instances $D = \{(\mathbf{x}_i, y_i)\}$ consisting of documents $\mathbf{x}_i \in \mathcal{R}^f$ and labels $y_i \in \{0, 1, 2, 3, 4\}$. A label indicates how relevant document $\mathbf{x}_i$ is to its query, ranging from "irrelevant" (if $y_i = 0$) to "perfect match" (if $y_i = 4$). A document is represented by an f dimensional vector of features that are computed from the document and the query. Our goal is to learn a regressor $h : \mathcal{R}^f \to \mathcal{R}$ such that $h(\mathbf{x}_i) \approx y_i$. At test time, the search engine ranks the documents $\{\mathbf{x}_j\}_{j=1}^m$ of a new query in decreasing order of their predicted relevance $\{h(\mathbf{x}_j)\}_{j=1}^m$.

The quality of a particular predictor $h(\cdot)$ is measured by specialized ranking metrics. The most commonly used metrics are Expected Reciprocal Rank (ERR) [10], which is meant to mimic user behavior, and Normalized Discounted Cumulative Gain (NDCG) [17], which emphasizes leading results (NDCG). However, these metrics can be non-convex, non-differentiable or even non-continuous. Although some recent work [33, 28, 11] has focused on optimizing these ranking metrics directly, the more common approach is to optimize a well-behaved surrogate cost function $\mathcal{C}(h)$ instead, assuming that this cost function mimics the behavior of these other metrics.

In general, the cost functions $\mathcal{C}$ can be put into three categories of ranking: pointwise [14], pairwise [16, 34] and listwise [8]. In pointwise settings the regressor attempts to approximate the label y_i of a document $\mathbf{x}_i$ directly, *i.e.* $h(\mathbf{x}_i) \approx y_i$. A typical loss function is the squared-loss

$$\mathcal{C}(h) = \sum_{i=1}^{n} (h(\mathbf{x}_i) - y_i)^2. \qquad (1)$$

The pairwise setting is a relaxation of pointwise functions, where pairs of points are considered. It is no longer important to approximate each relevance score exactly, rather the partial order of any two documents should be preserved. An example is the cost function of GBRANK [34],

$$\mathcal{C}(h) = \sum_{(i,j) \in \mathcal{P}} \max(0, 1 - (h(\mathbf{x}_i) - h(\mathbf{x}_j)))^2, \qquad (2)$$

where $\mathcal{P}$ is the preference set of all document pairs (i, j) belonging to the same query, where i should be preferred over j. Listwise approaches [8] are similar to the pairwise approach, but focus on all the documents that belong to a particular query and tend to have slightly more complicated cost functions. Recent research [20] also focused on breaking the ranking problem into multiple binary classification tasks.

In this paper, we do not restrict ourselves to any particular cost function. Instead we assume that we are provided with a generic cost function $\mathcal{C}(\cdot)$, which is continuous, convex and at least once differentiable.

3. BACKGROUND

In this section we first review Gradient boosting [14] as a general meta-learning algorithm for function approximation and then remind the reader of regression trees [5].

Gradient Boosting

Gradient boosting [14] is an iterative algorithm to find an additive predictor $h(\cdot) \in \mathcal{H}$, that minimizes $\mathcal{C}(h)$. At each iteration t, a new function $g_t(\cdot)$ is added to current predictor $h_t(\cdot)$, such that after T iterations, $h_T(\cdot) = \alpha \sum_{t=1}^{T} g_t(\cdot)$, where $\alpha > 0$ is some non-negative learning rate. Function $g_t(\cdot)$ is chosen as an approximation of the negative derivative of the cost function in function space, $g_t(\cdot) \approx -\frac{\partial C}{\partial h_t(\cdot)}$.

In order to find an appropriate function $g_t(\cdot)$ in each iteration t, we assume the existence of an oracle $\mathcal{O}$. For a given function class $\mathcal{H}$ and a set $\{(\mathbf{x}_i, r_i)\}$ of document and target pairs, this oracle returns the function $g \in \mathcal{H}$, that best approximates the response values according to the squared loss (up to some small $\epsilon > 0$):

$$\mathcal{O}(\{(\mathbf{x}_i, r_i)\}) \approx \operatorname*{argmin}_{g \in \mathcal{H}} \sum_i (g(\mathbf{x}_i) - r_i)^2. \qquad (3)$$

In iteration t, gradient boosting attempts to find the function $g(\cdot)$ such that $\mathcal{C}(h_t + g)$ is minimized. Let us approximate the inner-product between two functions $\langle f(\cdot), g(\cdot) \rangle$ as $\sum_{i=1}^{n} f(\mathbf{x}_i) g(\mathbf{x}_i)$. By the first-order Taylor Expansion,

$$g \approx \operatorname*{argmin}_{g \in \mathcal{H}} \left[\mathcal{C}(h) + \sum_{i=1}^{n} \frac{\partial C}{\partial h_t(\mathbf{x}_i)} g(\mathbf{x}_i) \right]. \qquad (4)$$

If we assume the norm of $g \in \mathcal{H}$ is constant,[1] *i.e.,* $\langle g, g \rangle = c$, the solution of the minimization (4) becomes $\mathcal{O}(\{(\mathbf{x}_i, r_i)\})$, where $r_i = -\frac{\partial C}{\partial h_t(x_i)}$. (This follows from a simple binomial expansion of (3), where the two quadratic terms are constants and therefore independent of g.) In the case where $\mathcal{C}(\cdot)$ is the squared-loss from eq. (1), the assignment becomes the current residual $r_i = y_i - h_t(i)$. Algorithm 1 summarizes gradient boosted regression in pseudo-code.

In web-search ranking, and other domains, the most successful choice for the oracle in (3) is the greedy Classification and Regression Tree (CART) algorithm [5] with limited depth size d, often 4 or 5. This paper focuses exclusively on the parallelization of limited-depth gradient boosted regression trees (GBRT). We review the basic CART algorithm in the following section.

Regression Trees

Regression tree construction in CART [5] proceeds by repeated greedy expansion of nodes until a stopping criterion, e.g. tree depth, is met. Initially, all data points are assigned to a single node, the root of the tree. To simplify notation, we explain the algorithm for data with one single feature only and drop the feature indices (*i.e.* x_i is the feature value for input i). In practice, the same steps are performed for all features, and the feature that leads to the split with the maximum reduction in cost is chosen.

[1]We can avoid this restriction on the function class $\mathcal{H}$ with a second-order Taylor expansion in eq. (4). We omit the details of this slightly more involved derivation in this paper, as it does not affect our algorithmic setup. However, we do refer the interested reader to [34].

Algorithm 1 Gradient Boosted Regression Trees

```
Input: data set $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$, Parameters: $\alpha$, $m$, $d$
Initialization: $r_i = y_i, \ \forall i$
$\mathcal{C}(\cdot) = 0$
for $t = 1$ to $m$ do
  $g_t \leftarrow \mathcal{O}(\{(\mathbf{x}_i, r_i\})$
  $\mathcal{C}(\cdot) \leftarrow \mathcal{C}(\cdot) + \alpha g_t(\cdot)$
  for $i = 1$ to $n$ do
    $r_i \leftarrow -\frac{\partial C}{\partial h_t(\mathbf{x}_i)}$
  end for
end for
return $\mathcal{C}$
```

Algorithm 2 Parallel CART Master

```
Parameter: maximum depth
tree ← unlabeled node
while tree depth < maximum depth do
  for each feature do
    instantiate an empty histogram at each leaf
    for each worker do
      initiate non-blocking receive for worker's histograms
    end for
    while ∃ non-completed receives do
      wait for some receive to complete
      merge received histograms at leaves
    end while
    update best splits for each leaf
  end for
  send next layer of leaves to each worker
end while
return tree
```

At every node, with data $S \subseteq D$, the prediction made by the classifier is the average label in S. Denote this predictor as $\bar{y}_S = \frac{1}{|S|} \sum_{(x_i, y_i) \in S} y_i$. It is straight-forward to show that $\bar{y}_S$ minimizes the loss of a constant predictor over a set S:

$$\bar{y}_S = \operatorname*{argmin}_{p} \sum_{i \in S} (y_i - p)^2. \qquad (5)$$

At each step, the data $S \subseteq D$ assigned to each leaf node node is partitioned into two subsets $L^s, R^s \subset S$ by splitting on a feature value s. We define the two subsets $L^s = \{(x_i, y_i) \in S \mid x_i < s\}$ and $R^s = S - L^s$. Let $j_1, \ldots, j_{|S|}$ be the indices of the ordered set S such that $x_{j_i} \leq x_{j_{i+1}}$. Therefore, we have $|S|-1$ potential split points, one between each consecutive element. The optimal value s^* is found by iterating over these possible split values P and evaluating the accumulative squared loss on both sides of the tree:

$$\begin{aligned} s^* &= \operatorname*{argmin}_{s \in P} \sum_{i \in L^s} (y_i - \bar{y}_L^s) + \sum_{i \in R^s} (y_i - \bar{y}_R^s) \qquad (6) \\ \text{where: } P &= \left\{ \frac{x_{j_{i+1}} - x_{j_i}}{2} \middle| 1 \leq i < |S| \right\} \end{aligned}$$

Eq. (6) can be solved in $O(n)$ time through dynamic programming [5]. However, the dynamic programming approach requires the samples to be sorted, which in itself requires $O(n \log(n))$ operations.

4. PARALLEL IMPLEMENTATION

In this section, we describe our approach for parallelizing the construction of gradient boosted regression trees.

Algorithm 3 Parallel CART Worker

```
Input: data set $D = \{(\mathbf{x}_i, r_i)\}_{i=1}^n$
Parameter: maximum depth
tree ← unlabeled node
while tree depth < maximum depth do
    navigate training data $D$ to leaf nodes $v$
    for each feature $f$ do
        instantiate an empty histogram $h_v$ for each leaf $v$
        for $(\mathbf{x}_i, r_i) \in D$ at leaf $v_i$ do
            merge($h_{v_i}$, $([\mathbf{x}_i]_f, 1, r_i)$)
        end for
        initiate non-blocking send for histograms from all leaves
    end for
    receive next layer of leaves from master
end while
```

In this approach, the boosting still occurs sequentially, as we parallelize the construction of individual trees. Two key insights enable our parallelization. First, in order to evaluate a potential split point during tree construction we only need cumulative statistics about all data left and right of the split, but not these data themselves. Second, boosting does not require the weak learners to be particularly accurate. A small reduction in accuracy of the regression trees can potentially be compensated for by using more trees without affecting the accuracy of the boosted regressor.

In our algorithm, we have a master processor and P workers. We assume that the data are divided into P disjoint subsets stored in different physical locations and each worker can access one of these locations. Let D_p be the set of data instances stored at site p, so that $\bigcup_{p=1}^{P} D_p = D$, $D_p \cap D_q = \emptyset$ for $p \neq q$ and $|D_p| \approx |D|/P$. The master processor builds the regression trees layer by layer. At each iteration, a new layer is constructed as follows: Each worker compresses its share of the data feature-wise using histograms (as in [2]) and sends them to the master processor. The master merges the histograms and uses them to approximate the best splits for each leaf node, thereby constructing a new layer. Then the master sends this new layer to each worker, and the workers construct histograms for this new layer. Therefore, the communication consists entirely of the workers sending histograms to the master and the master sending a new layer of the tree to the workers. Since the depth of the regression trees is small, the amount of communication is small.

We now explain this algorithm using three steps: First, we identify cumulative statistics that are sufficient to train a regression tree over a data set. Second, we describe how we can construct histograms with a single pass over the data and use them to approximate the cumulative statistics and the best split. Finally, we describe the algorithms that run on the master and workers and how we overlap computation and communication to achieve good parallel performance.

Cumulative Statistics

We wish to evaluate (6) with cumulative statistics about the data set. Consider the setting from Section 3. A data set S is split along a feature value s into $L^s, R^s \subseteq S$. Let ℓ_s denote the sum of all labels and m_s the number of inputs within L^s:

$$\ell_s = \sum_{(x_i, y_i) \in L^s} y_i \quad \text{and} \quad m_s = |L^s|. \tag{7}$$

With this notation, the predictors $\bar{y}_L^s, \bar{y}_R^s$ for the left and right subset can be expressed as

$$\bar{y}_L^s = \frac{\ell_s}{m_s} \quad \text{and:} \quad \bar{y}_R^s = \frac{\ell_\infty - \ell_s}{m_\infty - m_s}. \tag{8}$$

Expanding the squares in (6) and substituting the definitions from (7) and (8) allows us to write the optimization to find the best splitting value s^* entirely in terms of the cumulative statistics ℓ_s and m_s:

$$s^* = \operatorname*{argmin}_{s \in P} -\frac{\ell_s^2}{m_s} - \frac{(\ell_\infty - \ell_s)^2}{m_\infty - m_s}. \tag{9}$$

Since ℓ_∞ and m_∞ are constants, in order to evaluate a split point s, we only require the values of ℓ_s and m_s.

Histograms

The traditional GBRT algorithm, as described in Section 3, spends the majority of its computation time evaluating split points during the creation of regression trees. We speed up and parallelize this process by summarizing label and feature-value distributions using histograms. Here we describe how a single split, evaluated on the data that reaches that particular node, is computed using these histograms.

Ben-Haim and Yom-Tov [2] introduced a parallel histogram-based decision tree algorithm for classification. A histogram h, over a data set S, is a set of tuples $h = \{(p_1, m_1), ..., (p_b, m_b)\}$, where each tuple (p_j, m_j) summarizes a bin $B_j \subseteq S$ containing $m_j = |B_j|$ inputs around a center $p_j = \frac{1}{m_j}\sum_{x_i \in B_j} x_i$. In the original algorithm, as described in [2], each processor summarizes its data by generating one histogram per label.

We lack discrete labels after the first iteration of boosting, so we cannot have different histograms for each label. Instead, our histograms contain triples, (p_j, m_j, r_j), where $r_j = \sum_{i, x_i \in B_j} y_i$ is the cumulative document relevance of the j^{th} bin. Recall that for simplicity in explanation, we are assuming that there is a single feature in the data. In reality, both classification and regression approaches generate different histograms for each feature.

Construction: A histogram h can be built over a data set S in a single pass. For each input $(x_i, y_i) \in S$, a new bin $(x_i, 1, y_i)$ is added to the histogram. If the size of the histogram exceeds a predefined maximum value b^* then the two nearest bins are merged. The nearest bins

$$i, j = \operatorname*{argmin}_{i', j'} |p_{i'} - p_{j'}| \tag{10}$$

are replaced by a single bin:

$$\left(\frac{m_i p_i + m_j p_j}{m_i + m_j}, m_i + m_j, r_i + r_j \right). \tag{11}$$

Once the workers send all of their respective histograms to the master, the master merges them by applying the same merging rule, i.e., merging bins of one histogram into another one by one.

Interpolation: Given the compressed information from the merged histogram $h = \{(p_j, m_j, r_j)\}_{j=1}^b$, $b \leq b^*$, we need to approximate the values of m_s and ℓ_s to find the best split point s^*, as defined in (9). Let us first consider the case where $s = p_j$ for some $1 \leq j \leq b$, i.e., we are evaluating a split exactly at a centroid of a bin. To approximate the values of m_{p_j} and ℓ_{p_j}, we assume that the number of points and relevances are evenly distributed around the mean p_j. In other words, half of the points m_j and half of the total relevance r_j lies on the left side of p_j. The approximations then become:

$$m_{p_j} \approx \sum_{k=1}^{j-1} m_k + \frac{m_j}{2} \text{ and } \ell_{p_j} \approx \sum_{k=1}^{j-1} r_k + \frac{r_j}{2}. \qquad (12)$$

If a potential splitting point s is not at the center of a bin, *i.e.* $p_i < s < p_{i+1}$, we interpolate the values of m_s and ℓ_s.

Let us consider ℓ_s first. As we already have ℓ_{p_i}, all we need to compute is the remaining relevance $\Delta = \ell_s - \ell_{p_i} = \sum_{p_i \leq x_i < s} r_i$, so $\ell_s = \ell_{p_i} + \Delta$. Following our assumption that the points around bins i and $i+1$ are evenly distributed, there is a total relevance of $R = \frac{r_i + r_{i+1}}{2}$ within the bin centers p_i and p_{i+1}. We assume this total relevance is evenly distributed within the area under the histogram curve between $[p_i, p_{i+1}]$. Let $a(s) = \int_{p_i}^{s} h(x)\partial x$ be the area under the curve within $[p_i, s]$. The sum of relevance within $[p_i, s]$ is then proportional to $a(s)/a(p_{i+1})$. We use the trapezoid method to approximate the integral $a(s)$ and interpolate ℓ_s:

$$\ell_s = \ell_{p_i} + \frac{a_r(s)}{a_r(p_{i+1})} R,$$

where:

$$a_r(s) \approx \frac{(r_i + r_s)(s - p_i)}{2} \text{ and}$$

$$r_s = r_i + \frac{r_{i+1} - r_i}{p_{i+1} - p_i}(s - p_i).$$

The interpolation of m_s is analogous to (13), except that m_x are substituted for all r_x.

Now that we can interpolate cumulative statistics ℓ_s and m_s from histograms for arbitrary split points s, there are potentially infinite number of candidate split points. We select the set of candidate split points $\{s_i\}$ positioned uniformly on the distribution of $x_i \in S$. These uniformly distributed points may be estimated by the Uniform procedure described in [2].

The use of histograms (even on a single CPU) speeds up the GBRT training time significantly, as it alleviates the need to sort the features to identify all possible split points. The time complexity for tree construction reduces from $O(n \log(n))$ to $O(n \log(b))$, where $b \ll n$.

Distributed GBRT

In addition to an inherent speedup in tree construction, histograms allow the construction of regression trees to proceed in parallel. Worker nodes compress distributed subsets of the data into small histograms. These, when merged, represent an approximate, compressed view of an entire data set, and can be used to compute the split points. We now explain our distributed algorithm in more detail. In order to explain the algorithm more completely, we no longer assume that there is just a single feature.

A layout in pseudo-code for the master and worker are depicted in Algorithms 2 and 3. As mentioned above, the data are partitioned into P sets, one for each worker. The workers are responsible for constructing histograms and the master is responsible for merging the histograms from all workers and finding the best split points. Initially, the tree consists of a single root node. At each step, the master finds the best split for all the current leaf nodes, generating a new layer of leaves, and sends this new layer to the workers. The workers first evaluate each data point in their partition and navigate it to the correct leaf node. They then create an empty histogram per feature per leaf. They summarize their respective data partition in these histograms (using the histogram construction algorithm described above) and send them to the master. The master merges these histograms to split the leaves again.

The number of histograms generated per iteration is proportional to the number of features times the number of leaves, $O(f \times 2^{d-1})$, where d is the current depth of the tree. As d increases, this communication requirement may become significant and overwhelm the gains made from parallelization. However, for small depth trees and sufficiently large data sets, we achieve drastic speedups since the majority of computation time is spent by the workers in compressing the data. In addition, notice that the number of histograms does not depend on the size of the data set, and therefore, the algorithm is scalable as the size of the data increases. Further, communication volume is tunable by the compression parameter b^* which sets the maximum number of bins in each histogram. At a possible sacrifice of accuracy, smaller histograms may be used to limit communication.

When using such master/worker message passing algorithms, it is important to consider two objectives in order to achieve good performance. First, the computation and communication should be overlapped so processors don't block on communication. Second, the number of messages sent should not be excessive since initializing communication carries some fixed cost. Our algorithm is designed to carefully balance these two objectives. In order to overlap communication and computation, the workers compute the histograms feature by feature and send these histograms to the master while they move on to the next feature. In order to allow this with just one pass over the data, we store our data feature-wise, that is, the values of a particular feature for all instances are stored contiguously. To avoid generating unnecessarily small messages, all the histograms corresponding to a particular feature, across all leaves, are sent as a single message. That is, instead of sending one message per histogram, the worker generates one message per feature. Therefore, the number of messages does not increase with depth, even though the size of messages does increase.

5. EXPERIMENTAL RESULTS

In this section, we describe the empirical evaluation of our algorithm using two publicly available web search ranking data compilations. We see impressive speedups on both shared memory and distributed memory machines. In addition, we found that, while the individual regression trees are weaker using our approximate parallel algorithm (as expected), with appropriate parameter settings, the final regressor did not lose much accuracy. In some cases, our parallel implementation generates a regressor that is just as good as the sequential implementation, while in others, it was slightly less accurate.

Features: For web data, each query-document pair is represented by a vector of features. This vector typically consists of three parts:

Query-feature vector, consists of features that depend only on the query q and have the same value across all the documents d in the document set. Examples of such features are the number of terms in the query, whether or not the query is a person name, etc.

Document-feature vector, consists of features that depend

TRAIN	Yahoo LTRC Set 1	Set 2	MSLR MQ2008 Folds F1	F2	F3	F4	F5
# Features	700	700	136	136	136	136	136
# Documents	473,134	34,815	723,412	716,683	719,111	718,768	722,602
# Queries	19,944	1266	6000	6000	6000	6000	6000
Avg # Doc per Query	22.723	26.5	119.569	118.447	118.852	118.795	119.434
TEST	Set 1	Set 2	F1	F2	F3	F4	F5
# Documents	165,660	103,174	241,521	241,988	239,093	242,331	235,259
# Queries	6983	3798	6000	6000	6000	6000	6000
Avg # Doc per Query	22.723	26.165	119.761	119.994	118.547	120.167	116.6295

Table 1: Statistics of the Yahoo Competition and Microsoft Learning to Rank data sets.

only on the document d and have the same value across all the queries q in the query set. Examples include the number of inbound links pointing to the document, the amount of anchor-texts (in bytes) for the document, and the language identity of the document, etc.

Query-document feature vector, consists of features that depend on the relationship between the query q and the document d. Examples are the number of times each term in the query q appears in the document d, the number of times each term in the query q appears in the anchor-texts of the document d, etc.

Data Sets: For our empirical evaluation, we use the two data sets from Yahoo! Inc.'s *Learning to Rank Challenge 2010* [9], and the five folds of Microsoft's LETOR [21] dataset. Each of these sets come with predefined training, validation and test sets. Table 1 summarizes the statistics of these data sets.

Experimental Setup: We conducted experiments on a parallel shared memory machine and a distributed memory cluster. The shared memory machine is an AMD Opteron 1U-A1403 48-core SMP machine with four sockets containing AMD Opteron 6168 Magny-Cours processors. The distributed memory cluster consists of 8-core, Nehalem based computing nodes running at 2.73GHz. They each have 24GB of RAM. For our experiments, we used 6 of these nodes (with a total of 48 cores).

On both machines, we tested the same implementation of the algorithm, which was built with MPI [26]. We will make the code available[2] under an open source license. We compare against the exact GBRT implementation[3] described in [22], which to our knowledge is currently the only large-scale open-source GBRT implementation.

For simplicity, we used the squared-loss as our cost-function $\mathcal{C}(\cdot)$ in all experiments. Our algorithm has four parameters: The depth of the regression trees d, the number of boosting iterations m, the step-size α and the number of bins b in the histograms. We perform most experiments on the sensitivity of these parameters only on the Yahoo Set 1 and 2, as these span two different ranges of data set sizes (Set 1 is almost one order of magnitude larger than Set 2).

Prediction Accuracy: As a first step, we investigate how much the ranking performance, measured in ERR [10] and NDCG [17], is impacted by the approximated construction of the regression trees.

Figure 1 shows the ERR and NDCG of the parallel implementation "pGBRT" and of the exact algorithm "GBRT" as a function of the number of boosting iterations on the Yahoo Set 1 and 2 under varying tree depths. For the parallel implementation, we used $b = 25$ bins for Set 2 and $b = 50$ for the much larger Set 1. The step-size was set to $\alpha = 0.06$ in both cases.

As expected, the histogram approximation reduces the accuracy of the weak learners. Consequently, with equal depth and iterations, pGBRT has lower ERR and NDCG than the exact GBRT. However, we can compensate for this effect by either running additional iterations or increasing the depth of the regression trees. In fact, it is remarkable that on Set 1 (Figure 1) the NDCG curves of pGBRT with $d = 6$ and $d = 5$ align almost perfectly with the curves of GBRT with $d = 5$ and $d = 4$, respectively. For Set 2 the lines are mostly shifted by approximately 200 iterations. The additional computation required by either of these approaches (increasing d or m) is more than compensated for by the better performance of the histogram method since it does not require feature sorting. (For small $d \leq 10$ – while the computation is dominated by computation – the running time increases roughly linearly with increasing d. On the Yahoo Set 1, training pGBRT with $m = 6000$ trees on 16 CPUs and depth $d = 5$ was only a factor 1.34 slower than $d = 4$ and a depth of $d = 6$ slowed the training time down by a factor of 1.75.)

Table 2 shows the test set results on all data sets for GBRT with $d = 4$ and pGBRT for $d = 4, 5, 6$. The number of trees m was picked with the help of the corresponding validation data sets. As the table shows, for all data sets, the difference between pGBRT with $d = 6$ and GBRT with $d = 4$ is in the third significant digit for all data sets. For the two Yahoo data sets, pGBRT provides slightly better accuracy, while for the Microsoft data sets, the exact algorithm is slightly better. The Microsoft data sets were run with parameters $\alpha = 0.1$, $b = 100$ and $m \leq 5000$. We increased the number of bins since the data set is larger.

We also evaluated the sensitivity of the algorithm to the number of histogram bins b and the number of processors. Figure 2 shows several runs ($\alpha = 0.05, d = 5$) with varying numbers of histogram bins assessed by quality measures which have been scaled by the best observed accuracy for each measure. For each run we report the best result as selected on the validation data set. We see that while the prediction accuracy increases slightly as the number of bins increases, it converges quickly at about 15 bins, and the differences thereafter are insignificant. In a similar experiment (not shown) we measured prediction accuracy while varying the number of processors. Despite more histograms being inexactly merged with each additional processor, we did not

[2] `http://research.engineering.wustl.edu/~tyrees/`
[3] `http://research.engineering.wustl.edu/~amohan/`

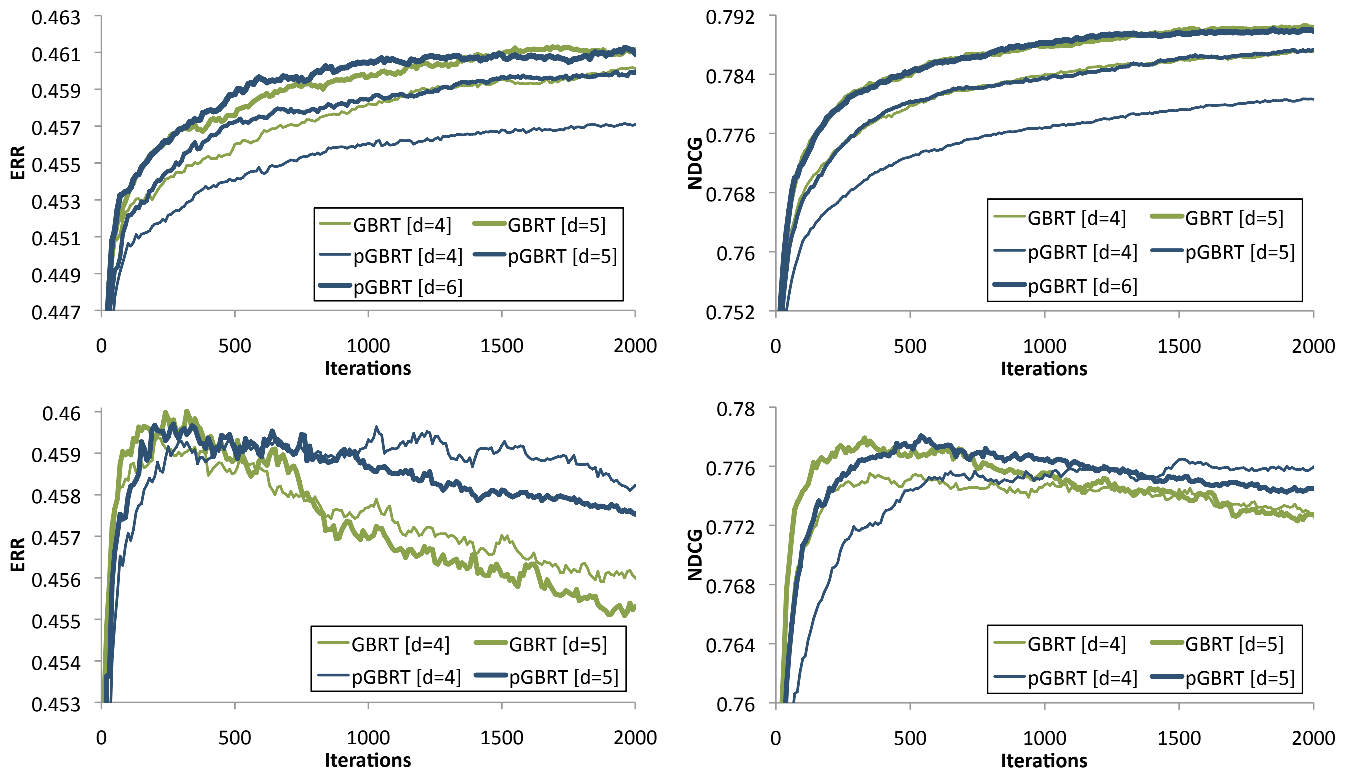

Figure 1: ERR and NDCG for Yahoo Set 1 (*top*) and Yahoo Set 2 (*bottom*) on parallel (pGBRT) and exact (GBRT) implementations with various tree depths d. The NDCG plot for Set 1 *(top right)* shows nicely that pGBRT with a tree depth of $d+1$ leads to results similar to the exact algorithm with depth d.

observe any noticeable drop in accuracy as the number of processors increased.

To demonstrate that our algorithm is competitive with the state-of-the-art, we selected the parameters m, d, b by cross-validation on the validation sets of both Yahoo data sets (for simplicity we fixed $\alpha = 0.06$, as GBRT is known to be relatively insensitive to the step-size). This yielded test set ERR of 0.4614 on Set 1 ($m = 3926$, $d = 7$, $b = 100$) and 0.4596 on Set 2 ($m = 3000$, $d = 5$, $b = 50$). The ERR score of Set 1 would have placed us 15^{th} (and 14^{th} for Set 2) on the leaderboard of the 2010 Yahoo Learning to Rank Challenge[4] out of a total of 1055 competing teams. This result – despite our simple squared-loss cost function – is only 1.4% below the top scoring team, which used an ensemble of specialized predictors and fine-tuned cost functions that explicitly approximate the ERR metric [6]. We assume that better scores could be achieved with our parallel algorithm as well by using more specialized cost functions, however this is beyond the scope of this paper.

Performance and Speedup: For performance measurements, we trained pGBRT for $m = 250$ trees of depth $d = 5$ using histograms with $b = 25$ bins. Figure 3 shows the speedup of our pGBRT algorithm on both the Yahoo and the LETOR Fold 1 while running on the shared memory machine. For the smaller data set (Set 2), we achieve speedup of up to 10 on 13 cores. For the larger data set (Set 1), we achieve much higher speedups, up to 33 on 41 processors, reducing the training time on Set 1 from over 11 hours to merely 21 minutes. On the Microsoft data (almost twice as many samples as Yahoo Set 1), we see the speedup of up to 42 on 48 cores, and there is potential for more speedup on more cores since the line hasn't plateaued yet. We see more speedup on the Microsoft data since it has fewer features (requires less communication) and more documents (increases the fraction of time spent on tree construction). While Set 1 and LETOR are among the largest publicly available data sets, proprietary data sets are potentially much larger, and we expect further speedup on those.

Figure 4 shows the speedup of our parallel GBRT on both the Yahoo and the LETOR datasets while running on the cluster. As expected, the speedup is smaller on the distributed memory machine due to communication latency. However, we still see the speedup of about 20 with Yahoo! Set 1 and about 25 with the Microsoft data on 32 cores, after which point the performance flattens out.[5] This result demonstrates the generality of our parallelization methods in that the same strategy (and even the same code) can provide impressive speedups on a variety of parallel machines.

All speed-up results are reported relative to the sequential pGBRT version (1 helper CPU). We do not report speedup

[4]`http://learningtorankchallenge.yahoo.com/leaderboard.php`

[5]We see some performance irregularities (in the form of zigzags) for the LETOR data set. We are investigating these irregularities and suspect that they are due to caching and memory bandwidth effects.

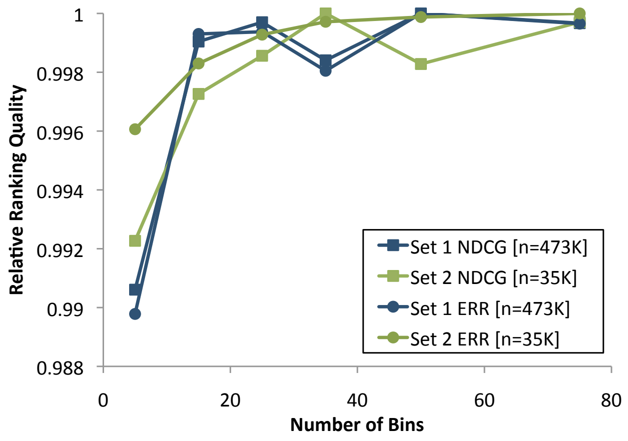

Figure 2: Ranking performance of pGBRT on the Yahoo Set 1 and 2 as a function of varying number of histogram bins b. With $b \geq 20$ both metrics are less than a factor 0.004 away from the best value.

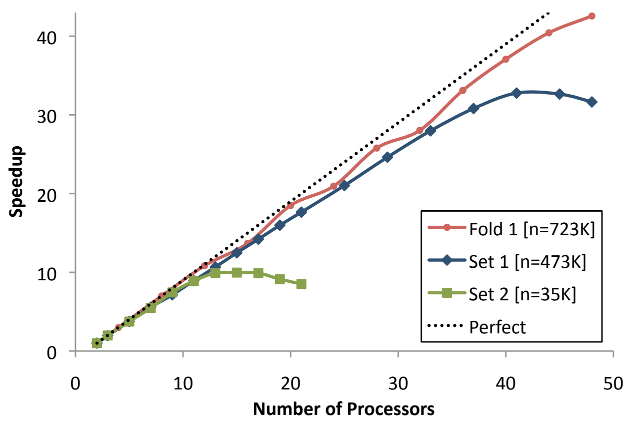

Figure 3: The speedups of pGBRT on a multicore shared memory machine as a function of cpu cores. The speedup increases with data set size and is almost perfectly linear for Yahoo Set 1 and the Microsoft LETOR data set. The latter set could potentially obtain even higher speedup with more cores.

compared to the exact algorithm, since this codebase uses different data structures and timing results might not be representative. In general, however, the speed-up with respect to the 1-CPU pGBRT runs understate the speedup over the exact algorithm, since the exact algorithm available to us is considerably slower than pGBRT, even on a single processor. For comparison, "sequentially" (1 helper CPU), our parallel algorithm completes execution in $3178s$ on Yahoo Set 2 and $43{,}189s$ on Set 1, both with depth 5. Even with a smaller depth 4, the exact GBRT implementation takes $5940s$ on Set 2 and $259{,}613s$ on Set 1. Particularly for Set 1, the exact algorithm is about 6 times slower than the approximate algorithm even when running with a smaller depth.[6]

6. RELATED WORK

In this section, we present a sample of previous work on parallel machine learning most related to our work. The related work falls into three categories: parallel decision trees, parallelization of boosting, and parallelization of web search ranking using other approaches such as bagging.

Parallel decision tree algorithms have been studied for many years, and can be grouped into two main categories: task-parallelism and data-parallelism. Algorithms in the first category [12, 27] divide the tree into sub-trees, which are constructed on different workers, e.g. after the first node is split, the two remaining sub-trees are constructed on separate workers. There are two downsides of this approach. First, each worker should either have a full copy of the data or large amount of data has to be communicated to workers after each split. Therefore, for large data sets, especially if the entire data set doesn't fit in each worker's memory, this scheme would likely provide slowdown rather than speedup. Second, small trees are unlikely to get much speedup, since they cannot utilize all the available workers. Our algorithm falls into the second approach [1], data-parallelism, where the training data are divided among the different workers. The data can be partitioned by features [13], by samples [25] or both [31]. Dividing by feature requires the workers to coordinate which input falls into which tree-node, as the individual workers do not have enough information to compute it locally. This requires $O(n)$ communication, which we try to avoid as we scale to very large data sets. Dividing the data by samples [25] avoids this problem. However, in order to obtain the exact solution, all nodes are required to evaluate the potential split points found by all other nodes [31]. Our approach distributes the data by samples and deliberately only approximates the exact split, making the communication requirement *independent* of the data set size.

Two sample-partitioning approaches bear similarities to our work. PLANET [23] selects splits using exact, static histograms constructed in a two stage process. Implemented in the MapReduce framework, PLANET samples histogram bin boundaries to achieve approximately uniform-depth bins and then tallies exact data counts and label sums for each bin. Initially we implemented a similar scheme, but later achieved better accuracy with a single stage process and our dynamic regression-oriented histograms.

Our algorithm is most similar to Ben-Haim and Yom-Tov's work on parallel approximate construction of decision trees for classification [2]. Our histogram methods were largely inspired by their publication. However, our approach differs in several ways. First, we use regression trees instead of classification – requiring us to interpolate relevance scores within histogram bins instead of computing one histogram per label. Further, our method explicitly parallelizes gradient boosted regression trees with a fixed small depth. The communication required for workers to exchange the feature-histograms grows exponentially with the depth of the tree. In our experiments, for trees with depth $d \geq 15$ (consisting of over 65,535 tree nodes), we saw a slowdown (instead of speedup) due to increased communication. This drastically reduces the benefit of parallelization of full decision or

[6]Exact implementations with clever bookkeeping [15] may be faster – however, to our knowledge, no large-scale implementations are openly available. We expect, nonetheless, that our algorithm can be optimized to perform, on a single CPU, comparably or faster than optimized exact implementations, since it does not require feature sorting.

ERR method	Yahoo LTRC Set 1	Set 2	MSLR MQ2008 Folds F1	F2	F3	F4	F5
GBRT (d=4)	0.461	0.458	0.361	0.358	0.355	0.367	0.373
pGBRT (d=4)	0.458	0.459	0.346	0.341	0.342	0.343	0.357
pGBRT (d=5)	0.460	0.460	0.355	0.348	0.355	0.353	0.367
pGBRT (d=6)	0.461	0.460	0.355	0.354	0.357	0.363	0.367
NDCG method	Yahoo LTRC Set 1	Set 2	MSLR MQ2008 Folds F1	F2	F3	F4	F5
GBRT (d=4)	0.789	0.765	0.495	0.493	0.484	0.498	0.500
pGBRT (d=4)	0.782	0.743	0.474	0.469	0.466	0.473	0.479
pGBRT (d=5)	0.785	0.754	0.483	0.479	0.479	0.484	0.491
pGBRT (d=6)	0.785	0.760	0.486	0.484	0.482	0.491	0.495

Table 2: Results in ERR and NDCG on the Yahoo and Microsoft data sets. The number of boosting iterations is selected with the validation data set. On both Yahoo sets, pGBRT matches the result of GBRT with $d = 4$ when the tree depth is increased. For the Microsoft sets, the ranking results tend to be slightly lower.

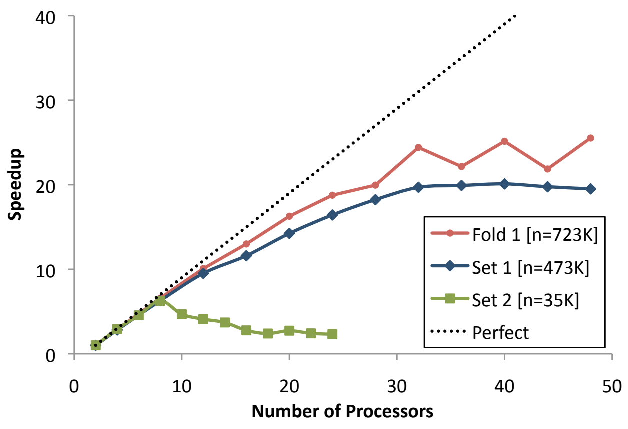

Figure 4: The speedups of pGBRT on the cluster as a function of CPU cores. We observe up to 25 fold speedups in this distributed setting for the Microsoft LETOR data set.

regression trees on large data sets, since the required tree depth grows with increasing data set size. In contrast, our framework deliberately fixes the tree-depth to a small value (e.g. d between 4 and 6 with 63 to 255 tree-nodes). Instead of deeper trees, larger data sets require more boosting iterations, which is not a problem for us. As shown in Figure 3, our pGBRT algorithm obtains *more* speed-up on larger data sets, as the parallel scan of the data to construct histograms takes a larger fraction of the overall running time.

Most of the previous work on parallelizing boosting focuses on parallel construction of the weak learners [23] or on the original AdaBoost algorithm [19, 29] instead of gradient boosting. MultiBoost [30] combines wagging with AdaBoost, which can be performed in parallel, but inherits AdaBoost's sensitivity to noise.

Finally, multiple approaches have applied bagging [3] to web-search ranking. Recent work by Pavlov and Brunk [24] uses bagged boosted regression trees. Bagging is inherently parallel but requires additional computation time as it is averaged over many independent runs (Pavlov and Brunk used a total of $M = 300{,}000$ trees of depth $d = 12$ in the Yahoo Learning to Rank Challenge). Usually, the choice between bagging and boosting is based on desired learning paradigm rather than computational resources.

7. CONCLUSIONS

We have presented a parallel algorithm for training gradient boosted regression trees. To our knowledge, this is the first work that explicitly parallelizes the construction of regression trees for the purpose of gradient boosting. Our approach utilizes the facts that gradient boosting is known to be robust to the classification accuracy of the weak learners and that regression trees are of strictly limited depth. We have shown that our approach provides impressive (almost linear) speedups on several large-scale web-search data sets without any significant sacrifice in accuracy.

Our method applies to multicore shared-memory systems as well as to distributed setups in clusters and clouds (e.g. Amazon EC2). The distributed setup makes our method particularly attractive to real-world settings with very large data sets. Since each processor only needs enough physical memory for its partition, and the communication is strictly bounded, this allows the training of machine-learned rankers on web-scale data sets even with standard off-the-shelf computer hardware.

We are planning to extend this work in several directions. First, we think we can further increase the efficiency and performance by eliminating the master and merging histograms pairwise among workers. In addition to freeing the master processor for useful work, this approach would further overlap computation and communication. Second, we are planning to run experiments with more workers on clouds to gauge the efficacy of this approach on non-dedicated machines. Third, we intend to investigate a more aggressive speed vs. accuracy tradeoff in the computation of the splits based on stochastic approximations of the histograms.

Given the current trend towards multicore processors, parallel computing and larger data sets, we expect our algorithm to increase in both relevance and utility in the foreseeable future.

8. ACKNOWLEDGEMENTS

We would like to thank Ananth Mohan for sharing his exact implementation of Gradient Boosted Regression Trees and Yahoo Labs for providing resources for this research.

9. ADDITIONAL AUTHORS

Additional author: Jennifer Paykin (Wesleyan University, `jpaykin@wesleyan.edu`).

10. REFERENCES

[1] N. Amado, J. Gama, and F. Silva. Parallel implementation of decision tree learning algorithms. *Progress in Artificial Intelligence*, pages 34–52, 2001.

[2] Y. Ben-Haim and E. Yom-Tov. A streaming parallel decision tree algorithm. *The Journal of Machine Learning Research*, 11:849–872, 2010.

[3] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[4] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[5] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. Chapman & Hall/CRC, 1984.

[6] C. Burges. From RankNet to LambdaRank to LambdaMART: An Overview. 2010.

[7] C. Burges, T. Shaked, E. Renshaw, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *Internation Conference on Machine Learning*, pages 89–96, 2005.

[8] Z. Cao and T.-Y. Liu. Learning to rank: From pairwise approach to listwise approach. In *Proceedings of the 24th International Conference on Machine Learning*, pages 129–136, 2007.

[9] O. Chapelle and Y. Chang. Yahoo! Learning to Rank Challenge overview. *Journal of Machine Learning Research, Workshop and Conference Proceedings*, 14:1–24, 2011.

[10] O. Chapelle, D. Metlzer, Y. Zhang, and P. Grinspan. Expected reciprocal rank for graded relevance. In *Proceeding of the 18th ACM Conference on Information and Knowledge Management*, pages 621–630. ACM, 2009.

[11] O. Chapelle and M. Wu. Gradient descent optimization of smoothed information retrieval metrics. *Information Retrieval Journal*, Special Issue on Learning to Rank for Information Retrieval, 2010. to appear.

[12] J. Darlington, Y. Guo, J. Sutiwaraphun, and H. To. Parallel induction algorithms for data mining. *Advances in Intelligent Data Analysis Reasoning about Data*, pages 437–445, 1997.

[13] A. Freitas and S. Lavington. *Mining very large databases with parallel processing*. Springer, 1998.

[14] J. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2001.

[15] J. Gehrke, R. Ramakrishnan, and V. Ganti. RainForestâĂŤa framework for fast decision tree construction of large datasets. *Data Mining and Knowledge Discovery*, 4(2):127–162, 2000.

[16] R. Herbrich, T. Graepel, and K. Obermayer. *Large margin rank boundaries for ordinal regression*, pages 115–132. MIT Press, Cambridge, MA, 2000.

[17] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[18] T. Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, 2002.

[19] A. Lazarevic and Z. Obradovic. Boosting algorithms for parallel and distributed learning. *Distributed and Parallel Databases*, 11(2):203–229, 2002.

[20] P. Li, C. J. C. Burges, and Q. Wu. Mcrank: Learning to rank using multiple classification and gradient boosting. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *NIPS*. MIT Press, 2007.

[21] T. Liu, J. Xu, T. Qin, W. Xiong, and H. Li. Letor: Benchmark dataset for research on learning to rank for information retrieval. In *Proceedings of SIGIR 2007 Workshop on Learning to Rank for Information Retrieval*, pages 3–10, 2007.

[22] A. Mohan, Z. Chen, and K. Q. Weinberger. Web-search ranking with initialized gradient boosted regression trees. *Journal of Machine Learning Research, Workshop and Conference Proceedings*, 14:77–89, 2011.

[23] B. Panda, J. Herbach, S. Basu, and R. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. *Proceedings of the Very Large Database Endowment*, 2(2):1426–1437, 2009.

[24] D. Pavlov and C. Brunk. Bagboo: Bagging the gradient boosting. Talk at Workshop on Websearch Ranking at the 27th International Conference on Machine Learning, 2010.

[25] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proceedings of the International Conference on Very Large Data Bases*, pages 544–555, 1996.

[26] M. Snir. *MPI–the Complete Reference: The MPI core*. The MIT Press, 1998.

[27] A. Srivastava, E. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. *High Performance Data Mining*, pages 237–261, 2002.

[28] M. Taylor, J. Guiver, S. Robertson, and T. Minka. SoftRank: optimizing non-smooth rank metrics. In *Proc. 1st ACM Int'l Conf. on Web Search and Data Mining*, pages 77–86, 2008.

[29] N. Uyen and T. Chung. A new framework for distributed boosting algorithm. *Future Generation Communication and Networking*, 1:420–423, 2007.

[30] G. Webb. Multiboosting: A technique for combining boosting and wagging. *Machine learning*, 40(2):159–196, 2000.

[31] J. Ye, J. Chow, J. Chen, and Z. Zheng. Stochastic gradient boosted distributed decision trees. In *CIKM '09: Proceeding of the 18th ACM Conference on Information and Knowledge Management*, pages 2061–2064. ACM, 2009.

[32] C. Yu and D. Skillicorn. Parallelizing boosting and bagging. 2001.

[33] Y. Yue, T. Finley, F. Radlinski, and T. Joachims. A support vector method for optimizing average precision. In *Proc. 30th Int'l ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 271–278, 2007.

[34] Z. Zheng, H. Zha, T. Zhang, O. Chapelle, K. Chen, and G. Sun. A general boosting method and its application to learning ranking functions for web search. In *NIPS*, 2007.