HarpGBDT: Optimizing Gradient Boosting Tree for Parallel Efficiency

Bo Peng[1] Langshi Chen[1] Jiayu Li[1] Miao Jiang[1] Selahattin Akkas[1]
Egor Smirnov[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, sergey.khekhnev, andrey.nikolaev}@intel.com

ABSTRACT

Gradient Boosting Decision Tree (GBDT) is a widely used machine learning algorithm in classification and regression tasks. Training large GBDT models on big datasets involve exponential computation and memory access over the tree depth and is challenging for both algorithm optimization and system design. In this paper, we focus on the study of parallel efficiency and conduct a comprehensive analysis of state-of-the-art GBDT training systems. They are all MPI/C++ based implementations of XGBoost, LightGBM, and HarpGBDT. In HarpGBDT, we propose a combination of intra-node block-based parallelism and inter-node optimized collective communication. It effectively reduces the overhead of irregular memory access and communication, and outperforms other approaches in our detailed performance evaluation.

PVLDB Reference Format:
. HarpGBDT: Optimizing Gradient Boosting Tree for Parallel Efficiency. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Gradient Boosting Decision Tree (GBDT) [10] is an important machine learning algorithm that has been widely applied to real problems in many different domains. Large scale GBDT trainers are implemented to handle billions of data instances with thousands to millions of features each. However, the parallel design in the existing tools are not well explained and are often built in an ad-hoc fashion. In this paper, we aim to investigate the critical factors and potential solutions to build the system for better parallel efficiency. Note that a fast implementation targets for some small datasets and tree depth does not necessarily scale well to high data volume, deep trees, and cluster size. A key question is how to parallelize a sequential algorithm so that it can scale to large problems.

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

The boosting procedure in GBDT is strictly sequential. Therefore, the way of parallelizing tree building for each tree is the object to investigate. It is a memory bound computation with a model size exponential grows over the tree depth. Meanwhile, parallel workers need to synchronize the model.

We explore the system design space for GBDT trainers and investigate the tradeoffs between different optimizations. Instead of focusing only on accuracy and execution time, we optimize based on parallel efficiency and analyze the memory footprint and scaling. We propose to use a general block-based parallelism model which can instantiate to different approaches by changing the configuration. Our main contributions can be summarized as follows:

- Review state-of-the-art GBDT training systems and summarize their design features on parallelism.
- Analysis the bottleneck of popular data parallelism and model parallelism implementations.
- Propose block-based parallelism to optimize the overhead of thread synchronization and memory access.
- Implement HarpGBDT based on block based parallelism and achieves 2.6 to 7 × speedup.

The outline of this paper is as follows: Section 6 introduces the background of the GBDT algorithm and related work, while Section 3 analyzes the architecture and parallel efficiency of existing solutions. Section 4 describes our system design and implementation details of HarpGBDT and Section 5 presents experimental results coupled with a performance analysis. Finally, Section 7 draws conclusions and discusses future work.

2. GRADIENT BOOSTING TREE ALGORITHM

2.1 Gradient Boosting Tree Algorithm

Gradient Boosting Decision Tree(GBDT) is one popular tree ensemble model. First, it is a stage-wise additive boosting algorithm that trains on the previous residual at each step, and the final prediction is the sum score of all the sub-models. Second, it adopts the decision tree as the weak learner to learn each sub-model. It provides a general framework supporting a wide range of loss functions and regularization methods, and the score function to build the decision tree is derived from the object function directly.

To find function approximation $\widehat{y_i} = \phi(x_i)$ that minimize regularized objective $\mathcal{L}(\phi) = \sum_{i=1}^{n} \ell(\widehat{y_i}, y_i) + \Omega(\phi)$, GBDT

adopts a boosting approach, $\widehat{y}_i = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i)$, where $f_k(x)$ is weight of the leaf node of a decision tree that maps each instance x to this leaf node.

Second order approximation [9] is applied on a general reguralized objective function [6]

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

where g_i, h_i are the first and second order gradients; $g_i = \partial_{\widehat{y}_i^{(t-1)}} \ell(\widehat{y}_i^{(t-1)}, y_i)$ $h_i = \partial^2_{\widehat{y}_i^{(t-1)}} \ell(\widehat{y}_i^{(t-1)}, y_i)$

Optimal weight w_j^* and objective value for leaf j can be obtained as

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

Then $\widetilde{\mathcal{L}}^{(t)}$ is used to devise the score function to guide node splitting into two subset $\langle L, R \rangle$.

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

where $G_J = \sum_{i \in I_J} g_i$, $H_J = \sum_{i \in I_J} h_i$

Algorithm 1: BuildTree()

```
input : dataset D = (x_i, y_i)_{i=1}^n, gradients GH = (g_i, h_i)_{i=1}^n
output: trees f(x) = w_q(x)
begin
    q: priority queue
    root: root node of tree
    // calculate the statistics summary GHSummary for
       node
    BuildHist(root)
    // find the best split feature and value for node
       based on GHSummary
    FindSplit(root)
    q.push(root)
    while q is not empty do
        // pop nodes from the priority queue according
           to the tree growth policy
        nodes = q.pop(K)
        // update the tree by splitting the node to
           the left and right children according to
           the split info found by FindSplit
        children = ApplySplit(nodes)
        for node in children do
            BuildHist(node)
            FindSplit(node)
            q.push(node)
```

As an additive model, building the t_{th} tree depends on the predictions based on the previous $1..t-1$ trees. This is a strict sequential procedure and cannot be parallelized. Therefore, we focus on parallelizing the single decision tree building in GBDT. Algo. 1 shows a general tree building pseudo code, composing with three core functions, FindSplit, BulidHist and ApplySplit.

FindSplit. Given the score function,as in Eq.3, a greedy algorithm is generally adopted to guide the tree node splitting in order to find the best tree structure in practice. For each feature, this greedy algorithm enumerates all the possible split points to find the one with the largest score. Then global maximum score will be selected among all the features. Normally, there can be many different types of features in one dataset, including binary features, categorical features, and real value features; and they can have very different distribution on the values and split point candidates. For features with real values, the split point is each unique feature value which can divide the tree node into left and right children by comparing their feature value with the one of the split point. Categorical features contain discrete values that do not have an order, in which comparing the feature values has no meaning. As the candidates of split points is the combination of the subset of unique feature values, simpler solutions are generally used in practice. Fix the size of the subset to a small number is feasible. Transforming categorical features into real value features by encoding is another wildly used method [8]. Label encoding is the simplest one that assigns a number to each unique category. Based on label-encoding, one-hot encoding expands one category feature to a group of binary features to avoid adding order to the feature values. Response variable replacement is another encoding method that converts category feature to real value features using the distribution of the response variables. As GBDT implementations differ in the way of dealing categorical features, we focus on the general supported method of encoding to deal with categorical features in this paper.

Algorithm 2: BuildHist()

```
input : dataset D = (x_i, y_i)_{i=1}^n, gradients
        GH = (g_i, h_i)_{i=1}^n, M:# feature, B:# bins, node
output: histogram of the node GHSum ∈ N^{M×B}
begin
    for x_i ∈ node do
        for m =1 to M do
            GHSum[m, k].g+ = g_i, where x_im belongs to
              k_th bin
            GHSum[m, k].h+ = h_i, where x_im belongs to
              k_th bin
```

BuildHist. To further reduce the number of candidates of split points, a technique called *histogram* is used in general. Instead of all unique feature values, histogram proposes to use a much smaller list of candidates selected from them. Continuous feature values are splitted into groups, called *bins*, according to their percentiles of distribution [6, 15]. Histogram is a proven method that achieves better performance without cost of accuracy. [6] proposes to re-initialize the "bins" data structure of histogram for each level of the tree growth to achieve better convergence rate. But re-initializing step is time consuming, reusing a static candidate list of split points, or static bins, is more widely used. All discussion later in this paer adopts the later static "bins" approach. BuildHist is the function to summaries the gradient statistics to prepare calculating the score according to Eq. 3. Given a initialized "bins", with splitting on one node as an example, BuildHist first goes through all the instances that belong to this node, sum up the gradient statistics that belongs to the same bin to build the gradient histogram, as in Algo. 2.

ApplySplit. ApplySplit updates the tree by splitting the

node to the left and right children according to the split info found by FindSplit.

Growth Method. There are two popular tree growth methods; one is "depthwise" that splits node level by level, and the other one is "leafwise" that select the leaf node with the largest changes in objective value to split. In Algo. 1, these two growth methods are unified by a priority queue which can support specific growth policy by specific comparison functions and the number of nodes to pop out. K is the maximum number of leaves in depthwise and is 1 in leafwise. We refer the size of a tree by a small number of tree size D instead of the number of leaves for simplicity. A tree of D size contains $2^D - 1$ leaves. In depthwise, D equals to the tree depth. In leafwise the tree is unbalanced and usually the tree depth is much larger than D.

Given a dataset $D \in N^{N \times M}$, to build a tree with size D in "depthwise" growth method. The tree will grow and stop at the scale of $L = 2^D - 1$ leaves. The time complexity of BuildHist is $\mathcal{O}(NMD)$ that goes through all the instances once at each tree level. FindSplit is $\mathcal{O}(MB)$ for each node, therefore is proportional to the number of leaves, or exponential to the tree size D, as $\mathcal{O}(MB2^D)$. ApplySplit contains simple operation of splitting a node, and is relatively trivial when compared with the other two functions, as $\mathcal{O}(2^D)$. BuildHist in "leafwise" is more dynamic that the number of instances involved in the computation in each node cannot be analytically predicted, while FindSplit and ApplySplit also keep the same complexity as in "depthwise" mode.

2.2 Parallel and Distributed GBDT Training

Data Parallelism and *Model Parallelism* are two typical patterns to parallel a serial machine learning algorithm including the decision tree building algorithm in GBDT. They describe how data are partitioned among parallel workers and how they are synchronized along the proceeding of the algorithm.

pGBRT [19] first proposed to utilize histograms to speed up the creation of regression tree in traditional GBRT algorithm. It parallelizes the construction of individual trees with a data parallelism approach, where the training data are divided among the different workers. All the workers run BulidHist procedure to cumulative statistics on its local data. A master processor merges the histograms and uses them to FindSplit and split the tree layer by layer. pGBRT also proposes to overlap communication and computation by computing histograms feature by feature and send previous one to master while moves to the next feature.

XGBoost [6] proposes to learn the model by minimizing a general regularized learning objective, which was found helpful in practice. The paper adopts a model parallelism approach, where collecting the histogram statistics are done for each feature column in parallel. With the success of the XGBoost open source project, its code evolves fast, and many new tree-building modules are added along the time. One latest module, tree_method=hist, changes to data parallelism and achieves much better performance than the version of implementation in the paper. We refer XGBoost to this specific data parallelism module in this paper.

LightGBM [13] proposes an exclusive feature bundling algorithm to speed up tree building for a dataset with a large number of sparse features, and a gradient-based one-side sampling algorithm which demonstrates better performance than standard random sampling method. Their system, LightGBM, is another state-of-the-art implementation of GBDT, which adopts a feature wised model parallelism approach. In [15], they proposes three different approaches for distributed GBDT training. The first one, attribute-parallel partition the data by column in which a binary vector indicating the split information is exchanged across machines; the second, data-parallel partition the data by row in which histograms need to exchange; the third, PV-Tree, is a voting approximation which avoids large volume of data communication.

DimBoost [12] proposes a large scale distributed GBDT system based on the parameter server architecture [14] and integrates with Yarn and HDFS for deployment which is evaluated in a typical industry production shared cluster environment, in which the memory and CPU cores are limited.

2.3 Sparsity in Data

Sparse feature is one interesting topic for GBDT. First, there are missing values in the input. Capable of dealing with missing value elegantly is one of the reasons that GBDT is widely used. One case of missing values is the zero values in the dataset which are usually not saved. Standard file format, such as libsvm format, does not save these zeros by default. Another case is the true missing value that for some reasons the specific feature value is unknown. In both BulidHist and FindSplit, the summary of statistics of gradients of missing values can be calculated by subtracting the NNZ sum from the total sum, in this way, GBDT can deal with missing values efficiently. Another source of sparse features is the encoding method. For example, one-hot encoding is one popular method to transform categorical features into real values by expanding each unique category to a new binary feature and leads to a very sparse input that the number of non-zero items are much smaller than the size of the matrix.

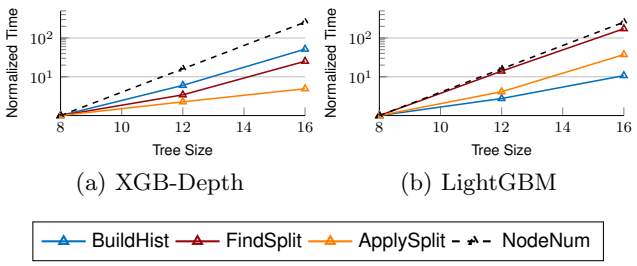
To deal with dataset with large number of sparse features, from thousands to millions features, feature bundling , such as in [13, 12], is a widely adopted method and can efficiently speed up the training. It bundles the sparse features into a much smaller number of groups when they are not active at the same time. This is treated as a preprocessing step and is supported by state-of-the-art GBDT systems. To avoid the performance variation coming from the differences of the bundling algorithms, in this paper, we run evaluations on the datasets that already preprocessed by a bundling step if needed.

3. ANALYSIS OF EXISTING GBDT SYSTEMS

XGBoost and LightGBM are two state-of-the-art GBDT trainers. XGBoost is representative of using data parallelism and LightGBM is the one using model parallelism. They are all implemented by C++, supporting shared memory multithreading by OpenMP and distributed training using MPI as the communication layer. In this section, we do bottleneck analysis of these two systems to investigate the efficiency of the parallel design.

3.1 Bottleneck Analysis

First, we run a group of experiments on the Higgs dataset with increasing tree size. The machine has 36 physical cores,



(a) XGB-Depth (b) LightGBM

Figure 1: Trend of Training Time Over Tree Size. When increasing tree size on Higgs dataset, execution time of each tree for the three core functions are normalized over that of D8.

Table 1: VTune Profiling of XGBoost and LightGBM

Trainer	XGB-Depth	XGB-Leaf	LightGBM
AverageCPUUtilization	13.9	13.9	19.2
OpenMPBarrierOverhead	0.42	0.42	0.23
Average Latency	35	37	25
Memory Bound	51.0	52.9	54

and the thread number is fixed at 32. Figure. 1 is the training time breakdown of the three core functions in GBDT training. Suffixes of "-Depth" and "-Leaf" refers to depthwise and leafwise tree growth method used for the trainer. As LightGBM only support leafwise method, no suffix is used.

BuildHist occupies 90% time for LightGBM, 60% time for XGBoost at tree size 8, denoted as "D8". At D16, these numbers go to 60% and 70% . As in Figure. 1, Both XGBoost and LightGBM have exponential growth $\mathcal{O}(2^D)$ observed for BulidHist. According to the time complexity analysis in Section. 2, the time complexity of BuildHist should be $\mathcal{O}(D)$ in case of "depth-wise" growth method. A large portion of overhead must have been introduced as the overhead of parallelism.

By hardware event counter profiling support with Intel VTune Amplifier, we can get a more precise view of the observed phenomena. In Table. 1, VTune reports high OpenMP Barrier wait overhead on both trainers. The best one, LightGBM, still spend 23% the effective CPU time in spinning. XGBoost spends up to 42%. Another important factor to performance, all the trainers show a high number of memory bound around 50%, which means nearly 50% percent of CPU cycles are waiting due to demand from load or store instructions.

3.2 Thread Synchronization Overhead

Frequent thread synchronization is one major source of parallelism overhead in current state-of-the-art GBDT trainers. Both XGBoost and LightGBM apply OpenMP in implementing the parallel tree building to fully utilize the computation power from modern many-core architecture. OpenMP provides an easy to use programming model that makes it the first choice of multi-threading programming tool. By adding #pragama before the for-loops inside the code, a serial algorithm potentially can turn into a parallel one. The complexity of threads initialization and scheduling are all encapsulated inside the OpenMP library. However, there is a cost for this easy-to-use feature that for-loop parallelization introduces a barrier wait at the end of the loop which might not be necessary from the original algorithm's aspect. When the workload among all threads is even, the barrier overhead is not a big issue according to the optimizations in OpenMP such as thread pool to minimum the overhead of thread spawning. However, when load imbalance observed in the computation, the overhead of threads spinning waiting becomes inevitable. Both XGBoost and LightGBM parallelizing BuildHist inside one leaf, therefore the number of threads synchronization are proportional to the numbers of leaves $\mathcal{O}(2^D)$. For leafwise algorithm, XGB-Leaf and LightGBM have to select the top one node with the largest loss change to split. Therefore, they are constrained to run parallelizing leaf by leaf. However, for depthwise method, the leaf nodes at the same level of the tree are independent and can be parallelized. However, as XGB-Depth adopts a data parallelism approach which maintains the model replicas in each thread local memory and reduce them at the end, to avoid uncontrolled memory foot-print of the model replicas, XGB-Depth adopts a strategy of parallelizing leaf by leaf. This can explain the observed high OpenMP barrier overhead and exponential growth of execution time of BuildHist in Figure. 1.

3.3 Memory Issue

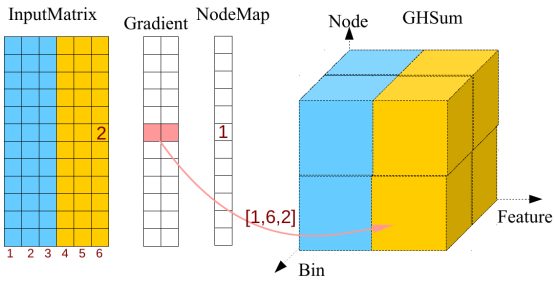


Figure 2: Computation and Memory Operation in BulidHist

The second important performance factor for GBDT is its memory bound nature, which is also a general issue when applying machine learning algorithms to big data problems. In general, a learning algorithm tries to learn a function f that achieves the least empirical loss $L(f(x), y)$ on the dataset $\langle X, Y \rangle$. Both the volume of input data $\langle X, Y \rangle$ and of the model data, intermediate data to calculate f, can be large and exceed the memory capacity of a single server. At the same time, low computation versus memory access ratio, which is defined as the ratio of floating point operations to memory access density(byte), is a general property of many machine learning algorithms.

As in Figure. 2, the core computation in bottleneck function BuildHist is to collect the statistics of gradients within the specific node and sum them up to the histogram. It involves four major data structures, *InputMatrix* is the input data which contains N rows and M columns of features, which can be sparse or dense. When utilizing histogram, a preprocessing step is often taken to maintain a map from the original feature value in each cell of InputMatrix to the "bin" id which indicates the location of the bins that this

value belongs to. This is only done once before training. *GradVec* contains the first order and second order gradients for records in each row. *NodeMap* maintains the membership of the records to the node id which it belongs to. It dynamically changes when the tree splits and grows. *GHSum* is the histogram that maintains the summary of gradients for each feature and all the tree nodes. It is a 3-D matrix.

There are one read accesses to each of the four major data structures and one write to GHSum for each cell of InputMatrix. As Double are commonly used in GHSum, Float, and Integer in other data structures, the computation to memory ratio here is

$$\frac{floating\ point\ operations}{memory\ access\ density} = \frac{2}{40} = 0.05 \tag{4}$$

Comparing with the potential capability of modern computer architecture that especially optimized for computation intensive applications, it is a quite small number.

Another aspect of challenges come from the complex nature of memory access in the machine learning algorithms. Input data can be organized to be scanned one pass by another, while the model data are generally observed with random and irregular access patterns. For example, in GBDT, the size of model data and the region to update for each instance in the input data dynamically changes when the tree grows. These algorithms prefer the systems which can efficiently provide high memory bandwidth.

Typical data parallelism solution of GBDT, such as XGBoost, access InputMatrix row by row in parallel, with each thread accesses to different row sets. This method has advantages in reading operations as it can utilize the cache line efficiently for each read operation from InputMatrix. However, because of the dynamic changing of the membership relationship between input rows and tree nodes, reading GradVec involves random memory access. If implemented naively, a long distance random access will brings large cache miss count. XGBoost proposes to split InputMatrix into smaller row blocks to mitigate this issue. Buffered memory is another common solution that helps, which copies the noncontinuous memory to a continuous buffer to accelerate later reading operations. Data parallelism solution introduces more write operations in general, as it has to write to the local replica of the model and add one reduce operation at the end. Also, when InputMatirx has a large number of features, the write operations will randomly write to a large size of memory region which is the matrix with the shape of $\#bins \times \#feature$, which can lead to a large number of LLC cache miss.

Model parallelism solution of GBDT, such as LightGBM, accesses InputMatrix column by column in parallel, with each thread accesses to different features. This method involves fewer write operations to GHSum, and the size of memory region of write is small, as $\#bins$, which can be more cache friendly. However, the write efficiency is achieved at the cost of less efficient read operations. Random read accesses is again a problem. A column wised data organization of InputMatrix is used in feature-wised parallelism to mitigate the issues of less efficient cache line utilization when comparing to the row-wised method. Moreover, some redundant read operations is now inevitable, as the same item in GradVec has to be read for many times for all the feature columns by individual threads.

Another shortcoming of feature-wised parallelism is its dependency on the number of features in the input. Datasets with small number of features, e.g., Higgs with 28 features, are hard to utilize available CPU cores that more than the number of features in solely feature-wised parallelism. Moreover, the issue of load imbalance can be more serious in this method than it is for data parallelism. For example, 3 features in Higgs contain only a small number of unique feature values and therefore small size of histogram, the cost of BuildHist on these smaller features are much less than the other full-fledged features. In the case of datasets with sparse features and missing values, load imbalance is a common setting for GBDT training.

4. HARPGBDT: DESIGN AND IMPLEMENTATION

4.1 Block-based Parallelism

In order to design an efficient parallel tree-building system, first, let us explore the potential basic unit for parallelism in this algorithm. From the data parallelism perspective of view, set of rows, or row blocks is the basic unit that can be assigned to a thread as an independent task in BuildHist. From the model parallelism perspective of view, each cell in the 3-D matrix GHSum, as in Fig.2, can be the basic unit for parallelism that no conflict of model updates will exist among threads in this way.

A general parallel solution can be a mixture of these two methods and configured using *BLOCK* as the basic unit for parallelism. A BLOCK is defined as a cube in the model GHSum and associated cube in the input InputMatrix. GHSum has three dimensions of node, feature, and bin. Size in "bin" dimension is the max size of the histogram for each feature, by default 256. Size in "feature" dimension is the number of features of the dataset, e.g., 28 in HIGGS. Size in "node" dimension is the number of leaf nodes in the current tree. As only leave nodes can be the candidates for splitting, size of "node" is no more than the max number of leaves. When a common optimization called "HalfTrick" is applied, only one child of the split node have to re-BuildHist, and the other one can obtain its histogram summary by subtracting that of its sibling from its parent node. "HalfTrick" accelerates computation at the cost double memory consumption when preserving all the model of parent nodes in memory. In this case, size of "node" dimension is about $2\times$ of the max number of leaves. InputMatrix can be viewed as a 3-D matrix correspondingly with three dimensions in the row, feature, and bin. All cubes in InputMatrix with the same $\langle feature, bin\rangle$ configurations will share the same associated model cube in GHSum.

By configuring the BLOCK by parameter $\langle row\_blk\_size, node\_blk\_size, bin\_blk\_size, feature\_blk\_size\rangle$, we can have many different designs to build decision tree in parallel. First, traditional feature-wised parallelism equals to $\langle X, X, 0, 1\rangle$ in blocked-based parallelism. In the original version of XGBoost [6], XGB-Approx, row blocks were proposed to mitigate the long-distance random memory access to GradVec in a feature-wised methods, which equals to set row_blk_size to "X" here. node_blk_size equals to 0 in XGB-Approx, which maintains a row id to node id mapping and the node id dynamically changes when the tree grows. In this way, it scans each column of InputMatrix sequentially

at the cost of writing to a relatively large region of model memory, a plain in the cube, that across all current tree leaves. node_blk_size equals to 1 in LightGBM, as we have discussed in 3.2 that it do BuildHist leaf-by-leaf. By maintaining a mapping from tree node to row set of input, it constrains the region of model memory of writing to a vector in the cube with fixed node and feature value for each thread, at the cost of random access to InputMatrix.

Secondly, traditional data parallelism equals to $\langle X, X, 0, 0\rangle$ in blocked-based parallelism. In XGB-Hist, the latest data parallelism version of XGBoost, row_block_size is not a fixed parameter but dynamically partitioned on the row set that belongs to each tree node. node_blk_size equals to 1 in order to constrain the memory footprint of the model replicas.

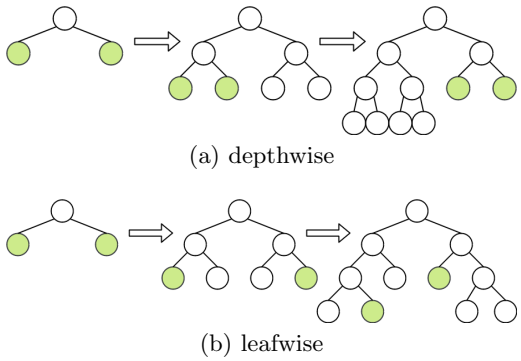
Beyond these two widely used parallel design, we can see there are many other options that are not fully explored. *feature level parallelism* Set feature_blk_size enables a trade-off of preferences between read operations and write operations. Small value, as in traditional feature-wised parallelism, is good for write but brings redundant reads to GradVec. Large value, as in traditional data parallelism, is good for read but may incur large cache miss when writing to a large size of memory region randomly.

bin level parallelism "bin" level parallelism has not to be discussed in related work. When organizing the InputMatrix 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 original data. Additional cost of memory access to the sparse data will be introduced if "bin" level parallelism is applied. However, it can provide more fine-grained parallelism when the number of features is not big enough to do model parallelism efficiently. E.g., model parallelism on HIGGS is hard to fully utilize the computation resources of a 32 cores machine because it only contains 28 features, while it will have no such issue when using "bin" level parallelism as the default size of "bin" is 256, and set bin_blk_size to 32 will work well.

node level parallelism "node" level parallelism is also not fully explored in related work. A pure node-level parallelism can not be utilized in the beginning phase of tree building where the number of leaf nodes is smaller that available CPU cores. Moreover, 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. Beyond these restrictions, node level parallelism has some special advantages. As in 3.2, thread synchronization is one of the major factors that drag performance down when using OpenMP to parallelize the for-loops within the core functions in building a tree. Set node_blk_size enables a trade-off between less number of thread synchronization and larger size of memory region of write operations. Furthermore, from the algorithm's perspective of view, those synchronizations are not required at all by the algorithm. The candidates selected for splitting can do their work independently and synchronize is only necessary when updating the tree structure and the priority queue. Therefore, node level parallelism potentially can resolve the issue of thread synchronization overhead.

4.2 TopK Growth and MixMode of Parallelism

In order to utilize node-level parallelism in leaf-wise growth method, we propose to extend the algorithm by selecting top K, rather than top 1, candidates with the largest loss change values from the priority queue, as in Algorithm. 2. In this way, a 'K' fold node-level parallelism is enabled. From one perspective of view, topK is a kind of mixture of depthwise and leafwise growth methods that enables the trade-off between robust and effectiveness. On one side, 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 side, it may build a more robust decision tree because it will mitigate the tendency of continuously splitting inside one node to form a very deep tree in case of noises in the data. By intuitions, the new algorithm can achieve similar performance of accuracy when K is not too large.



(a) depthwise

(b) leafwise

Figure 3: TopK Tree Growth Method. Example with $K = 2$ to demonstrate the extension in depthwise and leafwise methods.

mix mode A mixed mode is a solution to apply different approaches of parallelism in different phase of the tree building process. One scenario to apply a mixed mode is node-level parallelism. Pure node-level parallelism is available when the number of leaves to split is larger than the number of threads. At the beginning and the end phase of tree building, the number of leaves is too small for it. Another scenario is data parallelism. In case of a dataset with a small number of features, data parallelism is a good 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 Sec. 4.1, synchronization between nodes is not necessary for depthwise tree growth method, and it is also not mandatory in leaf-wise method if not confined by a strict topK. We refer a strict topK method as "SYNC" which always select the global topK candidates, and we refer a loosely coupled topK method as "ASYNC" in which K threads select the top candidates as best as each can. In summary, as in Table. 2, based on block-based parallelism, we have four modes that can be configured with a specific block size to fit the scenarios with different input shape and size to achieve optimal performance.

4.3 Optimization for Thread Synchronization Overhead

As in Sec 3.2, parallelizing the for-loops in the functions of tree building with OpenMP would introduce big overhead because of the ubiquitous load imbalance.

Table 2: Four mode of parallel designs for GBDT.

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

Reducing the number of for-loops is one straight-forward solution. By setting the node_blk_size to H, H selected candidates will be scheduled as a single task. In this way, we can reduce the number of for-loops from L to $\frac{L}{H}$, when comparing to the XGBoost and LightGBM implementation.

A more aggressive solution is applying ASYNC mode parallelism. Different from feature parallelism and bin parallelism, ASYNC schedules all the computation involved within one tree node as a single task in the middle phase by applying pure node parallelism, in this way, avoiding all the for-loops barrier wait overhead. Of course, splitting nodes in tree building is not a pleasing parallel process, different threads on different nodes should synchronize when accessing to the shared data structure, 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.

4.4 Optimization for Memory Access

As in Sec 3.3, GBDT training is a memory bound application. Optimizing memory access is another critical part of our work.

InputMatrix InputMatrix is organized as a cube as well by the parameter $\langle row\_blk\_size, bin\_blk\_size, feature\_blk\_size \rangle$. row_blk_size defines the number of the rows of input would be scheduled as a single task to process. It is a static partition and must be set for data parallelism. 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. Sparse data is supported by a general CSR format storage by adding offset pointers to each element. Missing value, includes zero and NAN, can be supported in this way. When the sparsity, ratio of non-missing value# over row #, is large than $\frac{1}{5}$, a dense storage format will be used to avoid unnecessary overhead of the offset pointers. A placeholder, not used bin id 255, will be put as the value.

NodeMap NodeMap maintains the mapping between node id and row id and is the bridge of how InputMatirx and GradVec would be accessed. When it keeps mapping from row id to node id, the algorithm would sequentially scan InputMatrix and random access to GradVec, which is possible in case of depthwise tree growth. When it keeps mapping from node id to row id, the algorithm accesses InputMatrix and GradVec with noncontinuous row index. The later approach is more general adopted because it supports all the tree growth methods. In our implementation, NodeMap keeps the same row partition as in InputMatrix to enable parallel operations on it.

GradVec Random memory access is inevitable in tree building because the node membership for each input data records dynamically changes when splitting happens. We adopt the node id to the row id mapping approach for NodeMap. Tree building will scan the row id set for each candidate node, fetching input data from InputMatrix and GradVec by the row id. Noncontinuous row ids lead to random memory access pattern to both InputMatrix and GradVec. Data in InputMatrix only needs to be read once, while the gradients in GradVec may need for multiple times, e.g., in feature parallelism, each thread works with a single feature column will read the same gradients from GradVec. As the gradients are always accessed in the same order of the row ids for a node, extending row id with corresponding gradients in NodeMap will remove random access to GradVec effectively, as in Fig 4.

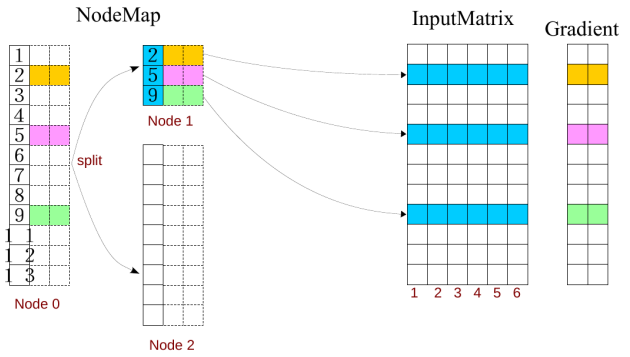


Figure 4: Extending NodeMap with GradVec. Keep a copy of GradVec in NodeMap is effective as they are accessed in the same order, otherwise accesses by row id incur random memory access.

GHSum There are one read and one write operation on GHSum for each element of InputMatrix, as in Fig 2. Because of the irregular nature of the tree building algorithm, these memory access randomly cross the 3-D matrix. Different approaches have been proposed to make a serial of these access cache friendly. First, 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, which by default is 256. Each element in GHSum is two doubles for the two summations of gradients. One feature occupies memory of size $256 \times 16 = 4K$ Bytes. Traditional data parallelism solutions process row of input as a whole, which means and a region of $4K \times M$ is involved at least. It will incur large number of cache miss when M goes large. Feature-wised parallelism has advantages in accessing GHSum. When NodeMap is set to sequentially scan InputMatrix, the bin id 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. When NodeMap maintains mapping from node id to row id, consecutive accesses happen on the same feature vector in GHSum, and the region size becomes $4K$. In both cases, the region size is small and leads to better cache efficiency.

In our block-based method, this region size can be configured, which is $16 \times bin\_blk\_size \times feature\_blk\_size$. It enables to keep balance between read and write operation efficiency by setting the size parameters corresponding to the shape of the input dataset.

4.5 Parameter Tuning

How to select the optimal parallelism solution by considering the trade-off between more parallelism and less overhead is challenging. There are also many data partition related parameters that bring huge impacts on the performance of a parallel machine learning algorithm. Although it is doable to run a grid search in the parameter space to find the optimal performance for a specific input problem. However, the result is not transferable to other datasets, and even not transferable to the same input with different configurations. At the same time, it is also hard to modeling the computation precisely because of the complexity and dynamic nature of the algorithm.

Table 3: Parameters of Input Problem

Parameter	*Description*
N	number of instances of input dataset
M	number of features of input dataset
S	sparseness of input dataset
C	coefficient of variation of feature values
D	tree size with number of leaves($L = 2^D$)
T	number of threads

Table 4: Parameters of System

Parameter	*Description*
K	number of candidates selected each time(1...32)
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)

Given a GBDT problem in Table 3, parameter tuning can find out optimal system parameters by searching the parameter space in Table 4. Here, a single parameter S is used to represent the sparseness, where $S = \frac{\#element}{N \times M}$. Besides the sparsity, the feature value distribution also matters. C is the measure of dispersion of the number of bins distribution for all the features. It is defined as the coefficient of variation (CV), $C = \frac{stdev}{mean}$, the larger this number, the more uneven of the distribution. For example, HIGGS has 28 features, in which 25 are real values that fill all the bins, and 3 of them only contains 3 unique values that occupy only 3 bins each. This uneven distribution leads to workload imbalance in feature-wised parallelism.

By parameter tuning experiments on a group of input problems with synthetic datasets, we can explore the answers to the following questions. How to select the mode of parallelism? How to set the block size? The rules we learned can then be the guideline to optimize the system parameters for real problems.

5. EXPERIMENTS

5.1 Setup of Experiments

GBDT Implementation. XGBoost, LightGBM are two systems for comparison, using the latest commit in Jan 2019. XGBoost is a full-featured GBDT implementation that widely used. It supports both depthwise and leaf-wise tree growth. It supports different tree-building methods that include data parallelism and feature-wised parallelism. LightGBM adopts a different histogram building algorithm and supports leafwise tree growth and featurewise parallelism. Our HarpGBDT is based on the XGBoost code base, reusing the code of histogram building algorithm and focusing on extending to support block-based parallelism. Other than write from scratch, this strategy enables us to fast prototype and do a precise performance evaluation on the extended features by controlled experiments. All the systems are implemented by C++ and support multi-threading by OpenMP; all support distributed computation by MPI or a socket communication layer.

Table 5: Dataset. N is # instances. M refers to #features. S refers to sparsity and C is CV of # bins distribution.

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

Datasets. Four datasets (see Table. 5) are used in the experiments. HIGGS[3] and AIRLINE[1] are two standard dataset widely used in GBDT benchmarks. HIGGS is real value dataset with only 7.8% zero values. AIRLINE mainly contains categorical features. After label-encoding or feature bundling on one-hot encoding, the dataset contains 8 real value features, and all records with missing value removed. CRITEO[2] contains 13 integer features and 26 categorical features for 24 days of click logs. By response value replace encoding, we first calculate the clickthrough rate (CTR) and count for these 26 categorical features from the first ten days. We are then replacing them on the next ten days' data. The whole training data have 1.7 Billion records, where a subset of 50 Million records is select for experiments on a single node. Records with missing value are all preserved. YFCC100M[5] is a large image dataset with 100 Million images in total. [18] trained deep learning models to learn 1,570 visual concept on this dataset. We combined the visual concept categories to its top-level hierarchy and selected "artifact" to make a binary classification dataset. Features for each image record is a 4K extracted from a VGG model. We use a subset of 1 Million records for experiments on a single node. SYNSET is a synthetic dataset that the feature values are randomly generated 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. We use SYNSET for parameter tuning experiments in order to explore the pros and cons of different parallelism mode and block size configurations.

These datasets come from different applications and have quite a different shape and size, and they are diverse and representative for our thorough experimentation and evaluation.

Hardware and Software Configure. In regards to hardware configuration, all experiments are conducted on a 128-node Intel Haswell cluster at Indiana University. Among them, 32 nodes each have two 18-core Xeon E5-2699 v3 processors (36 cores in total), and 96 nodes each have two 12-core Xeon E5-2670 v3 processors (24 cores in total). All the nodes have 128 GB memory and are connected by QDR InfiniBand. As for the software configuration, all C++ trainers are compiled

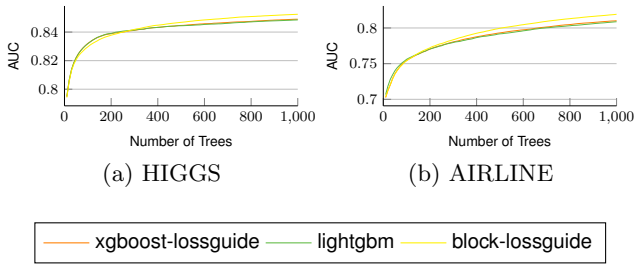
with gcc 4.9.2 and -O3 compilation optimization.

Algorithm and Evaluation Parameters. We fixed the training related parameters as: $learning\_rate = 0.1$, $min\_child\_weight = 1$(minimum sum of instance weight needed in a child), $gamma = 1.0$(minimum loss reduction required to make a further partition), and logistic regression loss for all the binary classification tasks.

Performance evaluation focus on training time, which is the wall clock elapse execution time that excludes the time on data loading and one-time initialization. We use the average training time per tree as the main metric. AUC(Area Under The Curve) is used to evaluate the accuracy of the learned model.

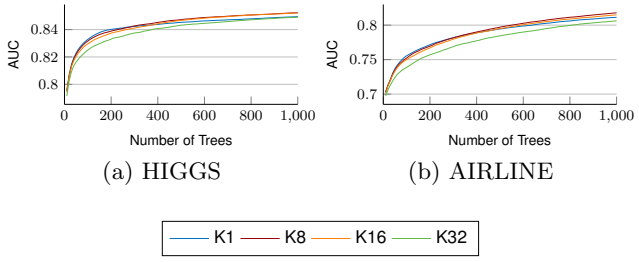
5.2 Experimental Results

5.2.1 Convergence of TopK Tree Growth Method



(a) HIGGS (b) AIRLINE

Figure 5: Comparison of Convergence Rate. $D = 8$, $K = 8$.



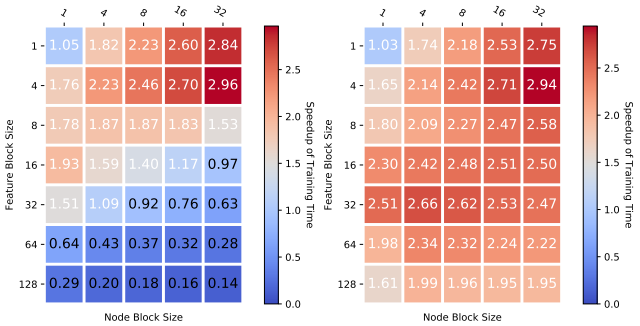
(a) HIGGS (b) AIRLINE

Figure 6: Influences of K on Convergence Rate. $D = 8$, **ASYNC mode.**

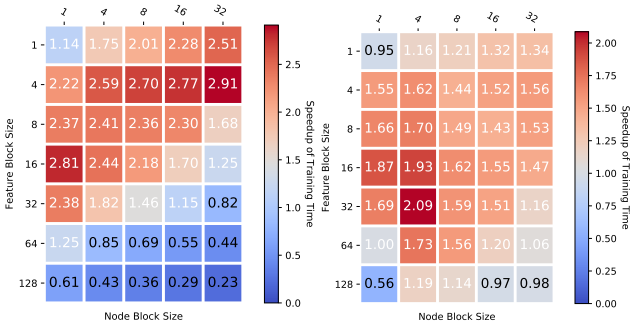
TopK growth method, which selects top K, rather than top 1, candidates with the largest loss change values from the priority queue will create a different decision tree. This set of experiments show that this extension does not sacrifice accuracy. Under the depthwise method, the TopK method achieves exactly the same convergence rate. Under the leaf-wise method, the TopK method starts from a lower accuracy but soon catch up and even get better accuracy on the experimental datasets, as in Fig. 5. When adjusting the parameter K, as in Fig 6, we found that the accuracy is quite robust for a large range of K. For a tree with only 256 nodes, accuracy under $K = 16$ can catch up very fast and exceed the original method with $K = 1$; $K = 32$ shows a larger gap and catch up slower. These results come from experiments running in ASYNC parallelism mode, in which each node split as fast as they can. Other results, omitted here because of page limitation, show that large K works even better in SYNC mode and larger trees. It is reasonable that the original top1 method converges faster in the beginning because it applies the split with the largest gain. When the tree grows, TopK method potentially provides a more stable model by avoiding to keep splitting within one branch of the tree to form a deep tree, which mitigates over-fitting to noise in the input data.

5.2.2 Parameter Tuning

We start parameter tuning experiments from a fixed input with thread number $T = 32$ on the SYNSET dataset. For the system parameters in Table 4, we set $K = 32$, as increasing K always improves performance but too large can degrade convergence rate; $row\_blk\_size = \frac{N}{T}$ to enable data parallelism to fully utilize the cpu cores; $bin\_blk\_size = 256$ to disable blocks along the bin dimension. By running a grid search on the parameters space of the feature_blk_size and node_blk_size, we investigate the relationship between performance and these block size configurations. Standard data parallelism is $feature\_blk\_size = 128, K = 1$ and standard model parallelism is $feature\_blk\_size = 1, K = 1$. We use a normalized score as a measure, which is the ratio of average training time per tree of the test system over that of standard model parallelism.



(a) MP D8 (b) DP D8



(c) MP D12 (d) DP D12

Figure 7: Speedup over Standard Model Parallelism. Leaf-wise Growth. MP refers to model parallelism, DP refers to data parallelism. Tree size denoted as D# .

From the experimental results showed in Fig 7, we have following observations. First, a maximum of 2.94× to 2.96×

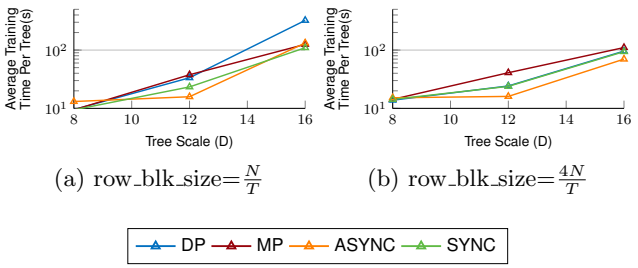
speedup is observed for DP and MP. It means that a significant performance gain can be achieved by adjusting the block size parameters only. Second, when comparing DP method and MP method, DP is more robust corresponding to the block size parameters that it gives better performance than standard baseline in most cases while MP has more limitations that it can not fully utilize CPU cores when the block number is less than the number of thread. Thirdly, both DP and MP prefer a medium size of feature block. The first columns of the figure, where node_blk_size set to 1, show that the medium size of the feature block gets the best performance. It justifies our presumption that there is a trade-off between read and write operations in GBDT, as read operation prefers large feature blocks while write operation prefers small ones in order to confine writing small smaller memory region to avoid a large number of cache miss. Fourthly, the influences of node_blk_size is similar. In MP, when the feature block size is small enough to provide enough blocks to the scheduler, larger of node_blk_size boost 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 definitely degrades the performance. The best configurations for MP are along the secondary diagonal. In DP, when the feature block size is small enough that write operations still have low cache miss rate, larger of node_blk_size boost 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.

Table 6: VTune Profiling on SYNSET with D8 K=32.

	Standard MP			Standard DP		
NodeBlkSize	1	8	32	1	8	32
CPUTime	817.19	372.10	283.50	481.46	382.03	377.59
SpinTime	10.14	11.54	12.80	185.41	46.14	21.90
SpinTime-Ratio	0.01	0.03	0.05	0.39	0.12	0.06
BuildHist	771.20	322.48	233.62	239.29	266.95	275.59
BuildHist-Ratio	0.94	0.87	0.82	0.50	0.70	0.73
AverageCpu	30.71	28.76	27.55	16.63	22.11	24.98

Table 6 shows results of VTune profiling on standard DP and MP modes when adjusting node_blk_size. Both DP and MP runs faster with larger node_blk_size, but caused by different reasons. SpinTime, the OpenMP barrier wait time, is the major performance issue in DP. Node_blk_size in DP controls the memory consumption of model replicas and the number of reduce operations is inversely proportional to it. In MP, node_blk_size changes the granularity of tasks rather than the number of thread synchronizations. Therefore, a stable SpinTime is observed in MP, and smaller tasks get better average CPU utilization but larger tasks get better performance due to better memory access efficiency.

Fig. 7 also shows a trend for DP and MP when the size of single tree increases. DP works better than MP in D8 in most cases, and MP exceeds when training on larger tree D12. To investigate the capability of different parallelism mode, we run a group of experiments on SYNSET with three tree scale parameters $D = \langle 8, 12, 16 \rangle$. DP and MP set their optimal block size parameters according to the result of the previous grid search. Parameter $\langle feature\_blk\_size, node\_blk\_size \rangle$, both set to $\langle 4, 32 \rangle$, NP set to $\langle 4, 32 \rangle$ at D8, $\langle 32, 4 \rangle$ at D12. ASYNC mode uses the same block configuration as DP. Result in Fig. 8 shows that DP works the best at D8 and



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

Figure 8: Trend of performance over Tree Size. Leafwise Growth. MP refers to model parallelism, DP refers to data parallelism, SYNC refers to DP-MP mixmode, ASYNC refers to DP-PureNodeParallelism mixmode. Logarithm scale on Y-axis.

its performance degrades when tree goes larger. The major overhead of DP after the block size optimization lies on the reduce operation for the model replicas. The number of this operation grows proportional to the number of tree nodes. MP get similar performance as DP at D8, but it scales better than DP over the tree size. This also demonstrates the advantage of model parallelism that supporting model updates without conflicts not only leads to a smaller memory footprint but also enable to scale. SYNC mode always achieves the better performance than DP and MP, which shows that the mix mode is effective. ASYNC shows the best capability of scaling over the tree size, because it enables MP to use larger feature block size by adding node-level parallelism beside reducing thread synchronization overhead.

In Fig. 8 (a), the performance of the DP related modes go worse from D12 to D16 than MP related ones, DP and ASYNC have a sharp degradation. Parameter, row_blk_size, is responsible for it. D16 is an extreme case that each single decision tree splits to 65536 leaves, i.e., each node contains about $\frac{N}{2^16} = 152$ data instances at the end. The previous setting of $row\_blk\_size = \frac{N}{T}$ create 32 row partitions on SYNSET, and this makes a large number of the very small row set. The task for each partition is too small that parallel efficiency goes down. For ASYNC, the synchronization overhead of updating shared data structure then becomes an issue when it updates at a very high frequency. By adjusting this parameter to a larger value, an obvious improvement is found in Fig. 8 (b), DP and ASYNC boost up about 50%. MP and SYNC does not gain much because they adopt big node_blk_size and are less influenced by this synchronization problem caused by small tasks.

The findings based on SYNSET helps to understand pros and cons of different parallelism mode. They give a useful guideline to help to set parameters with proper values. SYNSET is limited as it does not represent the dynamic work load scenarios which we will explore further on real datasets in the next section.

5.2.3 *Shared Memory Parallel Efficiency*

In this section, we run experiments on HIGGS to investigate the parallel efficiency of block-based implementation in shared memory system. By the guideline learned from parameter tuning, we adopt DP mode for D8 and ASYNC mode for larger trees, run a group of experiments with the

parameters $K = 32, T = 32$, and $\langle feature\_blk\_size, node\_blk\_size \rangle$ set to $\langle 4, 32 \rangle$ in DP and in ASYNC.

We mainly use two metrics to evaluate the relative performance of the GBDT trainers. The first is the speedup over average training time per tree. HarpGBT is built upon codebase of XGBOOST and inherits all its parameters so that they can be compared exactly by setting the same parameters, in which they do the same amount of computation and achieve exactly the same convergence rate (with $K = 1$), as shown in Fig. 5. LightGBM has its own optimizations that does not do exactly the same computation and leads to a different convergence rate. In the beginning, it convergences faster than the other two but the rate slows down later. GBDT training is irregular that the workload of computation on consecutive trees are different, gradually shrinks. HarpGBT and XGBOOST keep the same pace of this shrinkage (with $K = 1$), while LightGBM accelerates more. In order to fairly compare the performance, we first evaluate their efficiency when they do the same or similar amount of workload. We use the average training time per tree for the first 100 trees as this metric. Secondly, we evaluate the overall performance corresponding to the convergence with the metric of convergence speed, i.e., the training time to achieve the same accuracy.

Table 7: VTune Profiling on HarpGBT

Trainer	Depth-DP	Leaf-DP	Leaf-ASYNC
AverageCPUUtilization	27.5	28.5	28
OpenMPBarrierOverhead	0.09	0.08	0.08
Average Latency	15	16	15
Memory Bound	38	41	40

Comparing with Table 1, Table 7, show that the OpenMP barrier overhead is significantly reduced in HarpGBT. DP 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, the ratio drops to 0.02 at D12. Memory-related metrics also improve due to the better block size configurations.

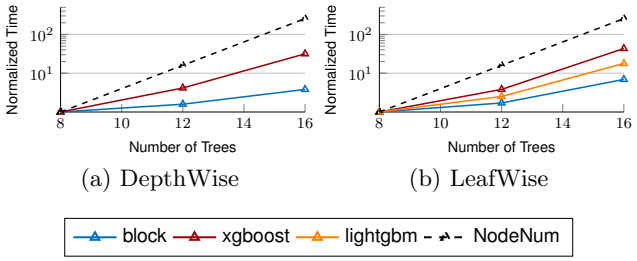


Figure 9: Trend of Training Time over the tree depth. Normalized execution time of each iteration.

Fig. 9 shows that HarpGBT scales better over the tree size than the other implementations. The optimizations that target on the overheads we discussed in section 3.1 are proved to be effective.

Fig. 10 shows the convergence speed comparison. In D8, although LightGBM is about 2 × slower thatn HarpGBT in the beginning, it finish the 1000 trees building with similar time. This shows that LigthGBM is optimized better in the code beside BuildHist and is very efficient in

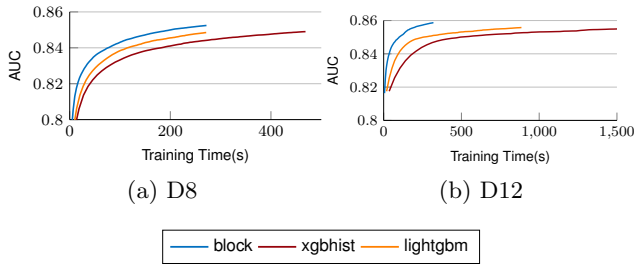


Figure 10: Convergence Speed. Train with leafwise growth method to 1000 trees.

cases of small workload. But LightGBM's model accuracy is lower than that of HarpGBT. When setting tree size to D12, HarpGBT shows a strong advantage over the other two. It converges much faster and also finishes the job much faster. This boosting of performance should be credited to the optimizations that enable it to scales better than the others. XGBOOST is the slowest implementation. The accuracy achieves in the end, and the time it spends to finish the job is used as the reference. Convergence Speedup is defined as the ratio of the reference time over the time spend to reach the reference accuracy value. HarpGBT achieves 1.57× and 4.49× convergence speedup over LightGBM, 2.57× and 9.63× over XGBOOST on D8 and D12 correspondingly.

5.2.4 *Overall Performance*

In this section, we run a performance evaluation on four datasets with different characteristics and comparing the overall relative performance by training time speedup and convergence speedup.

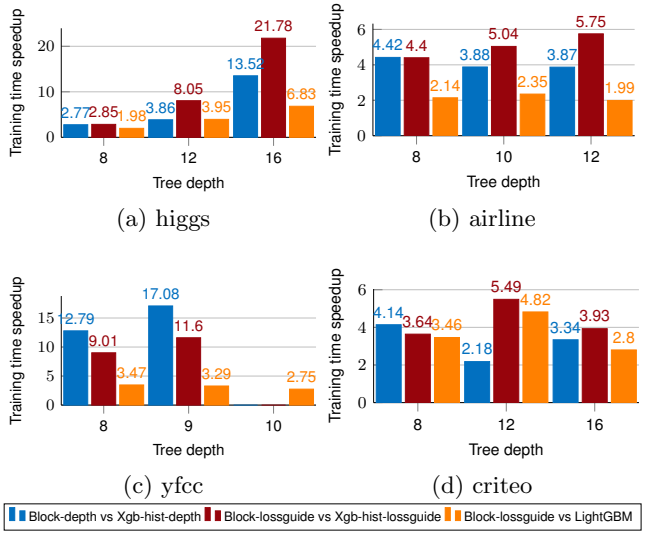


Figure 11: Training Time Speedup on four datasets.

YFCC and AIRLINE are two types of input with a very different shape, the first one is a fat matrix with many feature columns and the second one is a thin matrix with a large number of rows. For fat matrix input YFCC, as we

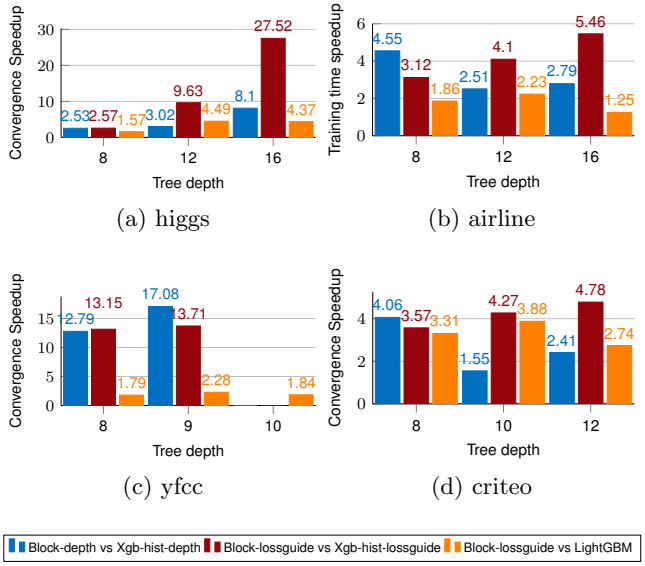


Figure 12: Convergence Speedup on four datasets.

have discussed in section 5.2.2, standard DP does not work well due to inefficient write operations, standard MP is also inefficient due to the overhead of redundant read operations. HarpGBT shows more than 10 × speedup over XGBoost in this scenario, and more than 2.75 × training time speedup and 1.80 × convergence speedup over LightGBM.

For thin matrix input AIRLINE, DP is a better choice than standard feature-wised MP which cannot fully utilize the CPU cores. HarpGBT adopts DP in small tree D8 and change to ASYNC on D12 and D16, it achieves more than 4 × training time speedup and 3 × convergence speedup over XGBoost. HarpGBT keeps around 4 × faster than LightGBM, but the convergence speedup is less than 2 ×. Overfitting is observed in large tree, it may root from the quality of input data which contains enough records,100 million, but spans 10 years.

Criteo is a huge dataset coming from the advertisement industry. We sampled 50M records from one day as the training dataset, 1M records from 15 days later as a test dataset. We set $min\_child\_weight = 100$ to control over-fitting, but it is still hard to learn a robust model by leafwise method. It was found that leafwise method builds a deep tree with depth more than 150 on criteo in case of large tree size. One possible reason is the encoding method of this dataset, response variable replacement would generate dataset with the feature values highly correlated with the response variable. There is higher probability for the algorithm to keep splitting within one branch of the tree. In this case, HarpGBT achieves an average 4.02 × training time speedup and 3.45 × convergence speedup over XGBoost, 3.69 × training time speedup and 3.31 × convergence speedup over LightGBM.

On average across the four datasets, HarpGBT achieves 7 × faster than XGBoost and 3.3 × faster in training time and 2.6 × faster in convergence speed than LightGBM.

6. RELATED WORK

Kaggle's survey in 2017[4] shows that 24% of the data mining and machine learning practitioners are users of Gradient Boosted Machines. Important applications for GBDT include: Higgs boson classification[7], credit scoring[21], computer aided diagnosis[16], insurance loss cost modeling[11], freeway travel time prediction[22], etc. Among them, Mizuho Nishio et al. used a GBDT with a maximum depth of 13 to train a computer-aided diagnosis system. XiaYufei et al. trained a credit scoring system on 10,000 samples with 24 features, and the maximum depth they used was 12.

The frameworks that implemented GBDT include: XGBoost[6], pGBRT[20], scikit-learn[17], GBM in R, LightGBM[13], and CatBoost[8]. Scikit-learn and gbm in R implement the pre-Sorted algorithm, and pGBRT implements the histogram-based algorithm. XGBoost supports both the pre-sorted algorithm and histogram-based algorithm.

7. CONCLUSIONS AND FUTURE WORK

GBDT is a widely used machine learning algorithm that successes in many applications. There have been many efforts to optimize this algorithm on different aspects, including dealing with sparse and categorical features, accelerating by GPU devices and optimizing communication in distributed implementations, etc. However, there is no existing work that clearly investigates the system optimizations on many-core architecture to speedup the performance.

In this paper, we start with analyzing the bottleneck on two state-of-the-art GBDT implementations, which are the two representatives of popular parallelism patterns adopted by implementations. Memory bound and large overhead of thread synchronizations caused by barrier wait under OpenMP for-loop parallelism are observed. We propose a block-based parallelism strategy to fully utilize the potential unit of parallelism in the GBDT algorithm. Input data and model data are organized as blocks in 3D matrices. Then data parallelism and model parallelism can be applied based on these blocks. By adjusting the block size, performance related to memory access can be tuned. By selecting different parallelism method according to the shape of the input matrix and during the process of tree growth, the overhead of thread synchronization can be reduced largely. To select the best parameters in our design to achieve the optimal performance when given a dataset is a challenging problem. As the GBDT algorithm is irregular, it is very difficult to know before hand the tree growth progresses on the dataset. We investigate parameter tuning by grid search on a synthetic dataset and use the learned rules as a basic guideline for parameter settings. We find that, 1) Block-based data parallelism works well for small trees in general. However, the static overhead of reducing of model replicas makes it a second choice for large trees. 2) Block-based feature parallelism provides stable performance on different tree size. It is limited for the input with only a few features. 3) Node parallelism should be preferred to reduce thread synchronization overheads, especially in the mixed mode in which the scheduler switches to the pure node parallelism method(ASYNC) when the tree grows large enough. Performance evaluations on four open datasets with quite a different shape and characteristics show that our block-based implementation exceeds the other two state-of-the-art. Our optimization has achieved a speedup up to 21×.

As a memory bound application, further reducing the performance bottleneck at memory access is interesting. Sampling methods are one promising direction we would like to explore in future work. Another interesting direction could

be applying the proposed method on real applications which needs the capability of big data processing.

8. ACKNOWLEDGMENTS

We gratefully acknowledge support from the Intel Parallel Computing Center (IPCC) Grant, NSF 1443054 CIF21 DIBBs 1443054 Grant, NSF OCI 1149432 CAREER Grant and Indiana University Precision Health Initiative. We also appreciate the system support offered by FutureSystems.

9. REFERENCES

[1] Airline on-time performance. http://stat-computing.org/dataexpo/2009/.

[2] Criteolabs data. http://labs.criteo.com/2013/12/download-terabyte-click-logs.

[3] Higgs data set. https://archive.ics.uci.edu/ml/datasets/HIGGS.

[4] Kaggle 2017 survey results. https://www.kaggle.com/amberthomas/kaggle-2017-survey-results.

[5] Yfcc100m. http://multimediacommons.org/.

[6] 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.

[7] 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.

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

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[18] A. Popescu, E. Spyromitros-Xioufis, S. Papadopoulos, H. Le Borgne, and I. Kompatsiaris. Toward an Automatic Evaluation of Retrieval Performance with Large Scale Image Collections. In *Proceedings of the 2015 Workshop on Community-Organized Multimodal Mining: Opportunities for Novel Solutions*, MMCommons '15, pages 7–12, New York, NY, USA, 2015. ACM.

[19] 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.

[20] 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.

[21] 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.

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

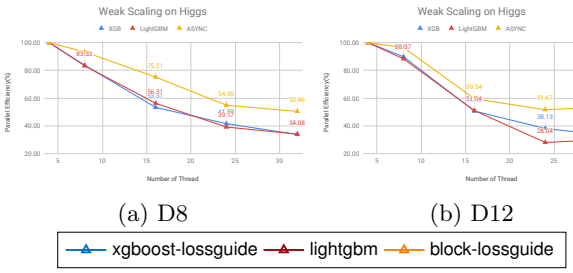
10. APPENDIX

Algorithm 3: Gradient Boosted Tree

input : dataset $D = (x_i, y_i)_{i=1}^n$, parameter λ, α, m
output: m trees $f(x) = w_q(x)$

begin
  Initialize()
  for *t=1 to m* **do**
    `// BuildTree(`$\{(x_i, y_i, g_i, h_i)\}$`)`
    $(w, q) =$ $\arg\min_{f_t} \sum_{i=1}^n [\ell(\widehat{y}_i^{(t-1)} + \alpha f_t(x_i), y_i)] + \Omega(f_t)$
    `// additive update`
    $f_t(x) = f_{t-1}(x) + \alpha f_t(x)$
    `// update the gradients`
    $g_i = \partial_{\widehat{y}_i^{(t-1)}} \ell(\widehat{y}_i^{(t-1)}, y_i)$ $h_i = \partial^2_{\widehat{y}_i^{(t-1)}} \ell(\widehat{y}_i^{(t-1)}, y_i)$

Algo. 3 is the pseudo code for GBDT training. As an additive model, building the $t-th$ tree depends on the predictions based on the previous $1..t-1$ trees. This is a strict



(a) D8 (b) D12

xgboost-lossguide lightgbm block-lossguide

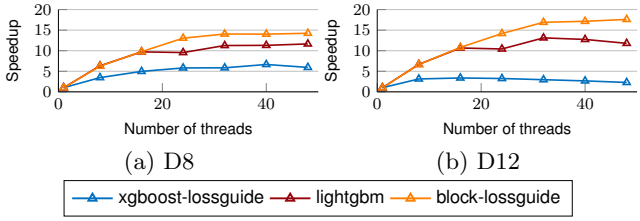
Figure 14: Weak Scaling of Training Time on HIGGS. Leafwise tree growth.

Algorithm 4: FindSplit(): Greedy Split Finding

input : I, instance set of current node; d, feature dimension
output: the feature position to split on with max score

1 **begin**
2 $score = 0$
3 $G = \sum_{i \in I} g_i$
4 $H = \sum_{i \in I} h_i$
5 **for** $k = 1$ *to* d **do**
6 $G_L = 0; H_L = 0$
7 **for** j *in sorted(I, by* X_{jk}*)* **do**
8 $G_L = G_L + g_j; H_L = H_L + h_j$
9 $G_R = G - G_L; H_R = H - H_L$
10 $score = max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda}) - \frac{(G_{I_L}+G_{I_R})^2}{H_{I_L}+H_{I_R}+\lambda}$

sequential procedure, for loop in line(3), and cannot be parallelized. Algo. 1 shows a general single tree building framework. There are two popular tree growth methods; one is "depthwise" that splits node level by level, and the other is "lossguide" that select the node with the largest changes in loss function to split. In Algo. 1, these two growth methods are unified by a priority queue which can support specific growth policy by specific comparison functions. BulidHist, FindSplit, and ApplySplit are three core functions in Algo. 1. With splitting on one node as an example, BuildHist first goes through all the instances that belong to this node, build a gradient histogram that summaries the gradient statistics, as in Algo. 2. FindSplit enumerates all split candidates to find the position with the largest split score, as in Algo. 4. Finally, ApplySplit applies it to the tree and split the node into two children.



(a) D8 (b) D12

xgboost-lossguide lightgbm block-lossguide

Figure 13: Strong Scaling of XGBBlock on Training Time. Leafwise tree growth method. D8 in DP mode, D12 in ASYNC mode.

Fig. 13 shows that HarpGBT scales better over the thread number than the other implementations. Due to the low computation over memory ratio, irregular and random memory access pattern in GBDT, all the implementations do not achieve a good strong scaling performance. HarpGBT 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. The parallel efficiency is $\frac{T_1}{T_N})100\%$. As in Fig 14, ASYNC shows significant better parallel efficiency than DP and MP system.