Parallel and Distributed GBT

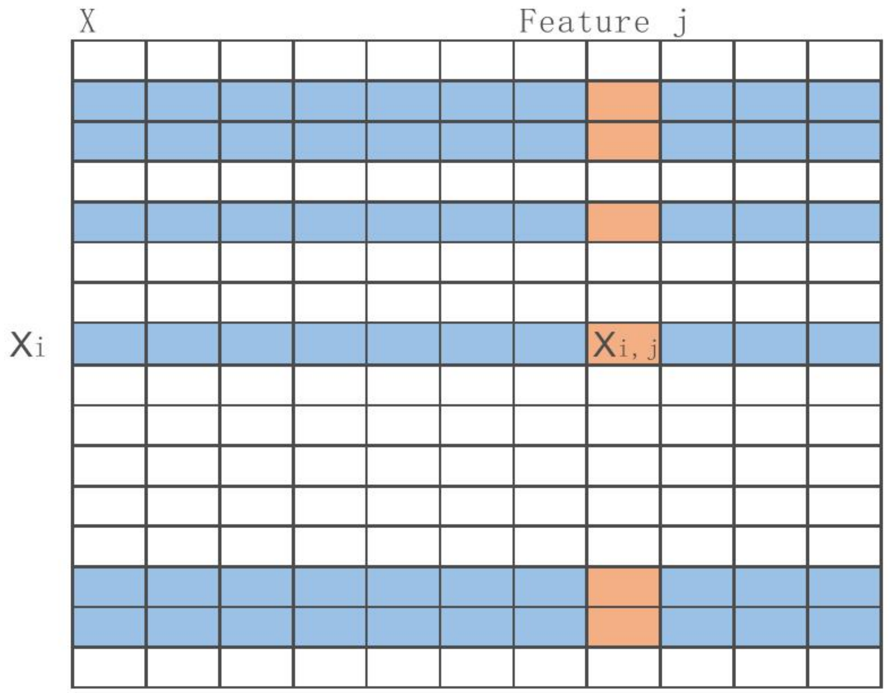
9/18/2018

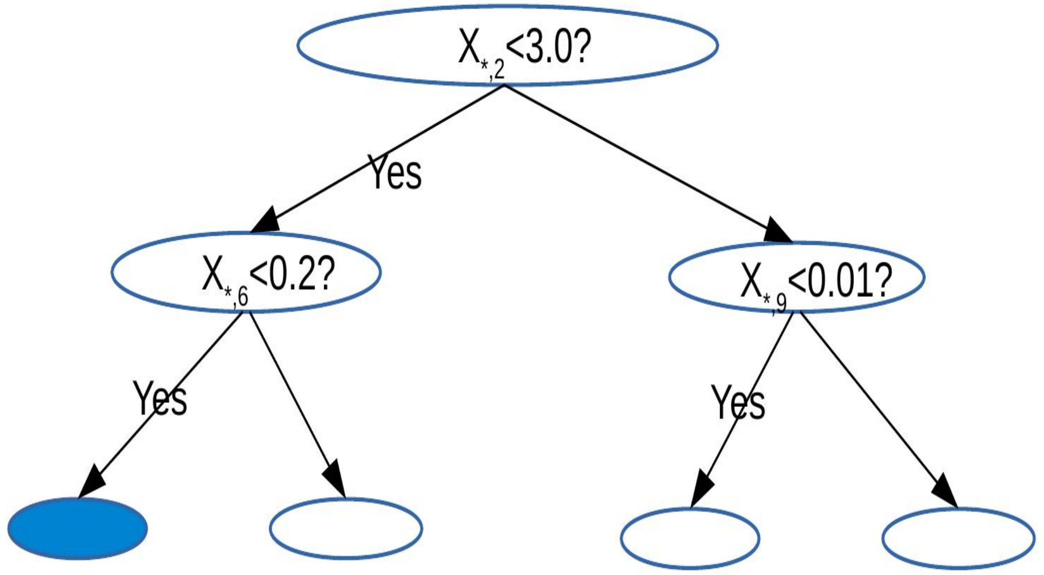
Bo Peng
IPCC@Indiana University

Outline

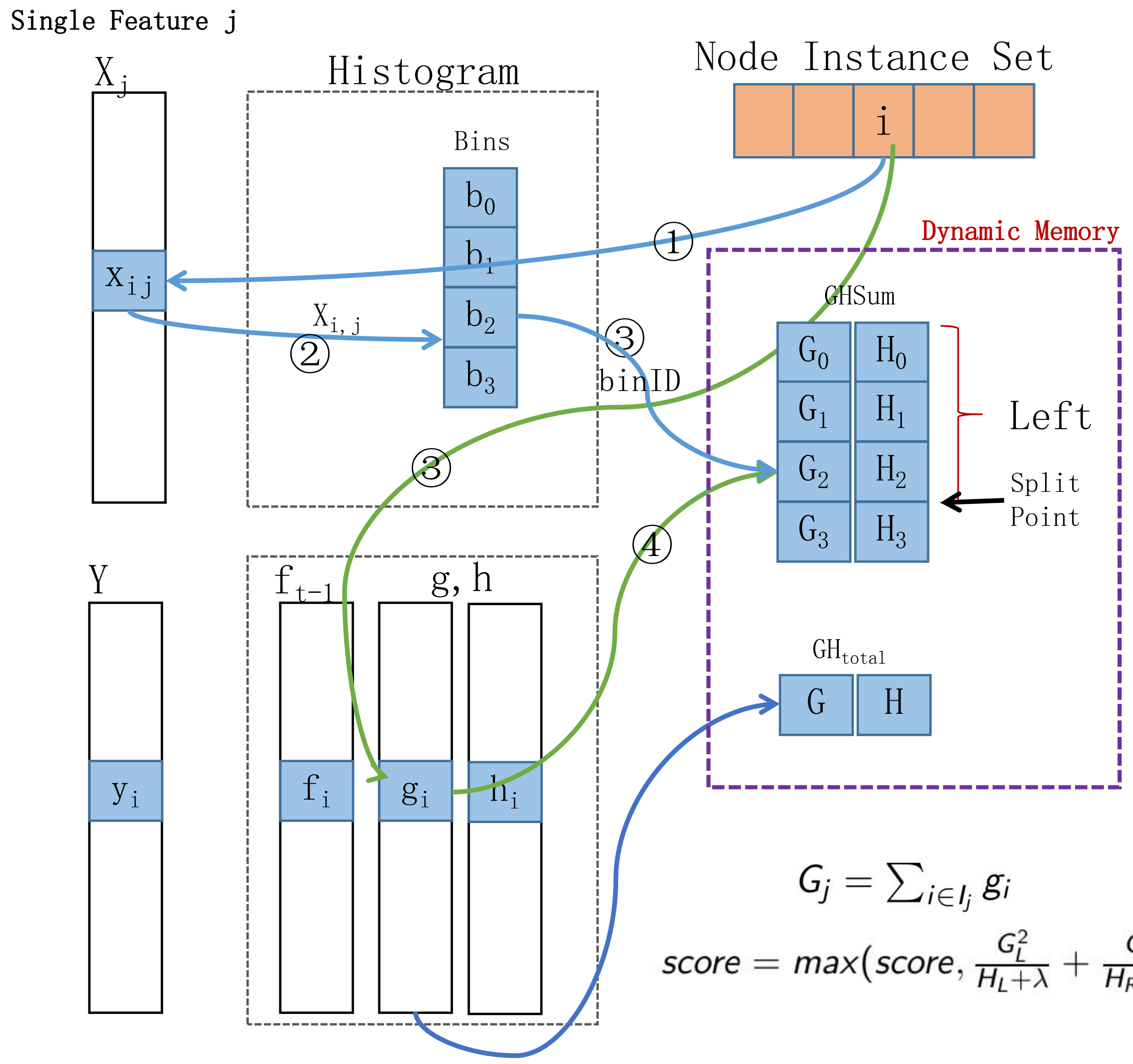
- Problem
- Related work
- Proposal

Building a Tree





- findBestSplit is the bottleneck
  - For each node, go through all instances in the node and all features
  - parallelism: node level, feature level, inside single feature level

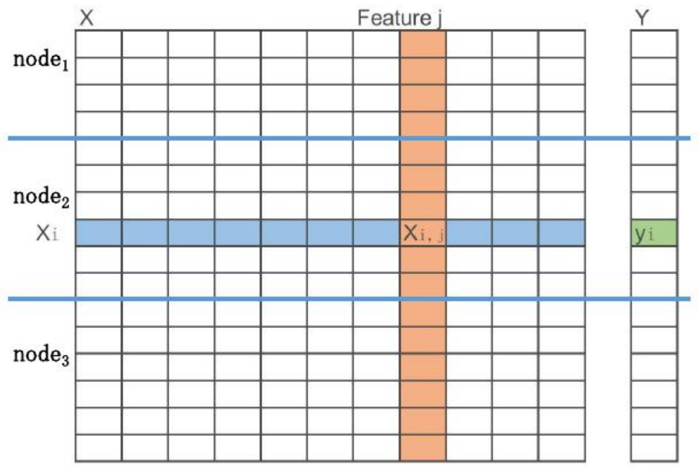


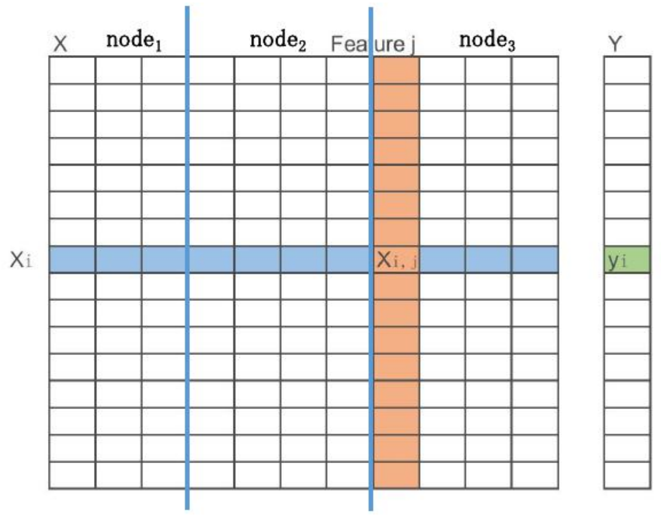
$$G_j = \sum_{i \in I_j} g_i$$

$$score = max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda})$$

- findBestSplit is the bottleneck
  - GHSum is the core data structure
    - for each feature
    - bins, fixed split points
    - g,h summation on instances whose value fall into the bin

Distributed findBestSplit





- Partition by rows (instances)
  - need gobal communication to build Bins, (build once if use global static Bins)
  - need gobal communication to build GHSum for each feature in findBestSplit() → allreduce(GHSum)
- Partition by columns (features)
  - Bins and GHSum are all local, no communication
  - need global communication to select the best feature in findBestSplit() → allreduce(maxscore), also need to broadcast(Split Instance Set)

Issues

- irregular memory access
  - instances in one tree node are dynamic set
  - non-continuous memory access to g,h
  - read/write after write dependency, GH[bin[X[i,j]]] += g[i], will stall when cache miss on access g[i]
- spasity
  - missing values
  - frequent zero entries
  - artifacts of feature engineering such as one-hot-encoding
- high dimensionality
  - |features| can be very large (10~50M)

Outline

- Problem
- Related work
- Proposal

XGBoost[1]

- Standard baseline
- Algorithm
  - approximate algorithm with both local and global proposal methods (Bins)*
  - sparsity-aware, only collect statistics of non-missing entries
- System
  - Column Block
    - each block is a subset of rows
    - in each block, data stored in compressed column(CSC) format, with each column sorted by corresponding feature value
    - support feature level parallelism
    - linear scan to find best split
  - Cache-aware access
    - choose correct block size to get gradient statistics(g,h) fit into the CPU cache. (2^{16})

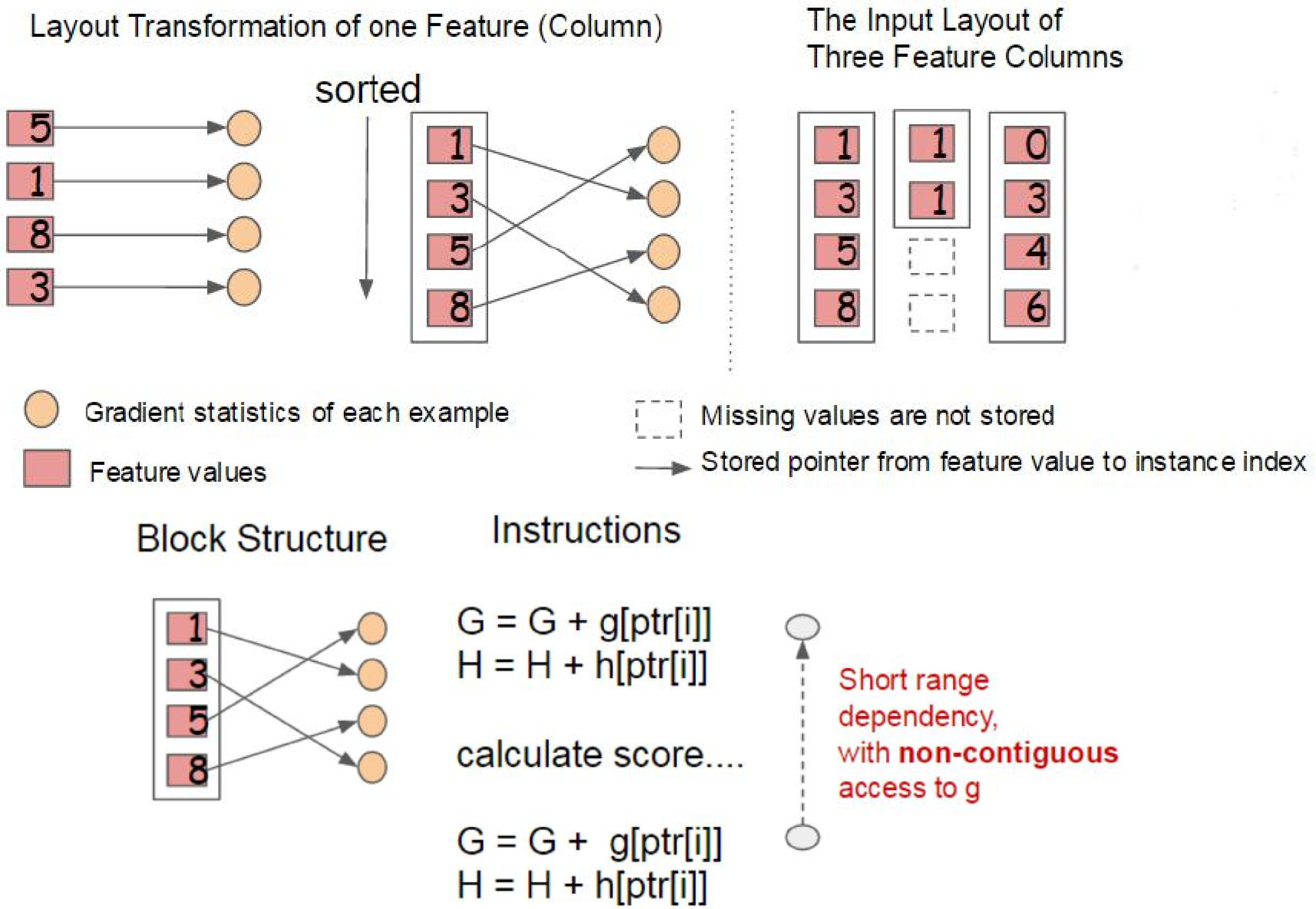

Layout Transformation of one Feature (Column)
sorted
5
1
8
3
1
3
5
8
The Input Layout of Three Feature Columns
1
3
5
8
1
1
0
3
4
6
Gradient statistics of each example
Feature values
Missing values are not stored
Stored pointer from feature value to instance index
Block Structure
Instructions
1
3
5
8
G = G + g[ptr[i]]
H = H + h[ptr[i]]
calculate score....
G = G + g[ptr[i]]
H = H + h[ptr[i]]
Short range dependency, with non-contiguous access to g

Figure 8: Short range data dependency pattern that can cause stall due to cache miss.

LightGBM[2][3]

- high communication costs ~O(|features|*binSize)
  - PV-Tree (Parallel Voting Decision Tree)
    - local voting: select top-k features based on local data
    - global voting: select top-2k features by votings from local candidates
    - collect full-grained histograms of the globally top-2k features, and findBestSplit
- high dimensionality
  - Gradient-based One-side sampling
    - exclude a significant proportion of data instances with small gradients in estimate the score
    - better than SGB(Stochastic GB) with the same sampling ratio
- Sparsity
  - Exclusive feature bundling
    - bundle mutually exclusive features(never nonzero values simultaneously) into a single feaure
    - letting exclusive features reside in different bins (adding offsets to original values)

DimBoost[4][5]

- high dimensionality (industry application with 330K features)
- Optimize Communication
  - parameter server
    - claims to be one communication step and take less time (?)
  - two-phase-split finding
    - server-side split
  - round-robin task scheduler
    - schedule the splitting tasks among the workers
  - low-precision gradient histogram
    - each item q in a histogram, encode to a d-bit integer $q'=\text{floor}(q/|c|*2^d)$
    - d=8 often enough to obtain no loss on final accuracy
- Optimize Computation
  - parallel batch construction
    - divides a range into batches for the big nodes in the first few layers
  - spare histogram construction
    - only non-zero entries

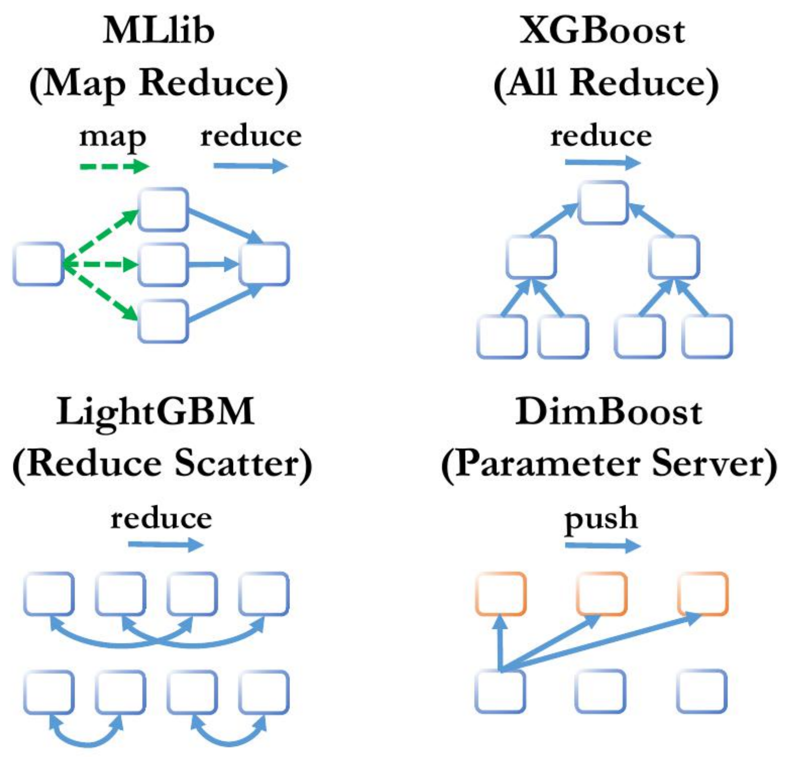


Figure 3: Existing Model Aggregation Methods.

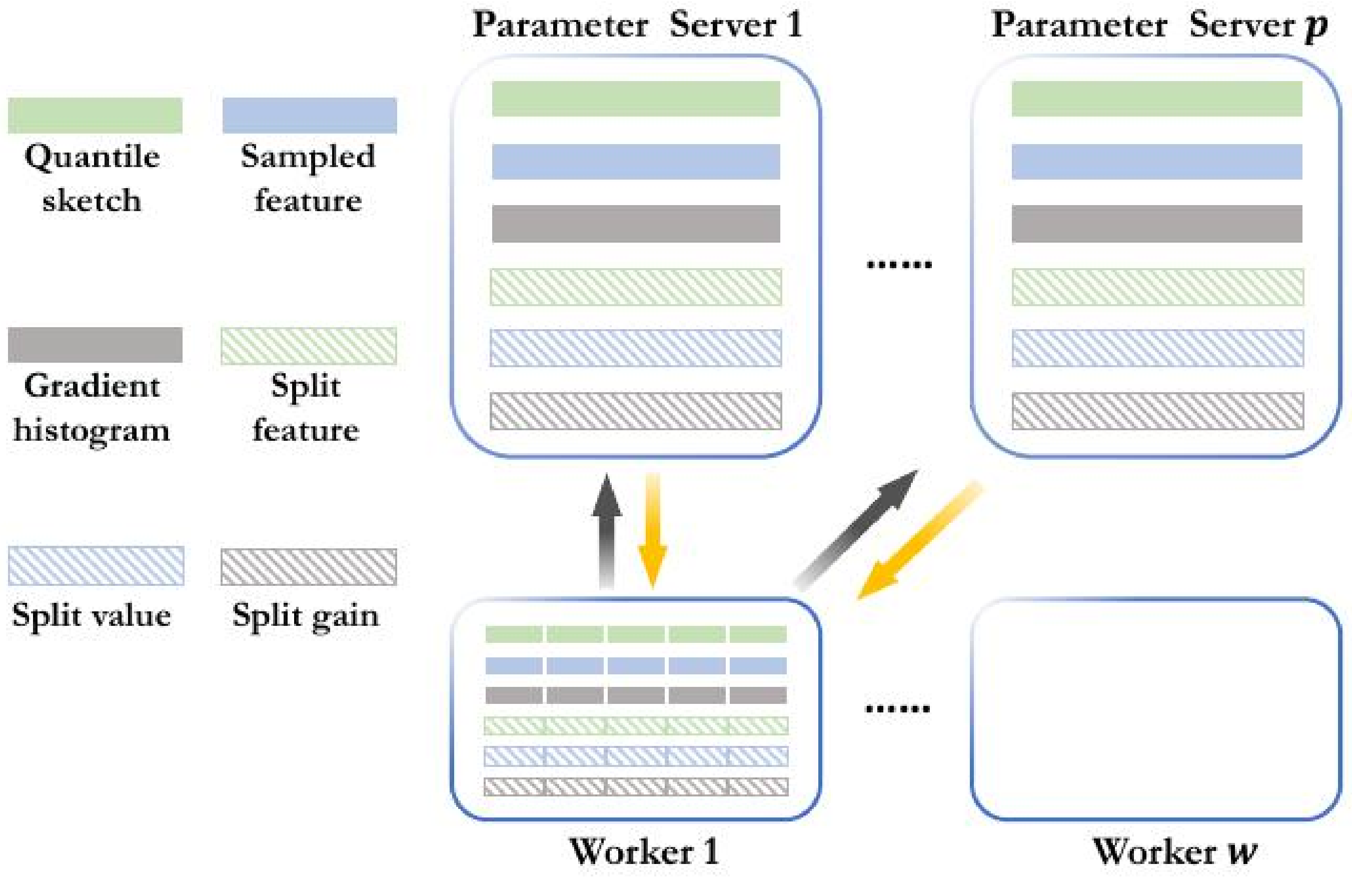
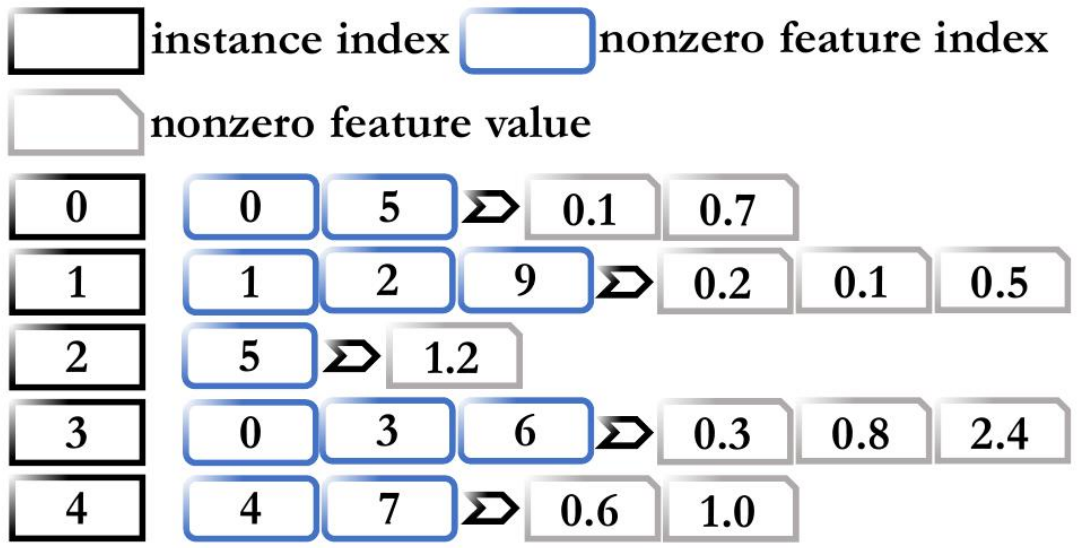


Figure 6: Parameter layout.



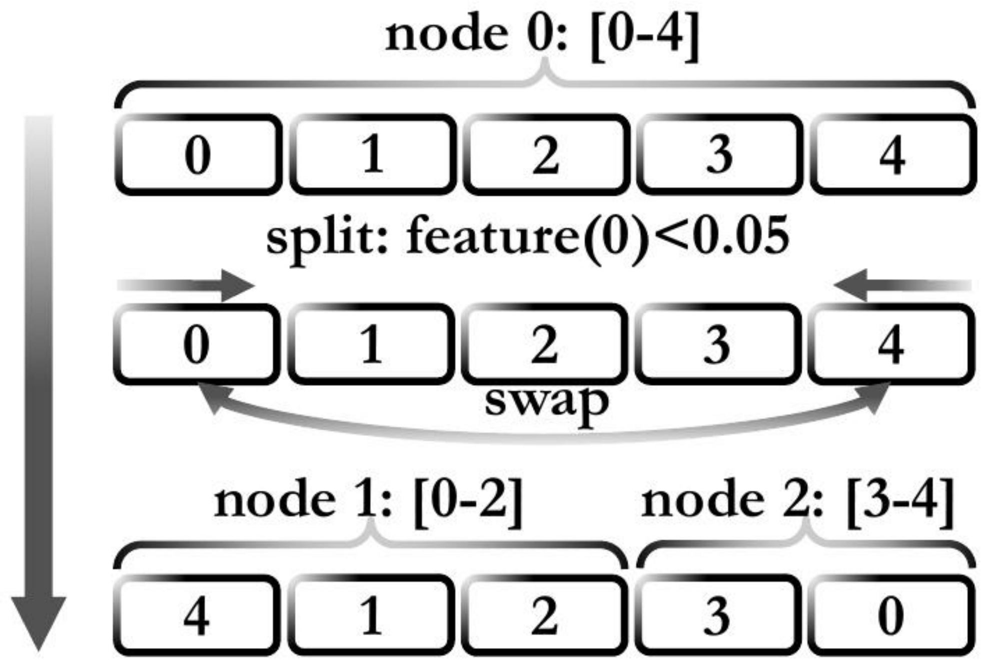


Figure 9: Node-to-instance Index.

GPU-GBDT[6]

- sparse representation
  - similar to Column Block
- fine-grain parallelism
  - parallelizing the score computation of each split point: segmented prefix sum --> GHSum
  - select best split point: semented reduction
  - splitting a node: maintain values of each feature in the new node in sorted order
- Run-length encoding compression
  - sorted by feature values
  - repeated values can be compressed
  - directly splitting RLE elements
- thread/block workload dynamic allocation
  - under a memory constrain

TABLE I

DENSE AND SPARSE DATA REPRESENTATION

	Dense	Sparse
$\boldsymbol{x}_1$	$\langle 0.0, 0.0, 0.1, 0.0 \rangle$	$(a_3 : 0.1)$
$\boldsymbol{x}_2$	$\langle 1.2, 0.0, 0.1, 0.6 \rangle$	$(a_1 : 1.2); (a_3 : 0.1); (a_4 : 0.6)$
$\boldsymbol{x}_3$	$\langle 0.5, 1.0, 0.0, 0.0 \rangle$	$(a_1 : 0.5); (a_2 : 1.0)$
$\boldsymbol{x}_4$	$\langle 1.2, 0.0, 2.0, 0.0 \rangle$	$(a_1 : 1.2); (a_3 : 2.0)$

$$a_1 = (\boldsymbol{x}_2 : 1.2); (\boldsymbol{x}_4 : 1.2); (\boldsymbol{x}_3 : 0.5)$$
$$a_2 = (\boldsymbol{x}_3 : 1.0)$$
$$a_3 = (\boldsymbol{x}_4 : 2.0); (\boldsymbol{x}_2 : 0.1); (\boldsymbol{x}_1 : 0.1)$$
$$a_4 = (\boldsymbol{x}_2 : 0.6)$$

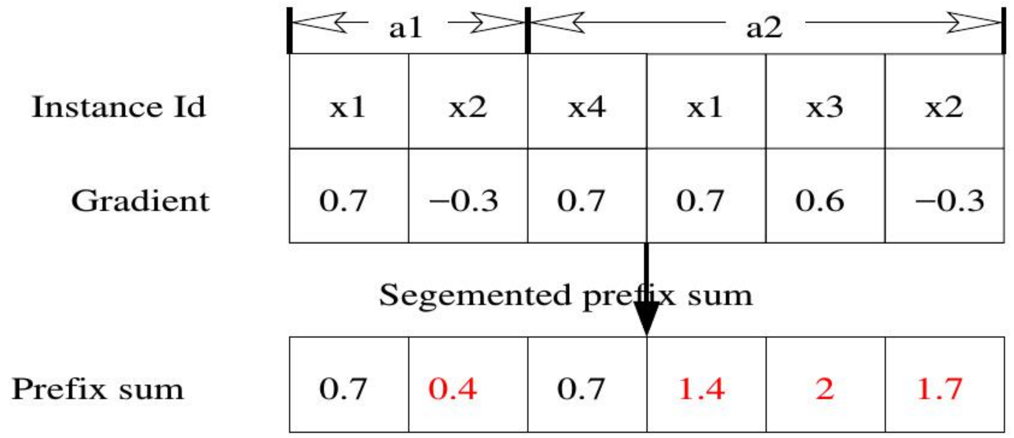


Fig. 1. Example results of segmented prefix sum

Outline

- Problem
- Related work
- Proposal

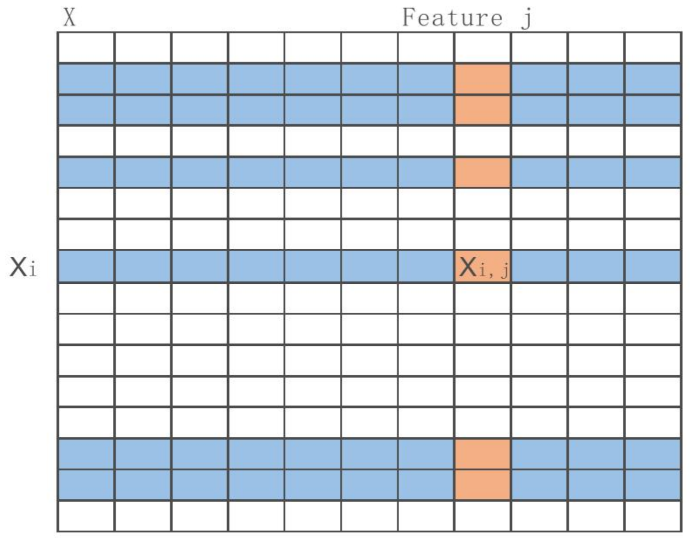
Assumption

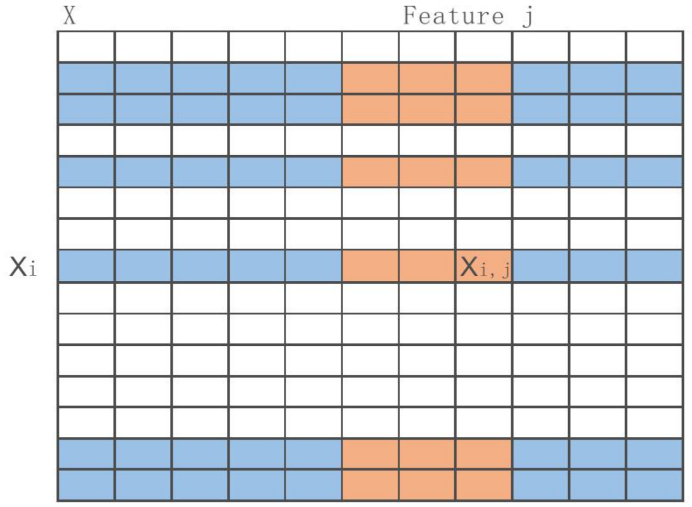
- Assumption:
  - Model: GHSum, (Bins, g, h)
  - Model size << Training Data size
- Training Data N samples, M Features, build tree with L levels
- Model GHSum with bin size B
  - at each node, size: B*M
  - at each level,size: 2^{L-1}*B*M
- Most cases
  - M <1K, L<8, B~[20,100]
  - high dimension can be an issue
  - deep tree can be an issue (GBT prefers shallow trees)

Dataset Statistics

- example:
  - Higgs: N = 10m, M = 28
    - L = 6, B = 128; model size at level L, 32*128*28 << 10m
  - KDD10: N=19M, M=29M
    - feature enginering: expand a categorical feature into a set of binary features, combination*

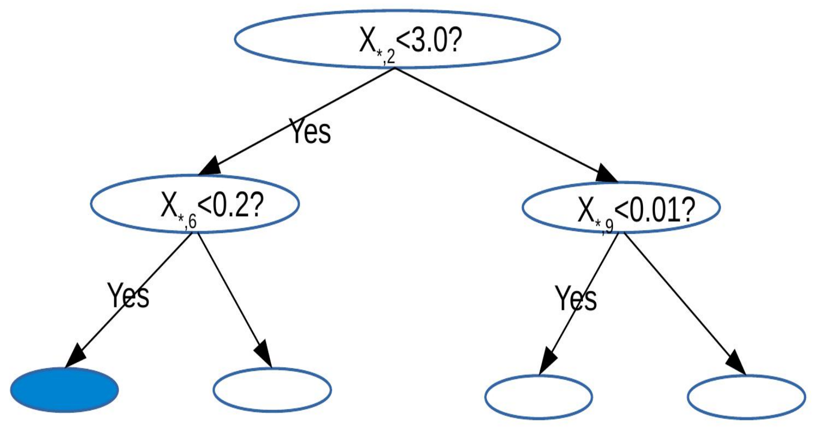
Cache Friendly Data Orgnization





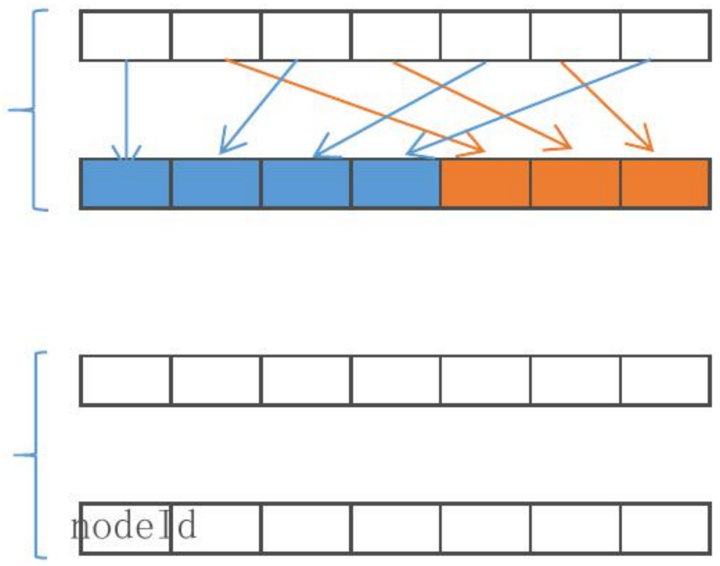
- Issue: non-continuous memory access in feature level parallelism
  - prefer column-wised data orgnization
  - dynamic instance set
  - N is large, (g,h) can not even fit into LLC cache
- Ideas
  - multi-columns to fit one cache line
  - clolumn block to get a small range of (g,h) access cache friendly

Training data access



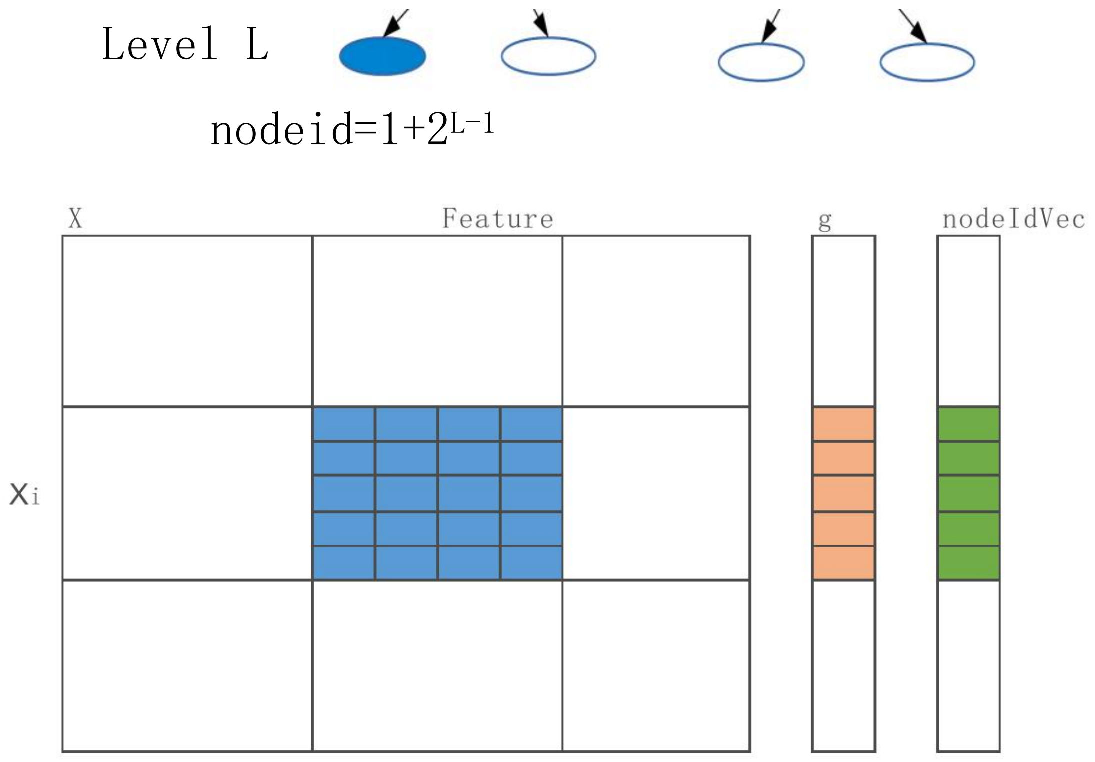
- Issue: redundent access to training data in node level parallelism
  - prefer row-wised data orgnization, however, it conflicts with feaure level parallelism
- Ideas
  - process layer-by-layer
  - Keep column-wised orgnization
  - Keep all GHSum of the same level in cache
  - Single pass on train data for each level. (half when reuse $\text{GHSum}_{\text{\_parent}}$)

Instance Set access



- Issue:Dynamic instance set
  - move instance id around after node split
  - maintain in order to process node-by-node
- Ideas
  - encode <nodeid, level> into a uniq nodeID, (easy for binary tree)
  - a nodeID vector to maintain the current leaf level instance sets directly
  - support random access by instance id <i>
  - sequential scan to get $<f_i, g_i, h_i>$

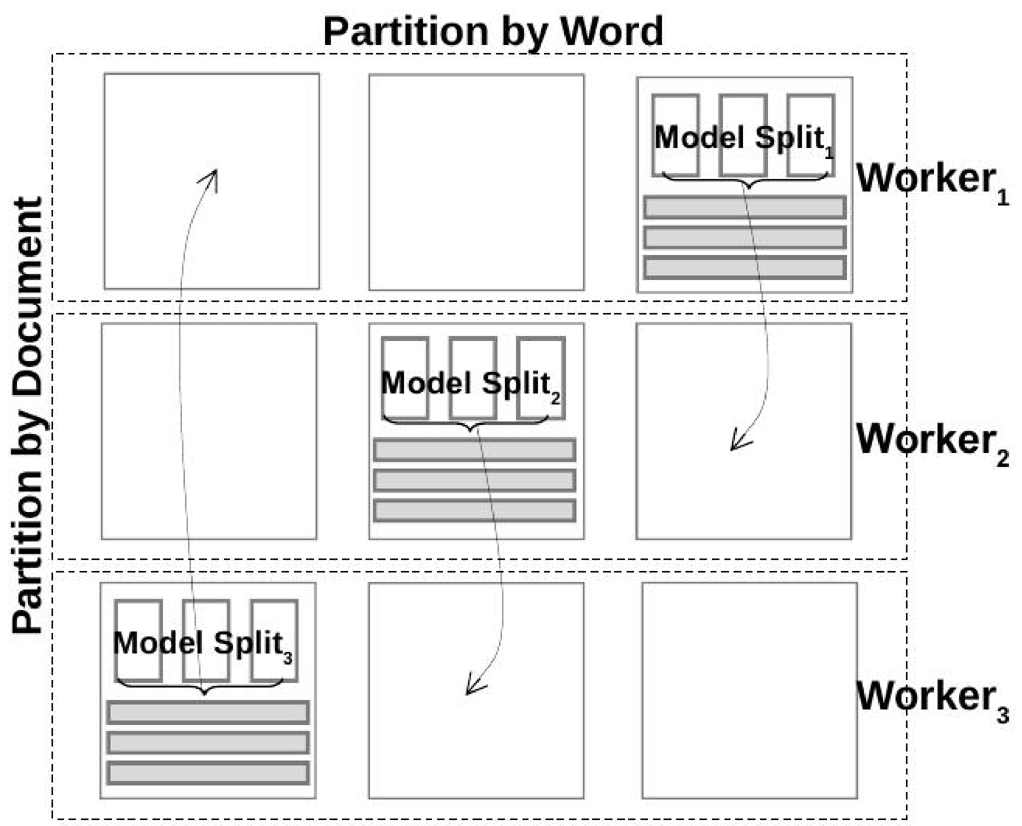
Block-based Data Orgnization and Parallelsim



```
for j in column_block:
  for each $x_{i,j}$ in the column $X_{,j}$:
    nodeid = nodeIDVec[i]
    bindid = Bins[j][$x_{i,j}$]
    GHSum[nodeid][j][binid] += g[i]
```

- Column Block
  - the basic unit for parallelism
  - column-wised data orgnization
  - dense/sparse support
  - sorted by feature values
- Set the block number/size to
  - fit all active <g,h>,nodeIDVec blocks in LLC
  - fit all active GHSum blocks in L2
  - loop order can be optimized

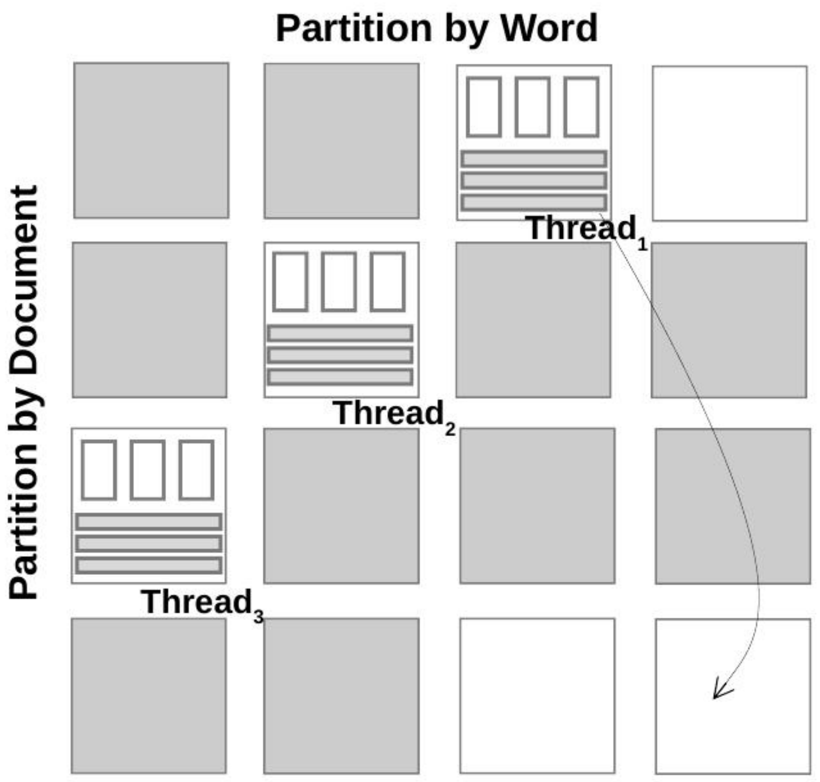
Unified Computation Model for Parallelism



(a) model rotation

- Proposal
  - unified computation model: model rotation
- Data partitioned to blocks
  - Data Parallelsim:
  - Model parallelism: GHSum split by feature columns
- Scheduler to avoid update conflicts

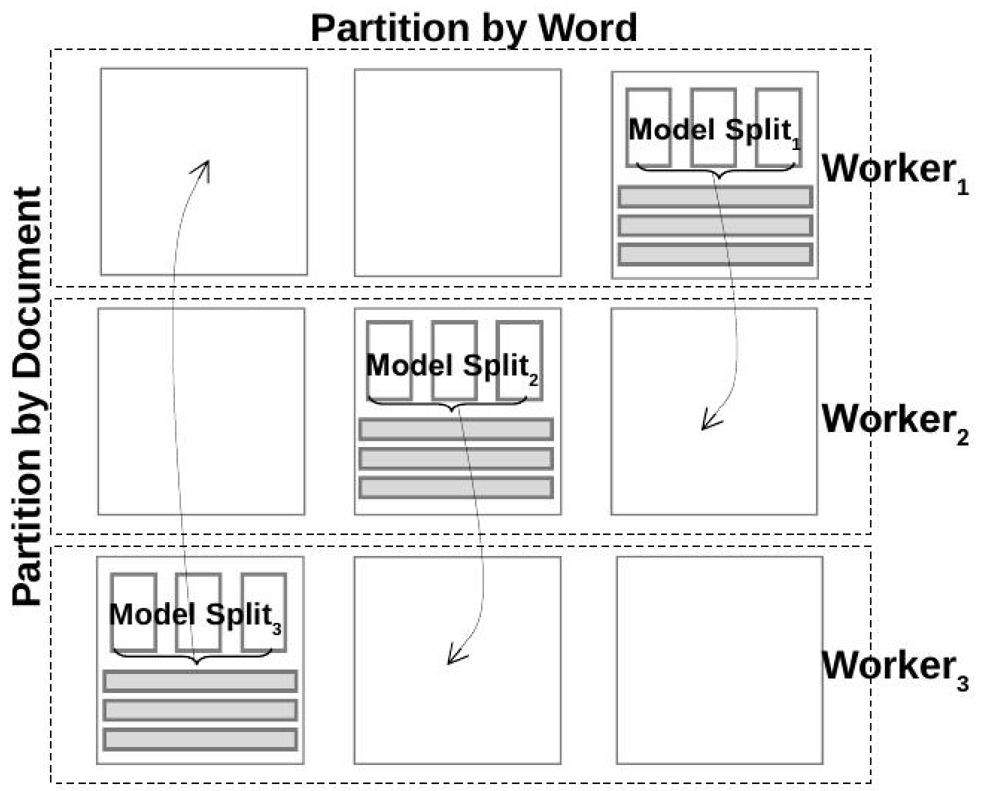
Dynamic Scheduling in Shared Memory systems



(b) dynamic scheduling

- a low cost solution to remove synchronization overhead
- number of partitions is larger than the number of threads,
- always ‘free’ rows and columns avaliable when one thread finishes its current task.
- cache-aware block size
- sparse representation ready

Distributed Model Rotation



(a) model rotation

- Training data randomly partitioned among nodes
- Model partitioned among nodes
  - split by feature columns
- Scheduler to avoid update conflicts
  - rotate local model partition to neighbor at each sub-step
  - finish after K sub-steps(K nodes)
  - kind of a ring-allreduce

Reduce Overhead of Collective Communication

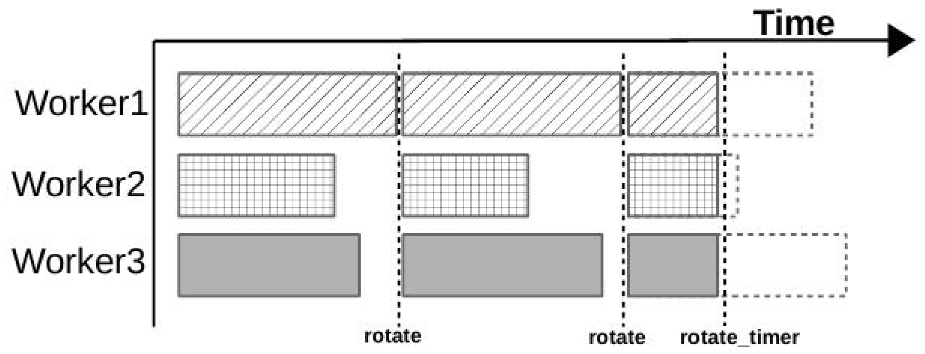


Figure 3: Timer to Control the Synchronization Point

- Design
  - Each worker only works for the same period of time and then they do synchronization all together.
  - They all use a timer to control the synchronization point rather than waiting until all the blocks to finish.
  - not ready for fault tolerance, but ready for straggler
- Stochastic GBT support
  - This adjustment does not change the property of the uniform random selection of blocks.

More to consider

- Feature Bundle
  - deal with category features
- Feature Sampling
  - design scheduler with Gradient-based priority
- Data Compression
  - model compression: low-precision compression
  - traning data compression: RLE

References

- [1]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, 2016, pp. 785–794.
- [2]Q. Meng et al., “A communication-efficient parallel algorithm for decision tree,” in Advances in Neural Information Processing Systems, 2016, pp. 1279–1287.
- [3]G. Ke et al., “LightGBM: A Highly Efficient Gradient Boosting Decision Tree,” in Advances in Neural Information Processing Systems, 2017, pp. 3149–3157.
- [4]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, 2018, pp. 1363–1376.
- [5]J. Jiang, J. Jiang, B. Cui, and C. Zhang, “TencentBoost: A Gradient Boosting Tree System with Parameter Server,” in 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017, pp. 281–284.
- [6]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), 2018, pp. 234–243.