HarpGBDT: Optimizing Gradient Boosting Decision Tree for Parallel Efficiency

Bo Peng[1] Langshi Chen[1] Jiayu Li[1] Miao Jiang[1] Selahattin Akkas[1]
Egor Smirnov[2] Ruslan Israfilov[2] Sergey Khekhnev[2] Andrey Nikolaev[2] Judy Qiu[1]
[1]Indiana University
[2]Intel Corporation
{pengb, lc37, jl145, miajiang, sakkas, xqiu}@indiana.edu
{egor.smirnov, ruslan.israfilov, sergey.khekhnev, andrey.nikolaev}@intel.com

***Abstract*—Gradient Boosting Decision Tree (GBDT) is a widely used machine learning algorithm, whose training involves both irregular computation and random memory access and is challenging for system optimizations. In this paper, we conduct a comprehensive performance analysis of two state-of-the-art systems, XGBoost and LightGBM. They represent two typical parallel implementations for GBDT; one is data parallel and the other one is parallel over features. Substantial thread synchronization overhead, as well as the inefficiency o f random memory access, is identified. W e p ropose H arpGBDT, a new GBDT system designed from the perspective of parallel efficiency optimization. Firstly, we adopt a new tree growth method that selects the top K candidates of tree nodes to enable the use of more levels of parallelism without sacrificing t he algorithm's accuracy. Secondly, we organize the training data and model data in blocks and propose a block-wise approach as a general model that enables the exploration of various parallelism options. Thirdly, we propose a mixed mode to utilize the advantages of a different mode of parallelism in different phases of training. By changing the configuration o f t he b lock s ize a nd p arallel mode, HarpGBDT is able to attain better parallel efficiency. By extensive experiments on four datasets with different statistical characteristics on the Intel(R) Xeon(R) E5-2699 server, HarpGBDT on average performs 8x faster than XGBoost and 2.6x faster than LightGBM.**

***Index Terms*—Machine learning algorithms, Parallel algorithms, Performance evaluation, Multithreading**

I. Introduction

Gradient Boosting Decision Tree (GBDT) [17] is a widely applied machine learning algorithm. It is not only one of the most popular algorithms in Kaggle competitions [5] but also an important method to solve industry production level problems such as click-through rate(CTR) prediction that deals with billions of advertisement impressions [19]. In recent years, it has been successfully applied to many different domains, such as Higgs boson classification [13], c redit s coring [33], computer-aided diagnosis [25], insurance loss cost modeling [18], and freeway travel time prediction [36].

Training a GBDT model is a compute-intensive problem. For example, on a moderate HIGGS dataset with 10M data instances and 28 features, the state-of-the-art system XGBoost requires 600 seconds on a 36 core Xeon server to train a model that consists of 1000 trees of depth 8. An accurate model needs several times of the number of iterations. Typical GBDT trainers provide tens of algorithmic parameters. Hundreds to thousands of runs of training are often required to find a combination of the parameters resulted in the accurate model for a given dataset. Thus, acceleration of the training of the GBDT is an important problem in research and practice.

GBDT is a boosting method that builds consecutive decision trees in a strictly sequential fashion. Since the decision tree provides the capability of modeling nonlinear functions, it is used as a building block to create complex models, for example, in Random Forest [10] and GBDT itself. Decision tree based deep models also demonstrate a comparable performance over deep learning models in [37]. Parallel decision tree construction is the key for a high-performance GBDT implementation and has been extensively studied in the data mining community [28] [30] [21] [26]. Recent work on GBDT show a trend for the convergence of HPC and big data communities, and state-of-the-art GBDT systems typically adopt HPC techniques to achieve good performance. [31] [12] [22] represent the progress of parallel GBDT on traditional CPU architectures. However, most of the papers lack parallel performance studies and do not address emerging many-core architectures. Our analysis of the existing state-of-the-art systems shows they are not efficiently parallelized, with a low utilization rate for the CPU cores. [35] [32] [15] represent another direction of utilizing GPU accelerators. Both of these approaches have advantages and limitations. As we will explain in details later in Section III-B2, random memory access to both the training data and model data are inevitable in GBDT. Current GPU-based implementations have a limitation on the allowed dataset and problem size due to the relatively small size of GPU memory when compared with the CPU-based system.

In this paper, we focus on optimizations on the CPU architecture. Based on our experience of parallelizing machine learning algorithms for the Harp framework [3] [34] [27] [11], we investigate a parallel GBDT system that is efficient when increasing the number of processors, the problem size, and the model size. In the rest of the paper, we use HarpGBDT to refer to our system. Our main contributions are summarized as follows:

- Review state-of-the-art CPU-based GBDT training sys-

tems and identify the existing performance issues, including thread synchronization overhead and memory access inefficiency.
- Propose a new tree growth method selecting top K tree nodes to split, which enables more concurrency at no cost in model accuracy.
- Propose a new block-wise parallelism method, which enables us to explore different levels of parallelism and to control the workset size.
- Propose a mixed mode, which utilizes the advantages of different modes of parallelism in different phases of the training.
- Implement HarpGBDT based on TopK growth method and block-wise parallelism and achieve 2.6x to 8.5x speedup on the Intel(R) Xeon(R) E5-2699 server.

The source code of HarpGBDT is available at https://github.com/DSC-SPIDAL/harpgbdt.

II. PRELIMINARIES

A. Gradient Boosting Decision Tree Algorithm

Given a set of feature vectors x_i of dimension M labeled as y_i, $i = 1, ..., N$, our problem is to find function approximation $\widehat{y_i} = \phi(x_i)$ that minimizes the regularized objective function $\mathcal{L}(\phi) = \sum_{i=1}^{N} \ell(\widehat{y_i}, y_i) + \Omega(\phi)$. GBDT adopts a boosting approach using tree learner, $\widehat{y_i} = \phi(x_i) = \sum f_t(x_i)$, where $f_t(x)$ is weight w of the leaf node to which x belongs in the t_{th} decision tree. For example, AIRLINE [1] dataset in Table III consists of flight arrival and departure details for all commercial flights within the USA from October 1987 to April 2008. A simple GBDT task is to predict delay for a flight, as illustrated in Fig. 1.

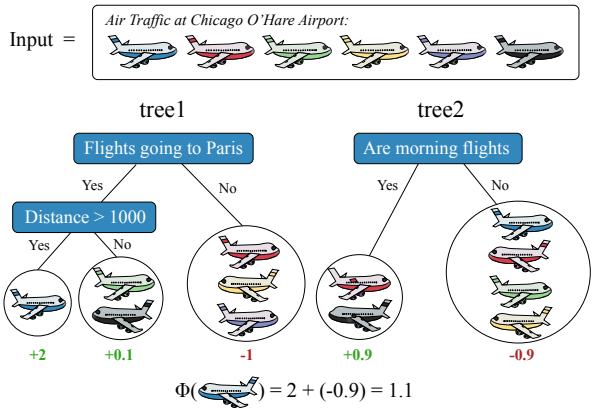


Fig. 1: An example of a tree ensemble of two trees

A second order approximation [16] is applied to a regularized objective function [12],

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} [\ell(\widehat{y}_i^{(t-1)}, y_i) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \tag{1}$$

where g_i, h_i are the first and second order gradients on the loss function ℓ, and the regularizer $\Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2$. Here T is the number of leaves, while γ and λ are regularization hyper-parameters.

The optimal weight w_j^* and objective value for each leaf j can be obtained as

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}, \quad \widetilde{\mathcal{L}}^{(t)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T \tag{2}$$

where I_j is defined as the set of instances for leaf j. Then a score function can be derived from $\widetilde{\mathcal{L}}^{(t)}$ to guide node splitting into two subset $\langle L, R \rangle$.

$$S(L, R) = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma \tag{3}$$

where $G_j = \sum_{i \in I_j} g_i$, $H_j = \sum_{i \in I_j} h_i$

Algorithm 1: BuildDecisionTree()

input : dataset $\mathbb{D} = (x_i, y_i)_{i=1}^N$, $Gradient = (g_i, h_i)_{i=1}^N$
output: tree $f(x)$

```
begin
    q: priority queue
    root: root node of tree
    // collect statistical summary GHSum for a node
    BuildHist(root)
    // find the best split point for a node based on GHSum
    FindSplit(root)
    q.push(root)
    while q is not empty do
        // pop nodes according to the tree growth policy
        nodes = q.pop()
        // update the tree by splitting each node to the left and right children according to the split point found by FindSplit
        children = ApplySplit(nodes)
        for node in children do
            BuildHist(node)
            FindSplit(node)
            q.push(node)
```

Algorithm 2: BuildHist()

input : dataset $\mathbb{D} = (x_i, y_i)_{i=1}^N$, $Gradient = (g_i, h_i)_{i=1}^N$, M:# features, B:# bins, node
output: histogram of gradients $GHSum \in \mathbb{R}^{B \times M}$

```
begin
    for x_i ∈ node do
        for m =1 to M do
            GHSum[k, m].g+ = g_i, where x_im ∈ k_th bin
            GHSum[k, m].h+ = h_i, where x_im ∈ k_th bin
```

From a single root node, a decision tree is built by recursively splitting the leaf nodes of the tree. Algorithm 1 shows a general tree building pseudo code, composing with three core functions, *BuildHist*, *FindSplit*, and *ApplySplit*. Taking AIRLINE in Fig. 1 as an example of input dataset, Fig. 2 illustrates the process of tree building. In the preprocessing step, the raw dataset is transformed into an internal representation of feature

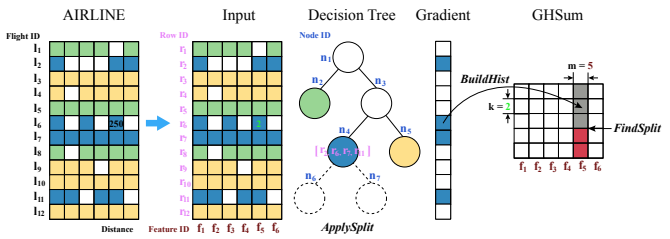


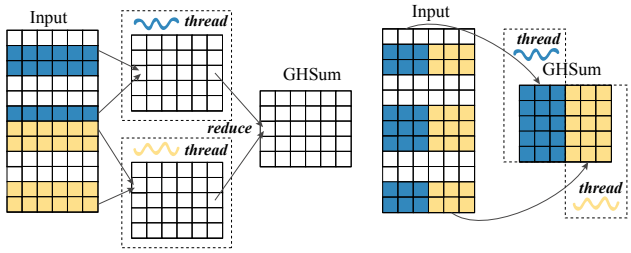
Fig. 2: Illustration of the process of tree building. Input is the input feature vectors with feature values mapped to "bin" id. Gradient contains the first order and second order gradients. GHSum maintains the statistical summary of gradients.

vectors, called "Input". First, feature values are converted into real numbers by encoding. Then a technique named *Histogram* is generally utilized to map these numbers into groups, called *bin*s. Histogram reduces the number of split point candidates later in the computation in BuildHist. In Fig. 2, we assume that the 5th column in Input is the feature 'Distance' in AIRLINE, its raw value in the 6th row is 250, and Histogram groups the values in this column into five subgroups by four split points $[200, 500, 1000, 1500]$. The feature value 250 falls into the 2nd subgroup and thus will be mapped to a binid equals 2. Tree building assigns each input data instance to a leaf node, and the different colors denote membership except that the white cells in Input represent the missing values. Taking the blue leaf node as an example, BuildHist first collects statistical summary of the gradients for this node according to its feature value distribution, as shown in Algorithm 2. Then FindSplit enumerates all possible split points, as pairs of (featureid, binid), in GHSum by calculating the split loss change according to Eq.3, and picks up the one with the maximum score. The split condition of '$Distance > 1000$' in Fig. 1 equals to the condition of '$binid > 3$' for the 5th column, which is the arrow of the FindSplit function indicates in Fig. 2. Finally, ApplySplit expands the tree by adding two child leaves and updates the membership for all input instances in this node correspondingly.

There are two popular tree growth methods: a *depthwise* method splits leaves level by level, and a *leafwise* method selects the leaf node with the largest value of loss change to split. In Algorithm 1, these two growth methods are unified by a priority queue via dedicated comparison functions. The number of nodes to pop out is the maximum number of leaves in depthwise and is 1 in leafwise.

D is used to represent the ***tree size***, which contains $2^D - 1$ nodes. In depthwise, D equals to the tree depth. In leafwise the tree is usually unbalanced, and the tree depth is much larger than D. Given a dataset $\mathbb{D} \in \mathbb{R}^{N \times M}$, the time complexity of BuildHist is $\mathcal{O}(NMD)$ in depthwise, in which it goes through all data instances once at each level of the tree. FindSplit is $\mathcal{O}(MB)$ for each node, and therefore it is exponential to the tree size D, with behavior $\mathcal{O}(MB2^D)$. ApplySplit contains simple operations and is relatively trivial, as $\mathcal{O}(2^D)$. BuildHist in leafwise is irregular because the number of instances in each node dynamically changes and cannot be analytically predicted, while FindSplit and ApplySplit keep the same complexity as in the depthwise method.

B. Parallel GBDT Training



(a) Data Parallelism (b) Feature Parallelism

Fig. 3: Parallelism patterns to parallelize BuildHist.

Data Parallelism and ***Model Parallelism*** are two major approaches to parallelizing a serial machine learning algorithm. They describe how data are partitioned among parallel workers and how these workers are synchronized as the algorithm progresses. In GBDT, 'Data' refers to the input data, and 'Model' refers to the intermediate data created within the algorithm as it forms the decision tree. Fig. 3(a) illustrates the data parallel approach in parallelizing BuildHist. It partitions input by row and replicates model to all spawned threads. Each thread works on one row partition and its local model. In the end, thread local model replicas are reduced to a global one. Feature Parallelism in Fig. 3(b) is a type of model parallelism, in which both input and model are partitioned by columns. Each thread works on one partition of feature columns and updates the global model without conflicts.

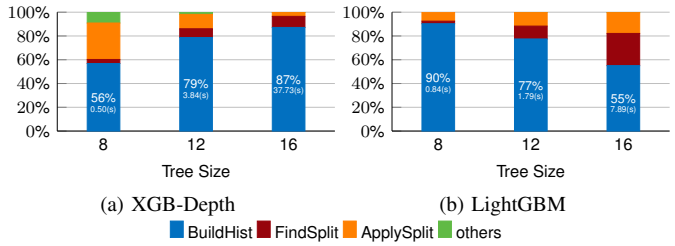
XGBoost [12] and LightGBM [22] are two state-of-the-art GBDT systems. In its original paper, XGBoost proposes a feature parallel approach, in which the histogram statistics are collected for each feature column in parallel. With the success of the XGBoost open source project, its code evolved quickly, adding new tree-building modules in an ongoing fashion. One latest module, tree_method=hist, changes to data parallelism, and achieves better performance. We use the term XGBoost to refer to this specific data parallel version. LightGBM adopts a standard feature-wise model parallelism approach.

III. Analysis of Existing GBDT Systems

To investigate the efficiency of the parallel design of GBDT trainers, we do a hotspot analysis of two representative systems: XGBoost and LightGBM. They are implemented in C++ and support multithreading using OpenMP. We run this experiment on a machine with 36 physical cores and fix the number of threads to use at 32. The detailed hardware and software configurations and datasets used are described in Section V-A.

A. Hotspot Analysis

We evaluate these two systems with the execution time breakdown on HIGGS dataset. BuildHist is identified as the hotspot among the three core functions for all values of the tree size. In Fig. 4 and Table I, suffixes of "-Depth" and "-Leaf" refers to depthwise and leafwise tree growth method used



(a) XGB-Depth (b) LightGBM

Fig. 4: Execution time breakdown for one iteration on HIGGS dataset.

in the trainer. As LightGBM only supports leafwise method, no suffix is used for it. We only show the results of XGB-Depth and LightGBM, which are the representatives of the depthwise and leafwise methods. The result from XGB-Leaf is similar and hence omitted for brevity. According to the time complexity analysis in Section II, the time complexity of BuildHist should be $\mathcal{O}(D)$ in depthwise method. However, we observe exponential growth $\mathcal{O}(2^D)$ for BuildHist in XGB-Depth, which indicates the existence of large parallel overhead.

TABLE I: Profiling of XGBoost and LightGBM

Trainer	XGB-Depth	XGB-Leaf	LightGBM
CPU Utilization(cores)	13.9	13.9	19.2
Barrier Overhead	**42%**	**42%**	**23%**
Latency(cycles)	35	37	25
Memory Bound	**51.0%**	**52.9%**	**54%**

Table I summarizes the profiling results of hardware event counters via Intel(R) VTune(TM) Amplifier on HIGGS with tree size 8. Low CPU Utilization indicates a poor parallel efficiency. VTune reports high OpenMP Barrier Overhead on both trainers. LightGBM spends 23% of the effective CPU time in spinning. XGBoost spends even more up to 42%. Both of them also show a high Memory Bound above 50%, which means over 50% of CPU cycles are waiting for load or store instructions.

B. Low Parallel Efficiency Problem

1) Thread Synchronization Overhead: OpenMP provides an easy to use programming model by adding #pragma before the for-loops to parallelize the code. However, this introduces a barrier wait at the end of the loop, which might not be necessary from the original algorithm's perspective of view. For leafwise algorithms, XGB-Leaf and LightGBM have to select the leaf with the largest loss change score in each split. Therefore, they are constrained to execute leaf by leaf. For depthwise method, the leaves at the same level of the tree are independent and can be constructed in parallel. However, as a data parallelism approach, XGB-Depth maintains a model replica for each thread. To avoid uncontrolled memory footprint of the model replicas, it also implements tree building leaf by leaf. The amount of thread synchronization overhead is proportional to the numbers of leaves $\mathcal{O}(2^D)$ in both of the two systems. Thread synchronization could introduce significant overhead, especially when load imbalance is common for datasets with sparse features and missing values. This explains the observed high OpenMP barrier overhead and exponential growth of execution time of BuildHist in Fig. 4.

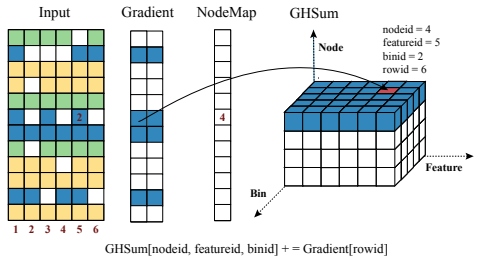


Fig. 5: Computation and memory operations in BuildHist.

2) Memory Bound: The hotspot function BuildHist involves four major data structures, including Input, Gradient, NodeMap, and GHSum, see Fig. 5. NodeMap maintains the membership of the data instances to the tree node, and it dynamically changes when the tree splits and grows. A typical procedure starts with selecting a tree node to grow. Given the nodeid, NodeMap is retrieved to get the rowid set for the corresponding data instances. Then, the binid for each feature is fetched from Input, and gradient is fetched from Gradient by the given rowid. Finally, GHSum is updated at the position with the index of $\langle nodeid, featureid, binid \rangle$. Memory size of Input is $\Theta(NM)$, NodeMap and Gradient are $\Theta(N)$, and GHSum is $\Theta(MB2^D)$.

First, the computation versus memory access ratio in GBDT is low. Excluding the accesses to Input and Gradient which can be amortized in best cases, one read operation and one write operation to GHSum, 16 Bytes in Double, involves only one floating-point computation. The computation versus memory access ratio here is $\frac{1}{16} = 0.0625$.

Secondly, random memory access is inevitable in BuildHist. Because of the dynamic nature of the tree splitting, NodeMap keeps changing along with the tree growth. For leafwise methods, random memory access to Input and Gradient is typically needed. And as at least one of the three indexes of request to GHSum would change dynamically, leading to random memory access to GHSum. No static memory layout for GHSum exists that can support sequential accesses all the time.

IV. HarpGBDT: Design and Implementation

To improve the efficiency of parallelization, we design HarpGBDT with optimizations on the hotspot BuildHist to reduce synchronization overhead, increase concurrency, reduce random memory access, and improve cache efficiency.

A. Block-wise Parallelism

Initially we investigate concurrency in the parallel tree construction. In data parallelism, a set of rows (or row blocks) are the basic unit that can be scheduled as a task in BuildHist. In model parallelism, each cell in the 3-D matrix GHSum, as in Fig. 5, can be the basic unit and no conflict of model updates will exist among the threads in this way. A more general parallel solution can be a mixture of data parallelism and model parallelism.

We propose to use a ***Block*** as the basic unit for data organization and parallelism. By viewing both GHSum and Input as three dimensional data structures, $\langle node, bin, feature\rangle$ for GHSum and $\langle row, bin, feature\rangle$ for Input, a Block is defined as a cube in GHSum and associated cube in Input. Each cube is implemented as a 3-dimensional array in a row-major layout. By configuring the Block parameter $\langle row\_blk\_size$, $node\_blk\_size$, bin_blk_size, $feature\_blk\_size\rangle$, we set the size of each dimension of the cubes. In this way, we can have many different designs to build a decision tree in parallel.

First, standard data parallelism equals to $\langle X, X, 0, 0\rangle$, where X indicates a variable instead of a fixed number. In XGB-Hist, the latest data parallelism version of XGBoost, row_block_size is not a fixed parameter. The row set for each tree node is dynamically partitioned. $node\_blk\_size$ is set to 1 in order to constrain the memory footprint of the model replicas.

Secondly, traditional feature-wise model parallelism equals to $\langle X, X, 0, 1\rangle$ in block-wise parallelism, where size 0 refers to all. In the original version of XGBoost [12], row blocks were proposed to mitigate the long-distance random memory access to Gradient in a feature-wise method. It equals to set row_blk_size to "X" here. $node\_blk\_size$ equals to 0 in XGB-Approx. It scans each column of Input sequentially at the cost of writing to a relatively large region of model memory, which is a vertical plane crossing all tree nodes in GHSum. LightGBM adopts standard feature parallelism with row_blk_size equals to 0 and $node\_blk\_size$ equals to 1.

Beyond standard feature-wise parallelism, model parallelism in GDBT has many other options which are not fully explored. 1) ***Block-wise feature level parallelism*** Set $feature\_blk\_size$ enables a trade-off of preferences between read operations and write operations. Small value, as in traditional feature-wise parallelism, is good for write operations but brings extra reads to Gradient. Large value, as in data parallelism, is good for read operations but may incur large cache miss when writing to the large size of memory region randomly. 2) ***Bin level parallelism*** When organizing Input with "bin" dimension partitions, it enables model parallelism even when the number of features is small. However, each data cube of Input becomes sparse. The additional cost to sparse data structure makes it less efficient. 3) ***Node level parallelism*** In node level parallelism, the computations in each tree node, including BuildHist and FindSplit, can be assigned to one thread, thus has the advantages of fewer thread synchronizations. Setting $node\_blk\_size$ enables a trade-off between a fewer number of thread synchronizations and a larger size of the memory region for write operations. Furthermore, from the algorithm's perspective of view, many of those thread synchronizations are not required at all in depthwise method. The candidates selected for splitting can do their work independently, and synchronization is only necessary when updating the tree structure and the priority queue. However, node level parallelism can only be applied if there are enough tree leaf nodes to split. It can not be applied in the beginning phase of tree building, and not in leafwise growth method, which strictly processes one node after another.

B. TopK Tree Growth Method

In order to utilize node-level parallelism in leafwise growth method, we propose a new tree growth method which extends the existing one by selecting top K, rather than top 1, candidates with the largest loss change values from the priority queue. In this way, a 'K' fold node-level parallelism is enabled, as in Fig. 6(d). TopK is a mixture of depthwise and leafwise growth methods. On the one hand, top K candidates splitting at the same time will build a different tree that achieves less accuracy on the training data when compared with the top 1 approach. On the other hand, it may build a more robust decision tree because it mitigates the tendency of continuously splitting inside one node to form a very deep tree in some particular cases. By intuition, the new algorithm can achieve a similar performance of accuracy when K is not too large. TopK method also supports the depthwise mode. As in Fig. 6(b), only a subset of K leaves are selected each time, rather than all leaves selected at the same time in depthwise.

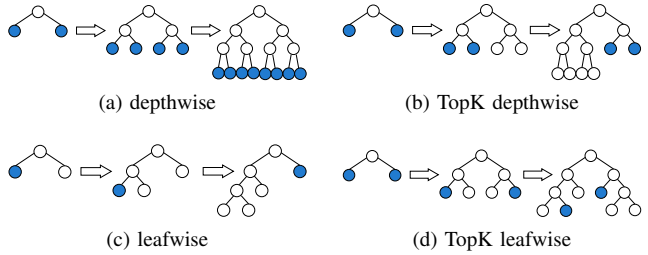


Fig. 6: Illustration of tree growth method. (a)(c) are examples of standard depthwise and leafwise. (b)(d) are examples of TopK methods(K=2). Blue nodes refer the candidates selected to split.

C. Mixed Mode of Parallelism

Data parallelism and model parallelism have their advantages and disadvantages and fit best to the different problem settings. We propose mixed modes of parallelism by applying different approaches in different phases of the tree building process. In this way, we combine the advantages of them.

One scenario to apply a mixed mode is for node-level parallelism. The beginning phase of tree building starts with a new parallelism mode. When the tree grows to a number of leaves larger than the threads number T, it switches to node-level parallelism. At the end phase, it switches back as the number of leaves to split drops below the threshold T. Another scenario is for the case of a dataset with a small number of features, where data parallelism is better choice in the beginning to fully utilize the available CPU cores. When the tree grows, an increasing number of leaves makes it an option to switch to model parallelism which schedules the tasks on both the feature and node level.

With TopK, thread synchronization is needed after processing K leaves so as to select the next global top K candidates. We refer to this strict TopK method as "SYNC". If we remove the constraint of the global top K and allow K threads to select the top candidate as best as they can, synchronization is no

TABLE II: Four Modes of Parallel Designs for GBDT.

Mode	*Description*
DP	data parallelism
MP	model parallelism
SYNC	mixed mode (DP, MP, DP)
ASYNC	mixed mode (DP/MP, node parallelism, DP/MP)

longer required. This loosely coupled TopK method is denoted by "ASYNC".

Table II categorizes four modes supported by HarpGBDT. Together with block-wise parallelism and TopK growth method, it helps HarpGBDT to fit the scenarios with respect to different input shape and size for optimal performance.

D. Reducing Thread Synchronization Overhead

Reducing the number of times entering the parallel regions in OpenMP is one straight-forward solution to reduce thread synchronization overhead. By setting the $node\_blk\_size$ to H, H selected candidates will be processed in one parallel region. To build a tree with L nodes, the number of synchronizations drops from L to $\frac{L}{H}$.

A more aggressive solution is applying ASYNC mode. ASYNC schedules all the computation involved within one tree node as a single task in the intermediate phase by applying node parallelism; in this way, it avoids all the for-loops barrier wait overhead. Note that splitting nodes in tree building is not a embarrassingly parallel process. Different threads have to synchronize when they access the shared data structures, including the priority queue and the tree. A lightweight spin mutex works well in this scenario and gives much less overhead comparing to for-loops barrier wait.

E. Optimization for Memory Access

Input In Input, the original feature values are replaced by its bin id counterpart in a prepossessing step. This will reduce the memory footprint to $\frac{1}{4}$ as bin id need only 1 Byte when max bin size is 256, which is sufficient in general.

Gradient Tree building will scan the row id set for each candidate leaf node, fetching input data from Input and Gradient by the row id. Non-contiguous row ids lead to random memory access. Moreover, Gradient may need to read for multiple times. E.g., in feature parallelism, threads working on different feature columns all need to read the same row of Gradient. To reduce the multiple random access, we extend row id in NodeMap with corresponding gradients, denoted as MemBuf, as in Fig. 7. Because the gradients are always accessed along with the corresponding row ids for a node, MemBuf can increase the cache efficiency.

GHSum Each element of Input incurs one read and one write operation on GHSum, as in Fig. 5. Consecutive access on GHSum should be confined to a small region to avoid frequent cache miss. For simplicity, assume a dataset with M features and all features have the same number of bins, 256 by default. Each element in GHSum is two doubles for summations of gradients. One feature occupies the memory of size $256 \times 16 = 4K$ Bytes.

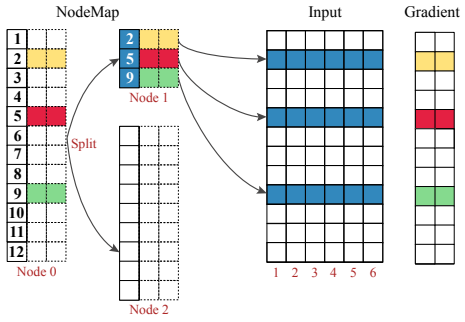


Fig. 7: MemBuf: Extending NodeMap with a replica of Gradient by keeping (rowid, gradients) pairs.

Data parallelism processes the whole row of input consecutively, in which a region of $4K \times M$ is involved. It will incur a large number of cache miss when M goes large. Feature parallelism has advantages in accessing GHSum. In depthwise method, the bin ids can be resorted in order by prepossessing, then the region of consecutive access is confined to size L, where L is the number of leaf nodes. In leafwise method, consecutive accesses happen on the same node and the same feature in GHSum, and the region size becomes $4K$. In our block-wise method, this region size can be configured, which is $16 \times bin\_blk\_size \times feature\_blk\_size \times node\_blk\_size$. By setting the size parameters corresponding to the shape of the input dataset, we can tune the performance and keep the balance between read and write operation efficiency.

V. Experiments

A. Experimental Setup

1) Hardware and Software Configure: All experiments are conducted on the server with 2x18-core Intel(R) Xeon(R) E5-2699 v3 processors and 128 GB memory. On this NUMA machine with 36 physical cores, we fixed the thread number to 32, with the default interleave memory policy. OS is Red Hat Enterprise Linux 7. All GBDT trainers are compiled with gcc 4.9.2 and -O3 compilation optimization. Intel(R) VTune(TM) Amplifier 2018 and Advisor(TM) 2019 are used as the performance profiling tool.

2) GBDT Implementations: XGBoost [8], LightGBM [6] are two systems for comparison. HarpGBDT is based on the XGBoost code base, reusing the code of histogram initialization algorithm and focusing on the optimizations to improve parallel efficiency.

TABLE III: Dataset. N is #instances. M is #features. S refers to sparseness and CV is dispersion of # bins distribution.

Dataset	N	M	S	CV	$Size$	$TestN$
HIGGS [4]	10M	28	0.92	0.40	5.3G	100K
AIRLINE [1]	100M	8	1	0.89	5.4G	1M
CRITEO [2]	50M	65	0.96	0.58	45G	1M
YFCC [9]	1M	4096	0.31	0.06	19G	100K
SYNSET	10M	128	1	0	18G	N/A

3) Datasets: Four open datasets are used in the experiments and listed in Table III, where $S = \frac{\#element}{N \times M}$ represents the sparseness, $CV = \frac{stdev}{mean}$ measures the dispersion of the number of bins distribution over all the features. The larger

this number, the more uneven of the distribution is, which indicates workload imbalance. SYNSET is a synthetic dataset with randomly generated feature values following a normal distribution. It has an even feature value distribution and always builds a balanced tree by GBDT, which represents an ideal even workload scenario.

4) Algorithm and Evaluation Parameters: We fix the training related parameters with widely used settings as: $\gamma = 1.0$, $\lambda = 1.0$, $learning\_rate = 0.1$, $min\_child\_weight = 1$, binary logistic loss. As we focus on efficiency evaluation in order to find out the pros and cons of different parallel designs and optimizations, keeping the same workload of computation in comparison is essential. Differences of algorithm optimizations, such as feature bundling, category feature encoding, and sampling, can potentially lead to large performance difference but are not evaluated in the experiments. Optimizations that achieve better speed at cost of compromised accuracy, such as using single precision in BuildHist, are also not included. For the training parameters, the settings have impacts on the accuracy, and some of them have significant impacts on the speed, such as tree size. A thorough discussion of the trade-offs caused by parameter setting is related to parameter tuning and is useful in practice, but beyond the scope of this paper.

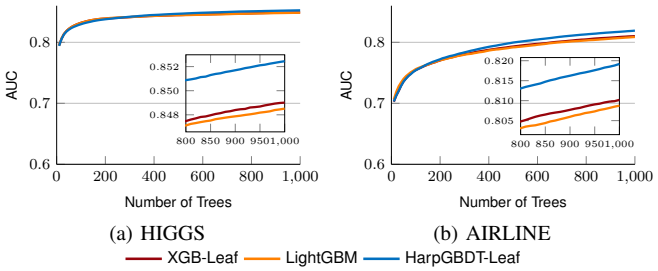
Performance evaluation focuses on training time using the wall clock time. It excludes the time spent on data loading and one-time initialization, which is a small constant value for a dataset in each implementation. In GBDT training, the workload of computation on consecutive trees gradually shrinks due to the decreasing of the gain to split the fine-tuned trees later on. Pruning algorithms adopted in the trainers can affect the execution time of tree building, especially in the later phase. In order to reduce this impact, the average training time per tree is measured for the first 100 trees.

Area Under The Curve (AUC) is used to evaluate model accuracy. When considering the accuracy, training time to achieve the same highest AUC when training with 1000 trees is used as the performance metric and Convergence Speedup is defined as the ratio of this metric on two systems.

B. Effectiveness of TopK Tree Growth Method

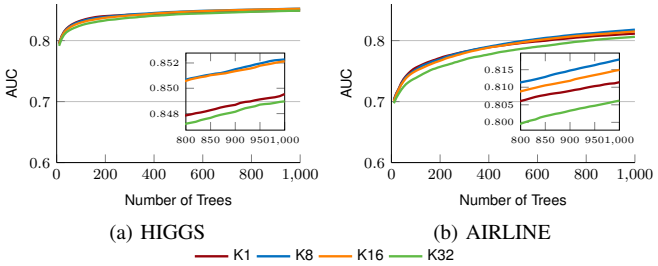
Fig. 8 shows the convergence rate of the three trainers in leafwise mode. TopK method starts from a lower accuracy but soon catches up and even gets better accuracy. In depthwise mode, TopK method and the standard method build the same tree and achieve the same convergence rate.

Fig. 9 demonstrates that accuracy is robust for a large range of K. Accuracy under $K = 16$ can catch up fast and exceed the standard method ($K = 1$). $K = 32$ shows a larger gap in the beginning and catches up slowly. We run this experiment under a worst-case condition for large K by setting the parameters with small tree size and ASYNC mode. Results on other datasets and settings, omitted here due to space limitations, show that $K = 32$ works well in other modes or on larger trees.



(a) HIGGS (b) AIRLINE

Fig. 8: Comparison of convergence rate for TopK. ($D = 8$,$K = 8$)



(a) HIGGS (b) AIRLINE

Fig. 9: Impacts of K on convergence rate for TopK. ($D = 8$,ASYNC)

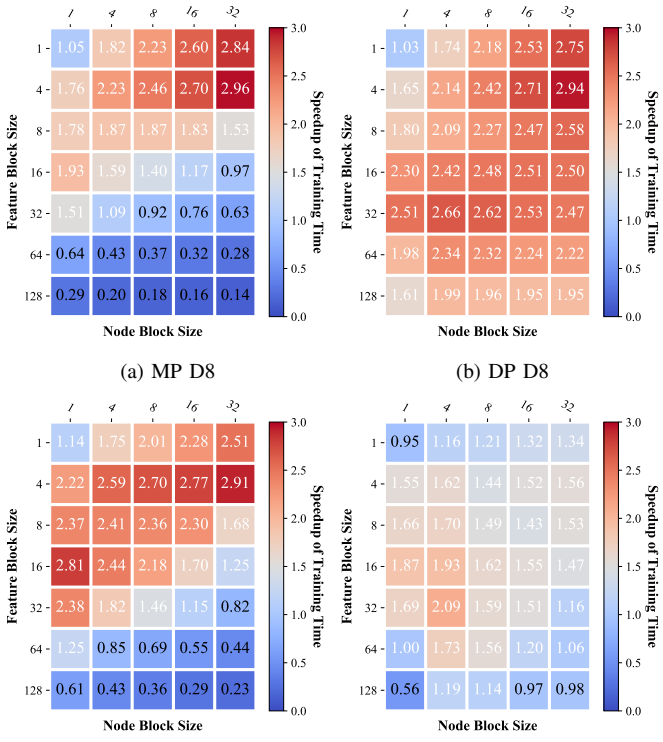
C. Block Configurations and Parameter Tuning

HarpGBDT provides a group of system parameters, as in Table IV. Proper settings of these parameters enable the system to deliver optimal performance and keep efficient with different inputs. We start parameter tuning experiments on SYNSET in order to learn the influence of configurations. We set the thread number $T = 32$ and $bin\_blk\_size = 256$, disable blocks along the bin dimension in the following experiments.

TABLE IV: System Parameters of HarpGBDT

Parameter	*Description*
K	number of candidates selected each time
$mode$	mode of parallelism(DP,MP,SYNC,ASYNC)
row_blk_size	row block size($\frac{N}{T}$,$\frac{2N}{T}$...1)
$node\_blk\_size$	node block size(1...K)
bin_blk_size	bin block size(1...256)
$feature\_blk\_size$	feature block size(1...M)

Fig. 10 shows the influence of feature and node block size on performance. In this experiment, we set $row\_blk\_size = \frac{N}{T}$ to enable data parallelism to fully utilize the CPU cores; $K = 32$. Training time is normalized over the result of standard model parallelism, which equals to $feature\_blk\_size = 1, K = 1$. We have the following observations: 1) Significant performance gain can be achieved by adjusting the block parameters, as a maximum of 2.94x to 2.96x speedup is observed for DP and MP, respectively. 2) Both Data Parallelism and Model Parallelism favor a medium size of feature block when $node\_blk\_size$ is 1. As shown in the first columns of Fig. 10, the medium size of the feature block gets the best performance. It justifies the presumption that there should be a trade-off between read and write operations in GBDT, as read operation works well on large feature blocks while write operation favors small ones. 3) A mutual restriction exists between these two parameters. Larger $node\_blk\_size$ boosts the performance only if the feature block size is small, when it

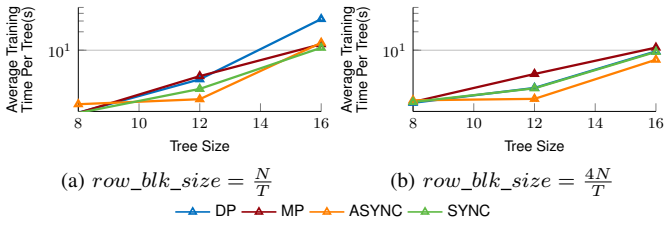


(a) MP D8 (b) DP D8

(c) MP D12 (d) DP D12

Fig. 10: Training time speedup over standard Model Parallelism. Leafwise Growth. MP refers to model parallelism, DP refers to data parallelism. Tree size denoted as D#.

provides enough blocks to the scheduler in Model Parallelism and low cache miss rate in Data Parallelism.



(a) $row\_blk\_size = \frac{N}{T}$ (b) $row\_blk\_size = \frac{4N}{T}$

Fig. 11: Performance of parallelism modes over tree size.

Fig. 11 shows the performance trend of the four parallelism modes over tree size. In this experiment, we set $\langle feature\_blk\_size, node\_blk\_size \rangle$ to $\langle 4, 32 \rangle$ at D8, $\langle 32, 4 \rangle$ at D12 and D16. We have the following observations: 1) SYNC mode is effective that it achieves similar or better performance than DP and MP in all the cases. 2) ASYNC has noticeable overhead in a small tree but achieves the best performance in large trees. It shows the best capability of scaling over the tree size. 3) D16 is a stress testing that each tree splits to 65536 leaves, i.e., each node contains about $\frac{N}{2^{16}} = 152$ data instances at the end. Fig. 11(a) shows the performance of all modes except MP degrade in D16. DP related modes are affected because of synchronization overhead, as the default setting of $row\_blk\_size = \frac{N}{T}$ makes a large number of small row set and thus too many small tasks. By adjusting row_blk_size to a larger value, as in Fig. 11(b), DP and ASYNC boost up about 50% at D16.

D. Effectiveness of Optimization

In order to evaluate the effectiveness of the proposed optimizations, we run a serial of experiments on SYNSET by adding optimizations incrementally. Starting from the baseline of standard Feature Parallelism($feature\_blk\_size = 1, K = 1$) and Data Parallelism ($feature\_blk\_size = 128, K = 1$), the first step is adjusting $feature\_blk\_size$, denoted as "+Block". We set it to 4 in Model Parallelism and 32 in Data Parallelism. Adding MemBuf is the second step. Then, we increase K to 32 and adjust $node\_blk\_size$ accordingly. Finally, we change the mode to a mixed mode, SYNC in D8, and ASYNC in D12. Table. V shows the training time speed up in each consecutive step. "+Block" for Data Parallelism incurs performance loss 13% in D8, and recovers by "+MemBuf." It demonstrates that a single optimization does not guarantee performance gain under every scenario and various optimizations work better together.

TABLE V: Performance Gain with Individual Optimizations.

Mode	Size	+Block	+MemBuf	+K32	+MixMode
MP	D8	104%	14%	60%	8%
MP	D12	146%	22%	51%	48%
DP	D8	-13%	16%	77%	4%
DP	D12	170%	2%	28%	96%

TABLE VI: Comparison of Profiling Result

Trainer	Depth		Leaf	
	HarpGBDT	XGB	HarpGBDT	LightGBM
CPU Utilization(cores)	27.5	13.9	28.5	19.2
Barrier Overhead	**9%**	42%	**8%**	23%
Latency(cycles)	15	35	16	25
Memory Bound	**38%**	51.0%	**41%**	54%

E. Parallel Efficiency

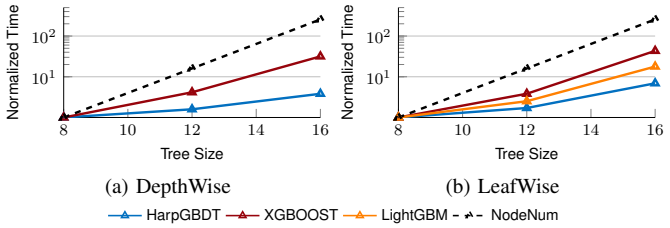
This set of experiments evaluate parallel efficiency from different angles on HIGGS. For the parameters setting, HarpGBDT adopts DP mode for D8 and ASYNC mode for larger trees, $K = 32, feature\_blk\_size = 4, node\_blk\_size = 32$.

First, profiling results on D8, as in Table VI, shows that the OpenMP barrier overhead is significantly reduced in HarpGBDT, which drops from 23% and 42% to around 9%. Data Parallelism with large K and $node\_blk\_size$ efficiently reduce the synchronization overhead. Memory-related metrics also improve due to the better block size configurations. Results on D12 shows more advantages of ASYNC, in which the barrier overhead ratio drops to 2%.

Secondly, HarpGBT scales better over the tree size, as shown in Fig. 12.

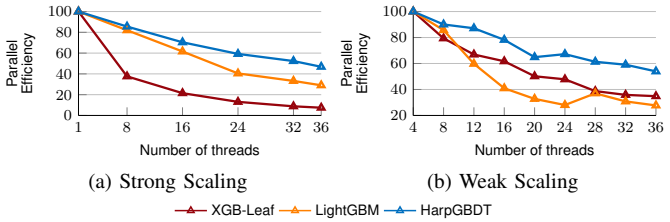
Thirdly, we run strong scaling and weak scaling tests. Given execution time T_n on n parallel workers, the parallel efficiency for strong scaling is $\frac{T_1}{n \times T_n} \times 100\%$, and is $\frac{T_1}{T_n} \times 100\%$ for weak scaling. In Fig. 13(a), HarpGBDT relatively scales better, shows that the optimizations enable it to utilize more computation resources efficiently. When applying GBDT, it is a challenging big data issue to deal with the ever-increasing

data volume. Weak scaling is more suitable in this case. We increase the input size proportional to the number of threads by duplicating the HIGGS dataset. As in Fig. 13(b), HarpGBDT shows significantly better weak scaling efficiency.



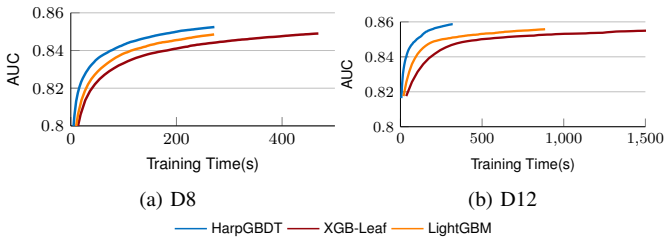
(a) DepthWise (b) LeafWise

Fig. 12: Trend of training time over the tree size.



(a) Strong Scaling (b) Weak Scaling

Fig. 13: Parallel efficiency on HIGGS D12.



(a) D8 (b) D12

Fig. 14: Trend of convergence speed.

Finally, we evaluate parallel efficiency by the convergence speed of the three systems. In Fig. 14(a), although LightGBM is about 2x slower than HarpGBT in the beginning, it finishes the 1000 trees at nearly the same time with lower accuracy. In Fig. 14(b), HarpGBT shows strong performance when increasing tree size to D12, and the job finishes with much faster convergence speed.

F. Roofline Model

Performance results demonstrated in Fig. 13 show that the scaling is not linear even after optimizations. The first reason for this is that the code is not fully parallelized. We focus on only the hotspot function BuildHist. There are also some serial parts of code, such as the tree structure update. The memory-bound nature of GBDT is another factor. We run a roofline analysis by Intel(R) Advisor 2019 on the GBDT systems. Results of BuildHist, which take the most time, are presented in Fig. 15. HarpGBDT, especially the ASYNC mode in D12 achieves much better performance in terms of bandwidth utilization than the baselines. However, our results still lie between the L3 cache and DRAM bandwidth diagonal lines, which indicate memory bandwidth limitations preventing the function from achieving better performance. On our experimental server, L3 cache bandwidth scales linearly over the number of cores, but DRAM bandwidth gets saturated soon after 16 cores.

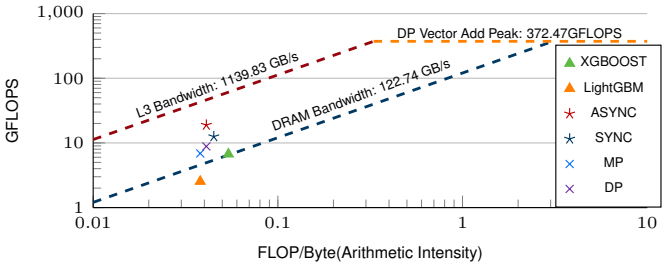
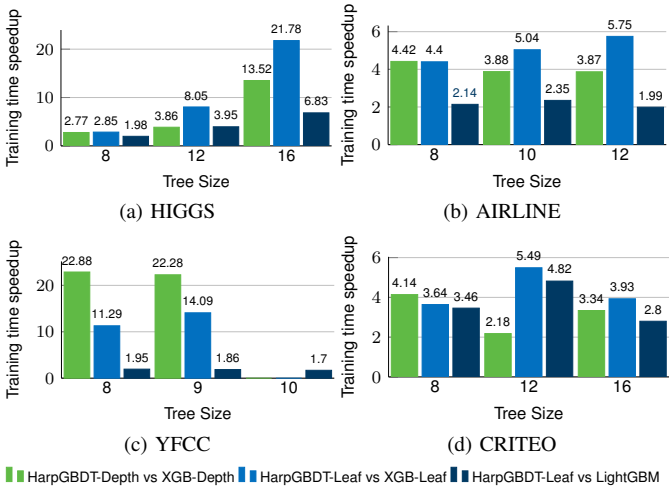


Fig. 15: Roofline Analysis on HIGGS D12.

G. Overall Performance

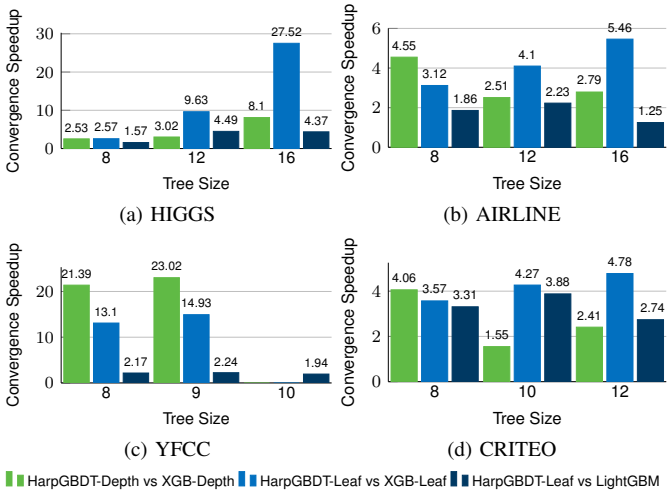
We use four datasets to evaluate the overall performance by the metrics of training time speedup and convergence speedup.



(a) HIGGS (b) AIRLINE

(c) YFCC (d) CRITEO

Fig. 16: Training time speedup on four datasets.



(a) HIGGS (b) AIRLINE

(c) YFCC (d) CRITEO

Fig. 17: Convergence speedup on four datasets.

YFCC and AIRLINE are two types of input with a very different shape. For fat matrix input YFCC, as discussed in Section V-C, standard DP does not work well due to inefficient write operations and large memory footprint. XG-

Boost fails with an error of out-of-memory at tree size 10. HarpGBDT shows more than 10x speedup over XGBoost. Standard feature-wise MP also does not work well due to inefficient read operations. HarpGBDT shows more than 1.7x training time speedup and 1.9x convergence speedup over LightGBM. For thin matrix input AIRLINE, DP is a better choice than MP in small tree D8. When changing to ASYNC on D12 and D16, it achieves more than 4x training time speedup and 3x convergence speedup over XGBoost. It is hard to learn a robust model by leafwise method on CRITEO, even after we set $min\_child\_weight = 100$ to control overfitting. The leafwise method builds deep trees with depth more than 150 in case of large tree size. One possible reason is the response variable replacement encoding, which generates features highly correlated with the response variable and make it prone to keep splitting within one branch of the tree. In this case, HarpGBT achieves an average more than 3x speedup over XGBoost and LightGBM.

Across the four datasets, HarpGBT is 8.7x faster in training time and 8.5x faster in convergence speed than XGBoost, and 3x faster in training time and 2.6x faster in convergence speed than LightGBM.

VI. Related Work

XGBoost and LightGBM are generally used as the baselines for GBDT evaluations. In addition to CPU architectures, using GPU to accelerate GBDT algorithm is also an important topic. GPU-GBDT [32] reports 1.5-2x speedup with Titan X vesus 2x10-core Xeon E5-2640. CatBoost [15] claims 2.3x speedup with Tesla P100 versus Xeon E5-2660, and [35] shows 7-8x speedup with GTX 1080 versus 2x14-core Xeon E5-2683. These works demonstrate promising performance speedup. Since GBDT training is a memory-bound problem, high bandwidth memory is critical in accelerating the computation. When all the training data and model data can fit into the device memory, a GPU-based implementation may outperform the CPU-based system due to the support of higher memory bandwidth. Compared with HarpGBDT with 2x18-core Xeon E5-2699 on HIGGS D8 and D12 datasets, the latest version of GPU-GBDT, ThuderGBM [7] shows 0.7-0.9x speedup on a single GPU of K80 and 3-4x speedup on Tesla V100. However, when the data can not fit into the device memory, the performance will degrade abruptly due to the bottleneck of the bandwidth between host memory and device memory, or can even fail with an out-of-memory error. For example, V100 has 16GB GPU memory, on which ThuderGBM fails with OOM error on HIGGS D16 of a large model with GHSum around 7GB; it also fails on YFCC and CRITEO, which have input larger than 16GB.

XGBoost and LightGBM build distributed GBDT upon a collective communication layer. [24] proposes PV-Tree, a voting approximation which avoids a large volume of data communication. DimBoost [20] deploys a large scale distributed GBDT system based on the parameter server architecture [23] and integrates with Yarn and HDFS, which is popular in a typical industry production environment. [14] proposes an asynchronous implementation on parameter server. HarpGBDT provides a high-performance kernel, and the experiences learned to improve parallel efficiency can also be helpful for the distributed system design.

Many related works focus on algorithm optimizations. pGBRT [31] first proposed to utilize *histograms* to speed up the creation of decision tree in the GBDT algorithm. LightGBM [22] and DimBoost [20] all propose feature bundling methods to deal with sparse features. GBDT-Sparse [29] proposes L1 regularization for high dimensional sparser output problem and demonstrates 40x speedup. These algorithm optimizations are orthogonal to our work on parallel efficiency and can integrate with our parallelism approaches for further performance increases.

VII. Conclusions and Future Work

This paper improved the parallel efficiency of decision tree building in the popular GBDT algorithm and implemented it with HarpGBDT as a high-performance kernel. The proposed approaches include a block-wise parallelism strategy and a TopK extension of tree growth method to fully utilize the potential parallelism in GBDT. By adjusting the block configuration, performance related to memory access can be tuned. By selecting different parallel method based on the shape of the input matrix and the phase of tree growth, thread synchronization overhead is reduced significantly. Performance evaluations on four open datasets with quite different shapes and characteristics show that HarpGBDT outperforms two state-of-the-art systems and achieves a speedup of 2.6x to 8.5x on average and up to 27x on large tree size.

Many topics not covered in this work can be explored further based on our high-performance HarpGBDT kernel. First, study the optimization approaches on CPU and GPU architectures and combining their advantages is an exciting direction. Secondly, exploring the applicability of proposed block-wise parallelism method to other machine learning algorithms is promising. Thirdly, further improving memory efficiency and multi-core scaling is still in need. One direction is the optimizations for NUMA system, and another one is the micro-optimizations on instructions level. Vector instruction sets and explicit prefetching employed in Intel(R) Data Analytics Acceleration Library demonstrate advantages over the compilers generated vectorization code. We can adopt their optimizations with our described approach to achieve better results. Finally, optimizations for other functions beyond BuildHist, such as histogram initialization, and exploring the trade-off between accuracy and efficiency can also be important in practice.

Acknowledgments

We gratefully acknowledge support from the Intel Parallel Computing Center (IPCC) Grant, NSF 1443054 CIF21 DIBBs 1443054 Grant, and Indiana University Precision Health Initiative. We also appreciate the system support offered by FutureSystems.

REFERENCES

[1] AIRLINE dataset. http://stat-computing.org/dataexpo/2009/. [Online; accessed 15-Apr-2019].

[2] CRITEO dataset. http://labs.criteo.com/2013/12/download-terabyte-click-logs. [Online; accessed 15-Apr-2019].

[3] Harp. https://dsc-spidal.github.io/harp/. [Online; accessed 15-Apr-2019].

[4] HIGGS dataset. https://archive.ics.uci.edu/ml/datasets/HIGGS. [Online; accessed 15-Apr-2019].

[5] Kaggle 2017 survey results. https://www.kaggle.com/amberthomas/kaggle-2017-survey-results. [Online; accessed 15-Apr-2019].

[6] LightGBM github repository. https://github.com/microsoft/LightGBM. [Online; accessed commit 7282533 on Dec 9, 2018].

[7] ThunderGBM github repository. https://github.com/Xtra-Computing/thundergbm. [Online; accessed on Apr 9, 2019].

[8] XGBOOST github repository. https://github.com/dmlc/xgboost/. [Online; accessed commit 6a569b8 on Jan 3, 2019].

[9] YFCC100M dataset. http://multimediacommons.org/. [Online; accessed 15-Apr-2019].

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

[11] L. Chen, B. Peng, B. Zhang, T. Liu, Y. Zou, L. Jiang, R. Henschel, C. Stewart, Z. Zhang, and E. Mccallum. Benchmarking Harp-DAAL: high performance hadoop on KNL clusters. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*, pages 82–89. IEEE, 2017.

[12] T. Chen and C. Guestrin. Xgboost: a scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2016.

[13] T. Chen and T. He. Higgs boson discovery with boosted trees. In *NIPS 2014 workshop on high-energy physics and machine learning*, pages 69–80, 2015.

[14] C. Daning, X. Fen, L. Shigang, and Z. Yunquan. Asynch-SGBDT: asynchronous parallel stochastic gradient boosting decision tree based on parameters server. *arXiv:1804.04659 [cs, stat]*, Apr. 2018. arXiv: 1804.04659.

[15] A. V. Dorogush, V. Ershov, and A. Gulin. Catboost: gradient boosting with categorical features support. *arXiv preprint arXiv:1810.11363*, 2018.

[16] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[17] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[18] L. Guelman. Gradient boosting trees for auto insurance loss cost modeling and prediction. *Expert Systems with Applications*, 39(3):3659–3667, 2012.

[19] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, and S. Bowers. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9. ACM, 2014.

[20] J. Jiang, B. Cui, C. Zhang, and F. Fu. DimBoost: boosting gradient boosting decision tree to higher dimensions. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1363–1376. ACM, 2018.

[21] R. Jin and G. Agrawal. Communication and memory efficient parallel decision tree construction. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 119–129. SIAM, 2003.

[22] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. LightGBM: a highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3149–3157. Curran Associates, Inc., 2017.

[23] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Operating Systems Design and Implementation (OSDI)*, 2014.

[24] Q. Meng, G. Ke, T. Wang, W. Chen, Q. Ye, Z.-M. Ma, and T. Liu. A communication-efficient parallel algorithm for decision tree. In *Advances in Neural Information Processing Systems*, pages 1279–1287, 2016.

[25] M. Nishio, M. Nishizawa, O. Sugiyama, R. Kojima, M. Yakami, T. Kuroda, and K. Togashi. Computer-aided diagnosis of lung nodule using gradient tree boosting and bayesian optimization. *PloS one*, 13(4):e0195875, 2018.

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

[27] B. Peng, B. Zhang, L. Chen, M. Avram, R. Henschel, C. Stewart, S. Zhu, E. Mccallum, L. Smith, T. Zahniser, J. Omer, and J. Qiu. HarpLDA+: optimizing latent dirichlet allocation for parallel efficiency. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 243–252, Dec. 2017.

[28] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *VLDB*, volume 96, pages 544–555. Citeseer, 1996.

[29] S. Si, H. Zhang, S. S. Keerthi, D. Mahajan, I. S. Dhillon, and C.-J. Hsieh. Gradient boosted decision trees for high dimensional sparse output. In *PMLR*, pages 3182–3190, July 2017.

[30] A. Srivastava, E.-H. Han, V. Kumar, and V. Singh. Parallel formulations of decision-tree classification algorithms. *Data Mining and Knowledge Discovery*, 3(3):237–261, Sept. 1999.

[31] S. Tyree, K. Q. Weinberger, K. Agrawal, and J. Paykin. Parallel boosted regression trees for web search ranking. In *Proceedings of the 20th international conference on World wide web*, pages 387–396. ACM, 2011.

[32] Z. Wen, B. He, R. Kotagiri, S. Lu, and J. Shi. Efficient gradient boosted decision tree training on GPUs. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 234–243, May 2018.

[33] Y. Xia, C. Liu, Y. Li, and N. Liu. A boosted decision tree approach using bayesian hyper-parameter optimization for credit scoring. *Expert Systems with Applications*, 78:225–241, 2017.

[34] B. Zhang, B. Peng, and J. Qiu. Parallelizing big data machine learning applications with model rotation. *New Frontiers in High Performance Computing and Big Data*, 30:199, 2017.

[35] H. Zhang, S. Si, and C.-J. Hsieh. GPU-acceleration for large-scale tree boosting. *arXiv:1706.08359 [cs, stat]*, June 2017. arXiv: 1706.08359.

[36] Y. Zhang and A. Haghani. A gradient boosting method to improve travel time prediction. *Transportation Research Part C: Emerging Technologies*, 58:308–324, 2015.

[37] Z. Zhou and J. Feng. Deep forest: towards an alternative to deep neural networks. *arXiv preprint arXiv:1702.08835*, 2017.