# 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 optimization. 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 inefficiency of random memory access, is identified. Accordingly, we propose HarpGBDT, a new GBDT system designed from the perspective of parallel efficiency by a co-design of the algorithm and system. First, 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 the algorithm's accuracy. Secondly, we organize both the training data and model data in blocks and propose a mixed mode of parallelism. By changing the configuration of the block size and parallel 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) [16] is a widely applied machine learning algorithm. It is not only one of the most popular algorithms in Kaggle data analytics competitions [5] but also an important method to solve industry production level problems such as click-through rate(CTR) prediction that deals with the impression of billions of advertisements [18]. In recent years, it has been successfully applied to many different domains, such as: Higgs boson classification [12], credit scoring [32], computer aided diagnosis [24], insurance loss cost modeling [17], and freeway travel time prediction [35].

GBDT is a boosting method that builds consecutive decision trees in a strict 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 [9] and GBDT itself. Decision tree based deep models also demonstrate a comparable performance over deep learning models in [36]. Parallel decision tree construction is the key for a high-performance GBDT implementation and has been extensively studied in the data mining community [20], [25], [27], [29]. Recent work on GBDT [11], [21], [30] shows 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. However, most of the work lack parallel performance studies and are not tailored for 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.

In this paper, we investigate a parallel GBDT system that is efficient when increasing the number of processors, the problem size, and the model size. Based on our experiences of parallelizing machine learning algorithms in the Harp framework [3], [10], [26], [33], we build it on co-design of algorithm and system optimization. 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 GBDT training systems and summarize their design features for parallelism.
- Analyze the hotspot of popular data parallelism and feature parallelism implementations.
- 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 can reproduce existing approaches as special cases with particular configuration choices.
- 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 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.

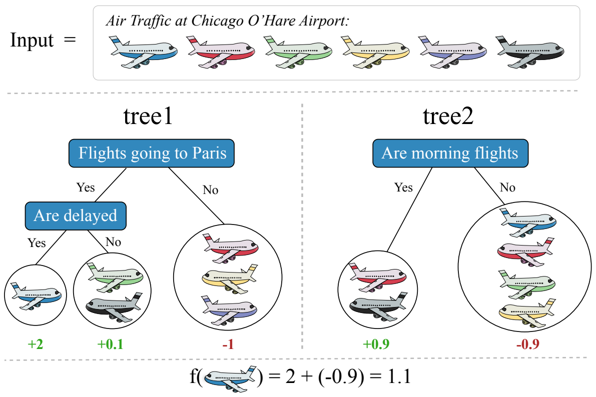


Fig. 1: An example of a tree ensemble of two trees

Second order approximation [15] is applied to a general regularized objective function [11],

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

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

Optimal weight $w_j^*$ and objective value for 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 instance set of leaf $j$. Then a score function can be derived from $\widetilde{\mathcal{L}}^{(t)}$ to guide node splitting into two subset $\langle L, R \rangle$.

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

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

Starting 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, *FindSplit*, *BuildHist*, and *ApplySplit*. As in Figure 2, each input data instance is assigned to a leaf node, denoted by different colors. 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. A technique named *histogram* is generally utilized to reduce the number of split point candidates, in which feature values are grouped into *bins*. Then FindSplit enumerates all possible split points, as pairs of (feature, value), in GHSum by calculating the split

**Algorithm 1:** BuildDecisionTree()

```
input  : dataset $\mathbb{D} = (x_i, y_i)_{i=1}^N$, gradients $GH = (g_i, h_i)_{i=1}^N$
output: tree $f(x)$
1  begin
2      q: priority queue
3      root: root node of tree
       // collect statistical summary GHSum for
          a node
4      BuildHist(root)
       // find the best split point for a node
          based on GHSum
5      FindSplit(root)
6      q.push(root)
7      while q is not empty do
           // pop nodes according to the tree
              growth policy
8          nodes = q.pop()
           // update the tree by splitting each
              node to the left and right children
              according to the split point found
              by FindSplit
9          children = ApplySplit(nodes)
10         for node in children do
11             BuildHist(node)
12             FindSplit(node)
13             q.push(node)
```

**Algorithm 2:** BuildHist()

```
input  : dataset $\mathbb{D} = (x_i, y_i)_{i=1}^N$, gradients $GH = (g_i, h_i)_{i=1}^N$,
         M:# features, B:# bins, node
output: histogram of gradients $GHSum \in \mathbb{R}^{B \times M}$
1  begin
2      for $x_i \in$ node do
3          for m =1 to M do
4              $GHSum[m,k].g += g_i$, where $x_{im} \in k_{th}$ bin
5              $GHSum[m,k].h += h_i$, where $x_{im} \in k_{th}$ bin
```

loss change according to Eq.3, and picks up the one with the maximum score. Finally, ApplySplit expands the tree by adding two children 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 size of a tree, 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, therefore is exponential to the tree size $D$, as $\mathcal{O}(MB2^D)$. ApplySplit contains simple

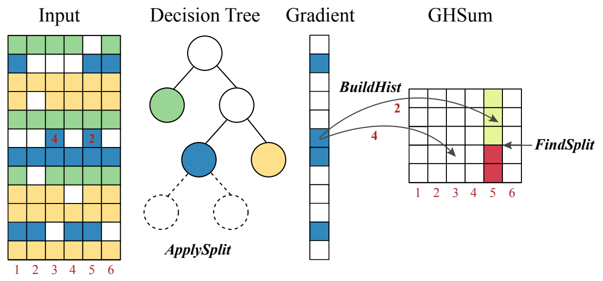
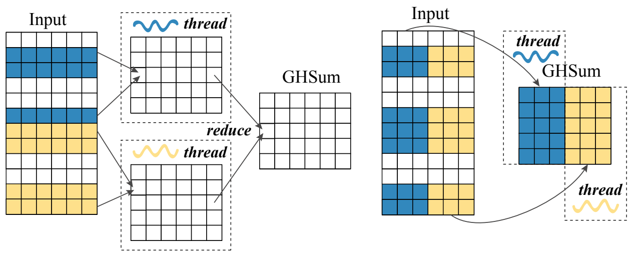


Fig. 2: Illustration of Building a Decision Tree. Input is the input feature vectors with feature values mapped to "bin" id. Gradient contains the first order and second order gradients. GHSum maintains the statistical summary of gradients.

operations and is relatively trivial, as $\mathcal{O}(2^D)$. BuildHist in leafwise is irregular because the number of instances in each node 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 typical patterns to parallelize a serial machine learning algorithm. They describe how data are partitioned among parallel workers and how these workers are synchronized along with the proceeding of the algorithm. In GBDT, 'Data' refers to input, and 'Model' refers to the intermediate data created within the algorithm to learn the decision tree. Figure 3(a) illustrates the data parallelism 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 Figure 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* [11] and *LightGBM* [21] are two state-of-the-art GBDT systems. In its original paper, XGBoost proposes a feature parallelism 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 evolves fast, adding new tree-building modules along the time. One latest module, tree_method=hist, changes to data parallelism, and achieves better performance. We refer XGBoost to this specific data parallelism version. LightGBM adopts a feature-wise model parallelism approach.

## III. Analysis of Existing GBDT Systems

In this section, we do hotspot analysis of the two representative systems, XGBoost and LightGBM, to investigate the efficiency of the parallel design of state-of-the-art GBDT trainers. They are implemented in C++ and support multi-threading by OpenMP. We run this experiment on a machine with 36 physical cores and fix the thread number to 32. Detailed experimental settings, such as hardware and software configurations and datasets, are described in Section V-A.

### A. Hotspot Analysis

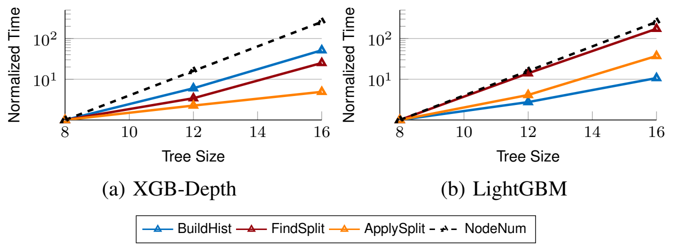


Fig. 4: Trend of Training Time Breakdown Over Tree Size on HIGGS dataset. Execution time per tree for the three core functions are normalized over the time of tree size 8.

We evaluate the two systems with the execution time breakdown on HIGGS dataset. BuildHist is identified as the hotspot, which occupies 90% time for LightGBM, 60% time for XGBoost at tree size 8. In Figure 4, suffixes of "-Depth" and "-Leaf" refers to depthwise and leafwise tree growth method used in the trainer. As LightGBM only supports leafwise method, no suffix is used. Figure 4 observes exponential growth $\mathcal{O}(2^D)$ for BuildHist in both XGBoost and LightGBM. However, according to the time complexity analysis in Section II, it should be $\mathcal{O}(D)$ in depthwise method. A large portion of parallel overhead has been introduced in parallelization.

TABLE I: Profiling of XGBoost and LightGBM

| Trainer | XGB-Depth | XGB-Leaf | LightGBM |
|---|---|---|---|
| Average CPU Utilization | 13.9% | 13.9% | 19.2% |
| OpenMP Barrier Overhead | 42% | 42% | 23% |
| Average 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. Low CPU Utilization indicates 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 up to 42%. Both of them also show a high Memory Bound above 50%, which means over 50% of CPU cycles are waiting due to 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 top one leaf with the largest loss change score to split. Therefore, they are constrained to parallelize 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 parallelizes tree building leaf by leaf. Therefore, the number of threads synchronization are 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. It explains the observed high OpenMP barrier overhead and exponential growth of execution time of BuildHist in Figure 4.

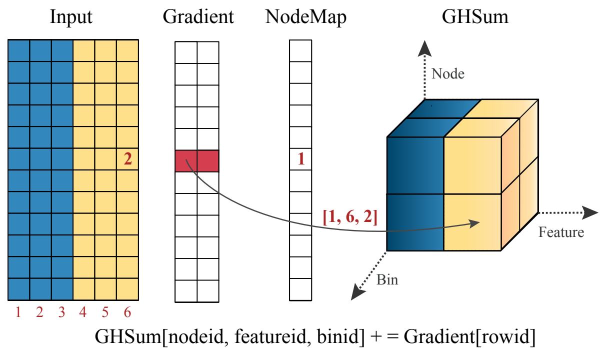


Fig. 5: Computation and Memory Operations in BuildHist

*2) Memory Bound:* The hotspot function BuildHist involves four major data structures, including Input, Gradient, NodeMap, and GHSum, see Figure 5. NodeMap maintains the membership of the data instances to the tree node, and it dynamically changes when the tree splits and grows.

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. At least one of the three indexes of request to GHSum would change dynamically, leading to random memory access. No a static memory layout for GHSum exists that can support sequential accesses all the time.

## IV. HarpGBDT: Design and Implementation

In order to improve the efficiency of parallelization, we design HarpGBDT with optimizations to reduce synchronization overhead, increase concurrency, reduce random memory access, and improve cache efficiency.

### *A. Block-wise Parallelism*

We begin with investigating the concurrency in the parallel tree construction. In data parallelism, 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 Figure5, can be the basic unit. No conflict of model updates will exist among threads in this way. A more general parallel solution can be a mixture of data parallelism and model parallelism.

We propose to use ***Block*** as the basic unit for data organization and parallelism. By viewing both GHSum and Input as three dimensional data, $\langle node, bin, feature\rangle$ 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, traditional feature-wise 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 [11], denoted as XGB-Approx, row blocks were proposed to mitigate the long-distance random memory access to Gradient in a feature-wise method, which 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 plain 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.

Secondly, standard data parallelism equals to $\langle X, X, 0, 0\rangle$. 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.

Beyond these two widely used parallel designs, many other options are not fully explored.

1) ***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 redundant 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*** "bin" level parallelism has not been discussed in related work. It enables model parallelism in case the number of features is small. When organizing Input with "bin" dimension partitions, each data cube of input becomes sparse because each feature column contains only one bin value in each row of

the input. An additional cost of memory access to the sparse data will be introduced if "bin" level parallelism is applied.

3) ***Node level parallelism*** "node" level parallelism is also not fully explored in related work. Node-level parallelism cannot be utilized in the beginning phase of tree building, where the number of leaf nodes is smaller than available CPU cores. Further, the leafwise growth method adds a dependency between tree nodes, strictly processes one node after another; in this case, node-level parallelism is also not available. On the other hand, node level parallelism has some unique advantages. Set node_blk_size enables a trade-off between less number of thread synchronization and larger size of the memory region of 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.

### *B. TopK Tree Growth Method*

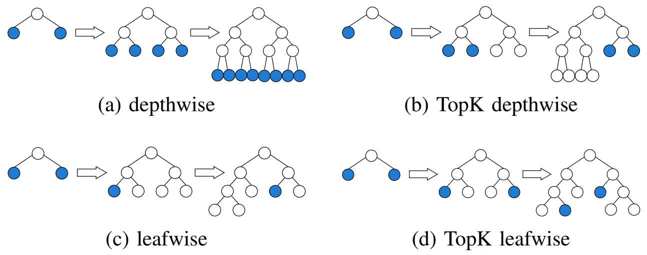


Fig. 6: Illustration of Tree Growth Method. (a)(c) are examples of standard depthwise and leafwise. (b)(d) are examples of TopK methods(K=2). Blue nodes refer the candidates selected to split.

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 Figure 6(b). TopK is a mixture of depthwise and leafwise growth methods such that enables the trade-off between robust and effectiveness. 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. Top $K$ method also supports the depthwise mode. As in Figure 6(a), only a subset of $K$ leaves are selected each time, rather than all leaves selected at the same time in depthwise. The same tree would be built.

### *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 node-level parallelism, which is not efficient when the number of leaves to split is less than the number of threads. Another scenario is data parallelism. In case of a dataset with a small number of features, data parallelism is a right 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. At the end phase, data parallelism can switch back. As we discussed in Section IV-A, synchronization between threads working on different nodes is not necessary for depthwise growth method, and it is also not mandatory in leafwise method if not confined by a strict top $K$. With TopK growth method, one synchronization of the working threads is needed after processing K leaves to select the next global top K candidates. We refer this strict topK method as "SYNC". If we further lose the constraint of top K and let $K$ threads to select the top candidate as best as they can, no such synchronization is needed anymore. We refer this loosely coupled TopK method as "ASYNC".

TABLE II: Four modes of parallel designs for GBDT.

| **Mode** | *Description* |
|---|---|
| DP | data parallelism |
| MP | model parallelism |
| SYNC | mix mode (DP, MP, DP) |
| ASYNC | mix mode (X, node parallelism, X) |

In summary, as in Table. II, based on block-wise parallelism and TopK growth method, four modes can be configured with a specific block size to fit the scenarios with different input shape and size to achieve optimal performance.

### *D. Reducing Thread Synchronization Overhead*

Reducing the number of for-loops is one straight-forward solution to reduce thread synchronization overhead. By setting the node_blk_size to $H$, $H$ selected candidates will be scheduled as a single task. In this way, the number of for-loops 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. Of course, splitting nodes in tree building is not a pleasingly 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 when they access the same row of Input. To reduce the influences of this 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.

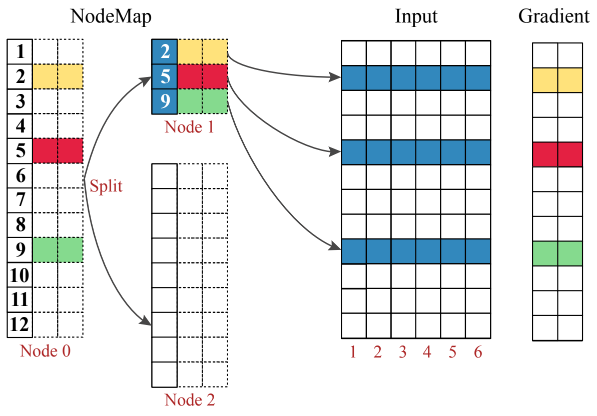


Fig. 7: MemBuf: Extending NodeMap with a replica of Gradient by keeping (rowid, gradients) pairs.

***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 region of small size 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.

Data parallelism processes the whole row of input, 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 can be 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 blockwise method, this region size can be configured, which is $16 \times bin\_blk\_size \times feature\_blk\_size \times node\_blk\_size$. It enables us to keep the balance between read and write operation efficiency by setting the size parameters corresponding to the shape of the input dataset.

## 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 machine with 36 physical cores, we fixed the thread number to 32. As for the software configuration, 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 is used as the performance profiling tool.

*2) GBDT Implementations:* XGBoost [7], 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 efficiency. Rather than write from scratch, this strategy enables quick prototype and to do a precise performance evaluation on the extended features by controlled experiments.

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 [8] | 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, see Table. III, where $S = \frac{\#element}{N \times M}$ represents the sparseness, $CV = \frac{stdev}{mean}$. CV measures the dispersion on the distribution of the number of bins of all the features. The larger this number, the more uneven of the distribution is, which leads to 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. Detailed descriptions to the datasets are online at code repository of HarpGBDT.

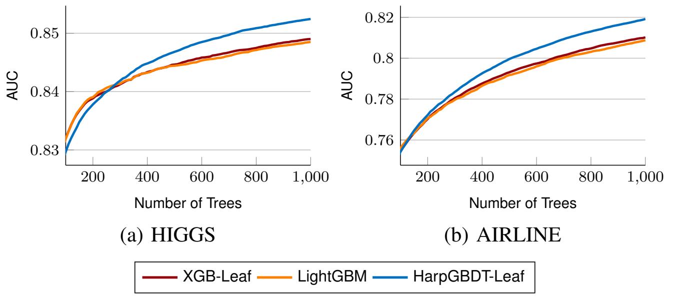
*4) Algorithm and Evaluation Parameters:* We fix the training related parameters as: $learning\_rate = 0.1$, $\gamma = 1.0$, $\lambda = 1.0$, $min\_child\_weight = 1$ (minimum sum of instance weight needed in a child), and use logistic regression loss for all the binary classification tasks. As we focus 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. Therefore, algorithm optimizations, such as feature bundling, category feature encoding, and sampling, can potentially lead to large performance difference but are not considered in the experiments.

Performance evaluation focus on training time, the wall clock elapse execution time that excludes the time spent on data loading and one-time initialization. As GBDT training is irregular, the workload of computation on consecutive trees gradually shrinks due to the decreasing of the gain to split the fine-tuned trees later on. Different pruning algorithms adopted in the trainers can affect the execution time of tree building in the later phase. Therefore, we use the average training time per tree for the first 100 trees as the efficiency metric in order to reduce the influences of this factor.

Area Under The Curve (AUC) is used to evaluate accuracy. When considering the accuracy, training time to achieve the same highest accuracy 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.

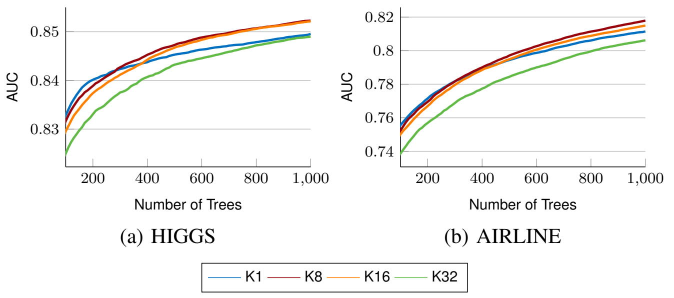
## *B. Convergence of TopK Tree Growth Method*

Figure 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 on both HIGGS and AIRLINE.



(a) HIGGS (b) AIRLINE

Fig. 8: Comparison of Convergence Rate. ($D = 8$,$K = 8$)



(a) HIGGS (b) AIRLINE

Fig. 9: Influences of K on Convergence Rate. ($D = 8$,ASYNC mode)

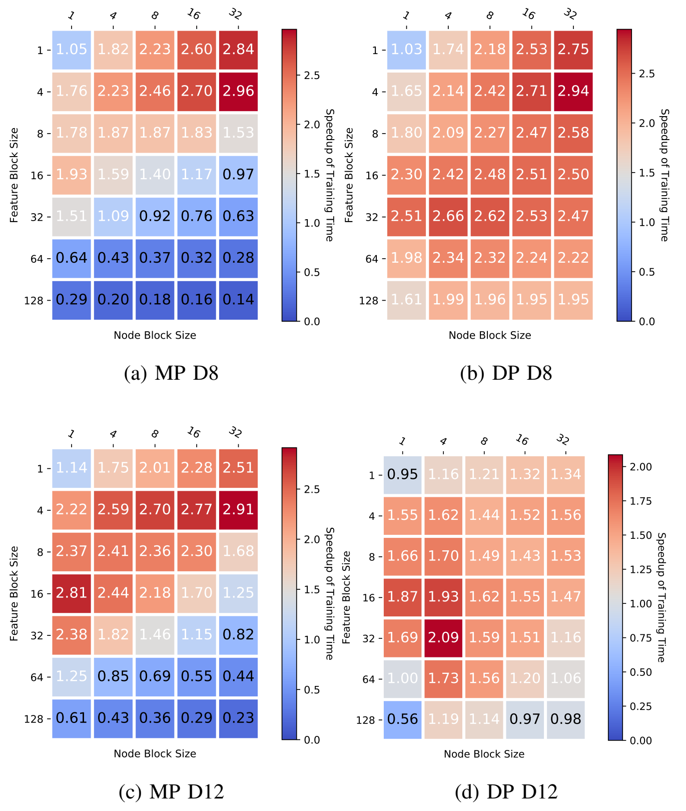
Fig 9 demonstrates that accuracy is robust for a large range of $K$. Accuracy under $K = 16$ can catch up very fast and exceed the standard method ($K = 1$). $K = 32$ shows a larger gap in the beginning and catches up slowly. We run this experiment under a worst-case condition for large $K$ by setting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

## *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 influences of configurations on the performance. We set the thread number $T = 32$ and $bin\_blk\_size = 256$ to 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) |



(a) MP D8 (b) DP D8

(c) MP D12 (d) DP D12

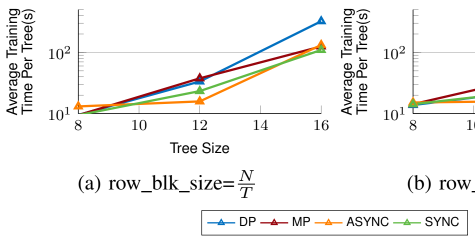
Fig. 10: Training Time Speedup over Standard Model Parallelism. Leafwise Growth. MP refers to model parallelism, DP refers to data parallelism. Tree size denoted as D#.

Figure 10 shows the influences 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:

- A maximum of 2.94× to 2.96× speedup is observed for Data Parallelism and Model Parallelism. A significant performance gain can be achieved by adjusting the block size parameters only. Data Parallelism is more robust corresponding to the block size parameters that it gives better performance than standard baseline in most cases, while MP cannot fully utilize CPU cores when the block number is less than the number of cores.
- Both Data Parallelism and Model Parallelism prefer a

medium size of feature block when node_blk_size is 1, as in the first columns of Figure 10 where 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 prefers large feature blocks while write operation prefers small ones.

- Mutual restriction exists between these two parameters. In Model Parallelism, when the feature block size is small enough to provide enough blocks to the scheduler, larger node_blk_size boosts the performance, such as the cases of feature_blk_size smaller than 4. When feature block size is big, increasing the size of the node block degrades the performance. The best configurations for Model Parallelism are along the secondary diagonal. In Data Parallelism, when the feature block size is small enough that write operations still have low cache miss rate, larger node_blk_size boosts the performance, such as the cases of feature_blk_size smaller than 16. When feature block size is big, the advantages of large node_blk_size are soon offset by the degradation of cache efficiency.



(a) row_blk_size=$\frac{N}{T}$ (b) row_blk_size=$\frac{4N}{T}$

Fig. 11: Performance of Parallelism Modes over Tree Size.

Figure 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 $\langle 32, 4 \rangle$ at D12 for DP, and $\langle 4, 32 \rangle$ for the other modes. We have the following observations:

- Data Parallelism works the best at D8, and its performance degrades when the tree goes larger. This is because the reduce operation for the model replicas grows proportional to the number of tree nodes.
- Model Parallelism scales better than Data Parallelism. Supporting model updates without conflicts not only leads to a smaller memory footprint but also enables Model Parallelism to scale.
- SYNC mode is effective that it achieves better performance than Data Parallelism and Model Parallelism.
- ASYNC shows the best capability of scaling over the tree size. It not only reduces thread synchronization overhead but also enables the use of larger feature block size.
- D16 is an extreme case as 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. Figure 11(a) shows the performance of all modes except Model Parallelism degrade in D16. For Data Parallelism related modes, setting $row\_blk\_size = \frac{N}{T}$ makes a large number of small row set and thus too many small tasks. For ASYNC, the synchronization overhead of updating shared data structure goes up when it updates at a high frequency. By adjusting row_blk_size to a larger value, as in Figure 11(b), Data Parallelism and ASYNC boost up about 50%. Model Parallelism and SYNC do not gain much because they adopt big node_blk_size and are less impacted by this synchronization problem caused by small tasks.

## D. Optimization Effectiveness

We evaluate the effectiveness of the proposed optimizations on SYNSET. We start from the baseline of standard Model Parallelism($feature\_blk\_size = 1, K = 1$) and Data Parallelism ($feature\_blk\_size = 128, K = 1$), add four types of optimizations incrementally. First, 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 to mix mode, SYNC in D8 and ASYNC in D12. Table. V shows the training time speed up in each step. "+Block" for Data Parallelism incurs performance loss 13% in D8, and recovers by "+MemBuf". This demonstrates that a single optimization does not guarantee performance gain under every scenario and multiple optimizations work better together.

TABLE V: Performance Gain with Itemized 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: Profiling of HarpGBDT

| Trainer | Depth-DP | Leaf-DP | Leaf-ASYNC |
|---|---|---|---|
| Average CPU Utilization | 27.5% | 28.5% | 28% |
| OpenMP Barrier Overhead | 9% | 8% | 8% |
| Average Latency(cycles) | 15 | 16 | 15 |
| Memory Bound | 38% | 41% | 40% |

## E. Parallel Efficiency

We run experiments on HIGGS to evaluate the parallel efficiency for the three systems. For HarpGBDT, we adopt Data Parallelism mode for D8 and ASYNC mode for larger trees, set the parameters $K = 32, feature\_blk\_size = 4, node\_blk\_size = 32$.

Compared with Table I, the OpenMP barrier overhead is significantly reduced in HarpGBT as shown in Table VI. Data Parallelism with large $K$ and node_blk_size efficiently reduce the number of for-loops. ASYNC does not show advantages on reducing barrier overhead in D8, but it works well when tree size goes larger with the ratio drops to 0.02 at D12. Memory-related metrics also improve due to the better block size configurations. Accordingly, Figure 12 shows that HarpGBT scales better over the tree size.

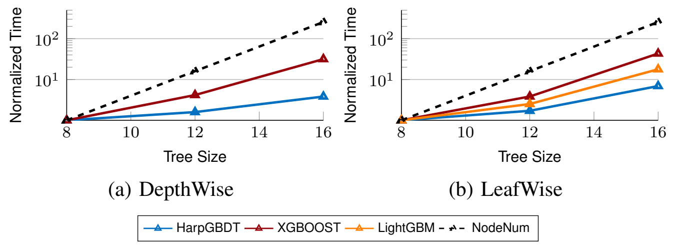


Fig. 12: Trend of Training Time over the tree size.

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 Figure 13(a), all the implementations do not achieve a good strong scaling performance on HIGGS which is a relatively small dataset. HarpGBDT relatively scales better, shows that the optimizations enable it to utilize more computation resources efficiently. As GBDT is a memory-bound application, weak scaling is more suitable for evaluating the parallel efficiency. By keeping the workload for each thread the same, we increase the dataset size proportional to the number of threads by duplicating the HIGGS dataset. As in Fig 13(b), HarpGBDT shows significant better weak scaling efficiency.

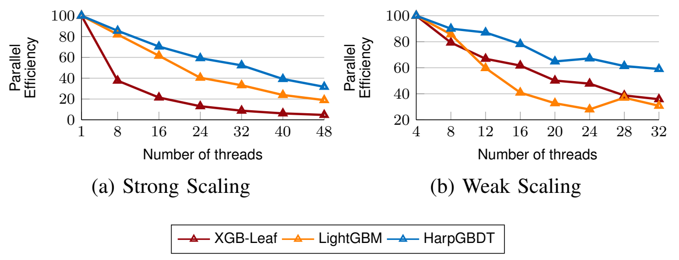


Fig. 13: Parallel Efficiency on HIGGS D12.

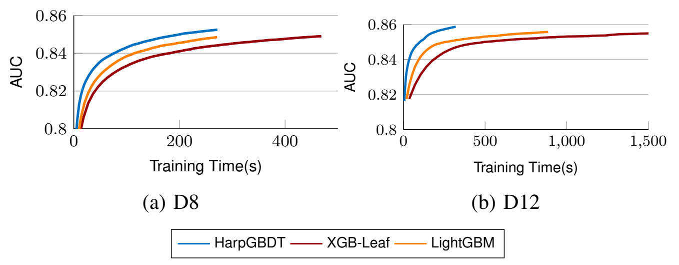


Fig. 14: Trend of Convergence Speed.

Finally, we evaluate the influences of parallel efficiency on the convergence speed of the three systems. In Figure 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 Figure 14(b), when increasing tree size to D12, HarpGBT shows a strong advantage owing to the performance optimizations. It converges and finishes the job much faster.

## F. Overall Performance

Four datasets are used for the overall performance measurement. We separately compare training time speedup and convergence speedup.

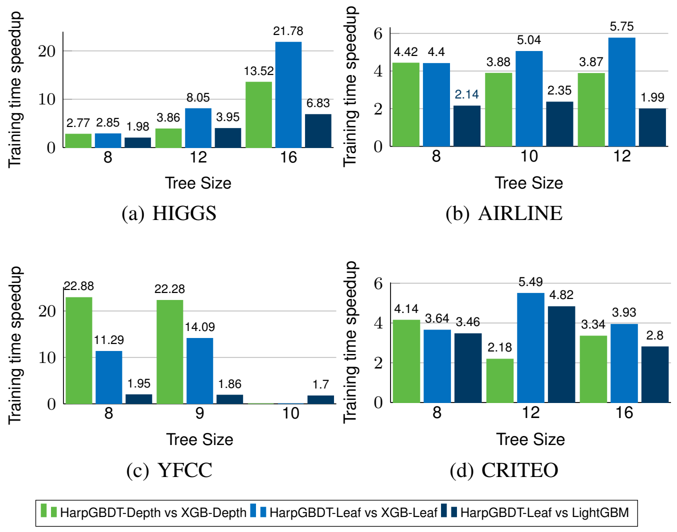


Fig. 15: Training Time Speedup on four datasets.

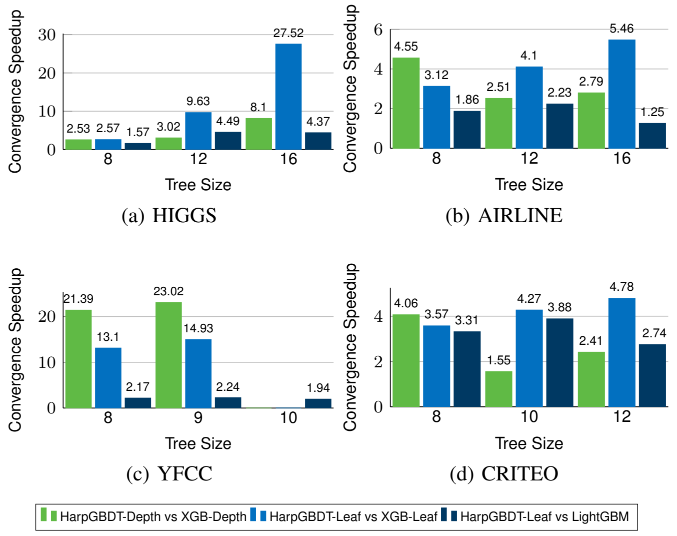


Fig. 16: Convergence Speedup on four datasets.

YFCC and AIRLINE are two types of input with a very different shape. For fat matrix input YFCC, as we have discussed in section V-C, standard DP does not work well due to inefficient write operations, standard feature-wise Model Parallelism is also inefficient due to the overhead of redundant read operations. HarpGBDT shows more than 10x speedup over XGBoost in this scenario, and more than 1.7x training time speedup and 1.9x convergence speedup over LightGBM. For thin matrix input AIRLINE, Data Parallelism is a better choice than standard feature-wise Model Parallelism 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. HarpGBT keeps around 2x faster than LightGBM, but the convergence speedup is less than 2x. It is hard to learn a robust model by leafwise method on CRITEO, even after we set $min\_child\_weight = 100$ to control over-fitting. The leafwise method builds a deep tree 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.

On average 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

In addition to CPU architectures, using GPU to accelerate GBDT algorithm is also an important topic. XGBoost and LightGBM are generally used as the baselines. ThuderGBM [31] reports 1.5-2x speedup with Titan X vesus 2x10-core Xeon E5-2640. CatBoost [14] claims 2.3x speedup with NVIDIA P100 versus Xeon E5-2660, and [34] shows 7-8x speedup with GTX 1080 versus 2x14-core Xeon E5-2683. Promising performance speedup is demonstrated. However, training on a large dataset with GPU is still challenging as the communication between main memory and device becomes the hotspot.

Distributed GBDT system is another important topic. Both XGBoost and LightGBM build distributed GBDT upon a collective communication layer. [23] proposes PV-Tree, a voting approximation which avoids a large volume of data communication. DimBoost [19] deploys a large scale distributed GBDT system based on the parameter server architecture [22] and integrates with Yarn and HDFS which is evaluated in a typical industry production environment. [13] 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. Extending HarpGBDT to a distributed system is one of our future work.

Many related works focus on algorithm optimizations. pGBRT [30] first proposed to utilize *histograms* to speed up the creation of decision tree in the GBDT algorithm. LightGBM [21] and DimBoost [19] all propose feature bundling methods to deal with sparse features. GBDT-Sparse [28] proposes L1 regularization for high dimensional sparser output problem and demonstrates 40x speedup. These algorithm optimizations are orthogonal to our work on parallel efficiency. Combination the techniques from both directions would be certainly important to build a high-performance GBDT system.

## VII. Conclusions and Future Work

In this paper, we focus on improving the parallel efficiency of the decision tree building in GBDT algorithm. We propose a block-wise parallelism strategy and a topK extension of tree growth method to fully utilize the potential unit of parallelism in the GBDT algorithm. By adjusting the block configuration, performance related to memory access can be tuned. By selecting different parallelism method according to the shape of the input matrix and the phase of tree growth, the overhead of thread synchronization can be reduced significantly. Performance evaluations on four open datasets with quite different shapes and characteristics show that HarpGBDT, our implementation of these optimizations, outperforms two state-of-the-art systems. The optimization achieves a speedup of 2.6x to 8.5x on average and up to 27x on large tree size.

Extending HarpGBDT to support distributed training is one of the future work. Many approaches of optimizations are not covered in this paper and can also be interesting directions for our future work. First, scalability for many-core architectures: the approach described in the paper helps to improve multi-core scaling, but does not completely cover this, especially for NUMA systems. Second, optimizations for other functions beyond BuildHist, such as histogram initialization, can be important in practice. Third, micro-optimizations on instructions level: vector instruction sets and explicit prefetching demonstrate advantages over the vectorization optimizations automatically generated by compilers in Intel(R) Data Analytics Acceleration Library. We can adopt their optimizations with our described approach to achieve higher results.

## 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] XGBOOST GitHub Repository. https://github.com/dmlc/xgboost/. [Online; accessed commit 6a569b8 on Jan 3, 2019].
[8] YFCC100M Dataset. http://multimediacommons.org/. [Online; accessed 15-Apr-2019].
[9] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
[10] 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.
[11] 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.
[12] 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.

[13] 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.

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] Z.-H. Zhou and J. Feng. Deep forest: Towards an alternative to deep neural networks. *arXiv preprint arXiv:1702.08835*, 2017.