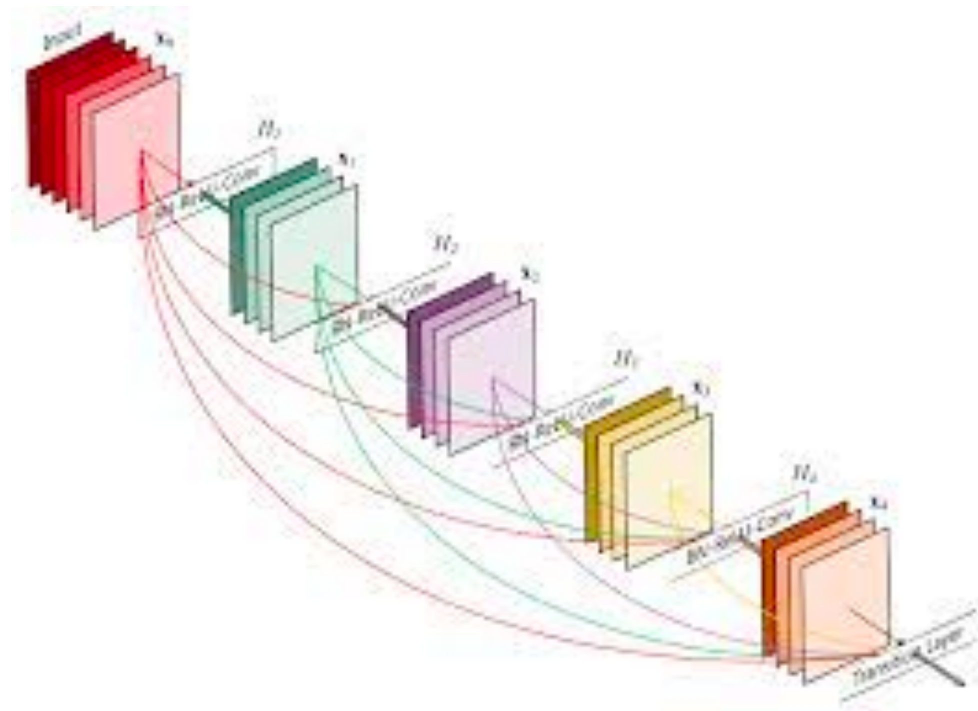
Scaling Distributed Machine Learning with Parameter Server

Authors: Mu li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su

Presenter: Zhihao Zhang, Date: 02/02/2022

Motivations

large models



large datasets

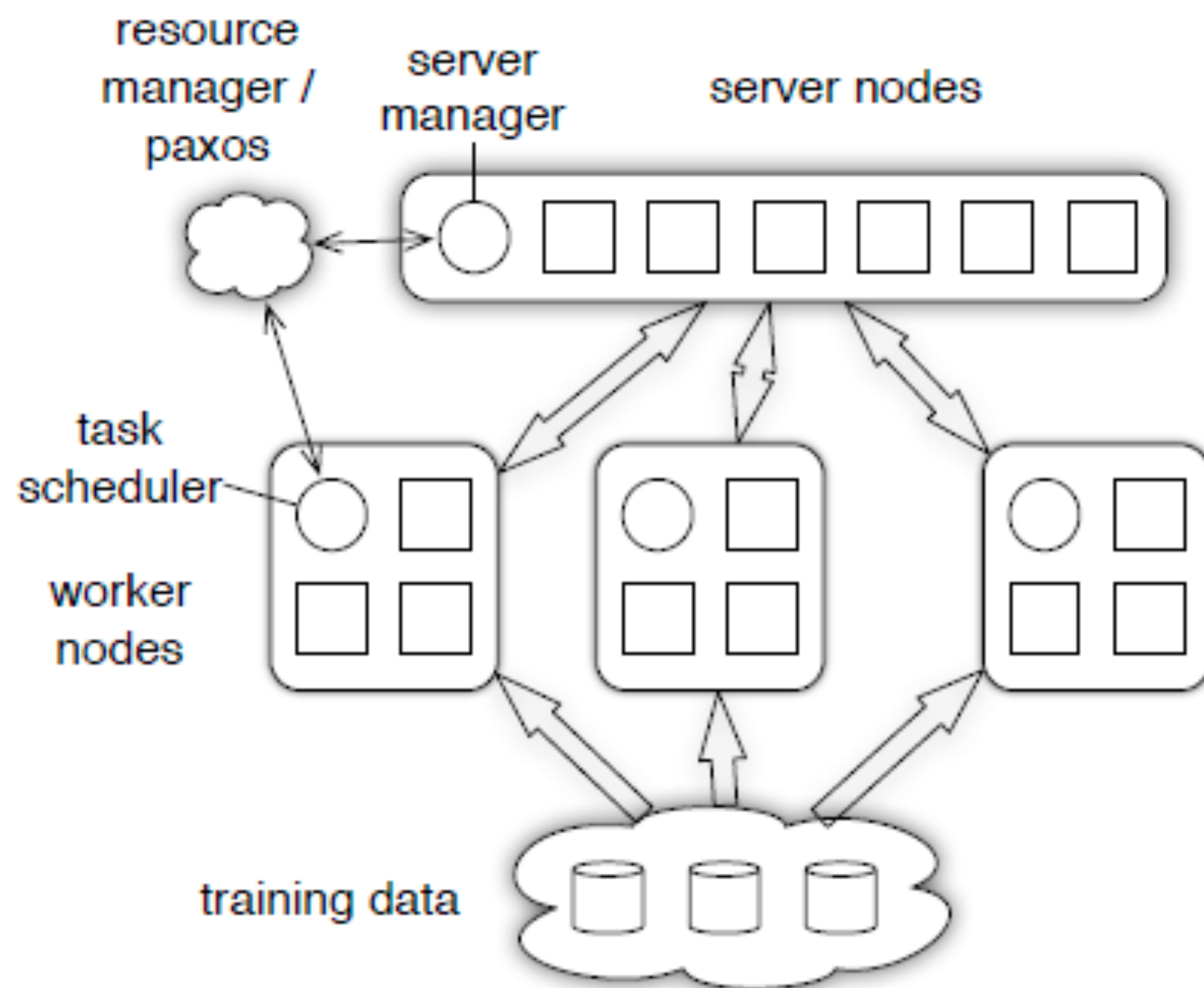
+

Distributed system

Challenges

- Accessing huge amount of parameters (push/pull/update gradients and weights)
- Sequential nature of DL training algorithms VS efficiency
- Fault tolerance and flexibility when scaling up

Preliminaries



Methodology

Accessing huge amount of parameters (push/pull/update gradients and weights)

Problem

- Message between server nodes :{timestamp, key-value pair}
- Suppose we have n workers with m parameters on each
- Spatial complexity O(nm) for both timestamp and key-value pair

ML features

- Usually update in a block fashion (layer by layer)
- Static computational graph
- Potentially sparse

Solutions

- Uses ranges instead of per-weight wise timestamp: O(nm) -> O(nk)
- Key compression: Cache key locally for the first message and hash key list afterwards
- Potentially sparse: Remove key-value pairs for value=0

Methodology

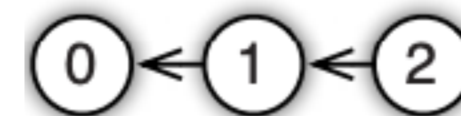
Sequential nature of DL training algorithms VS efficiency

Problem

- DL model training is usually sequentialized: SGD, BGD
- Synchronization needs to wait for the slowest worker to finish
- Asynchronies update might lead to slow convergence rate

Distributed system features

- Sequential

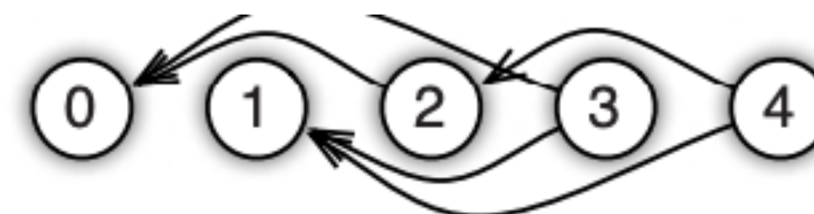


(a) Sequential

- Eventual

(b) Eventual

- Bounded delay



(c) 1 Bounded delay

Solutions

- Uses hyper-parameter τ in bounded delay to balance between convergence rate and efficiency for different tasks

Methodology

Fault tolerance and flexibility when scaling up

Problem

- Fault tolerance when machine failure happens
- Add or remove nodes
- Avoid full restart

Distributed system and DL features

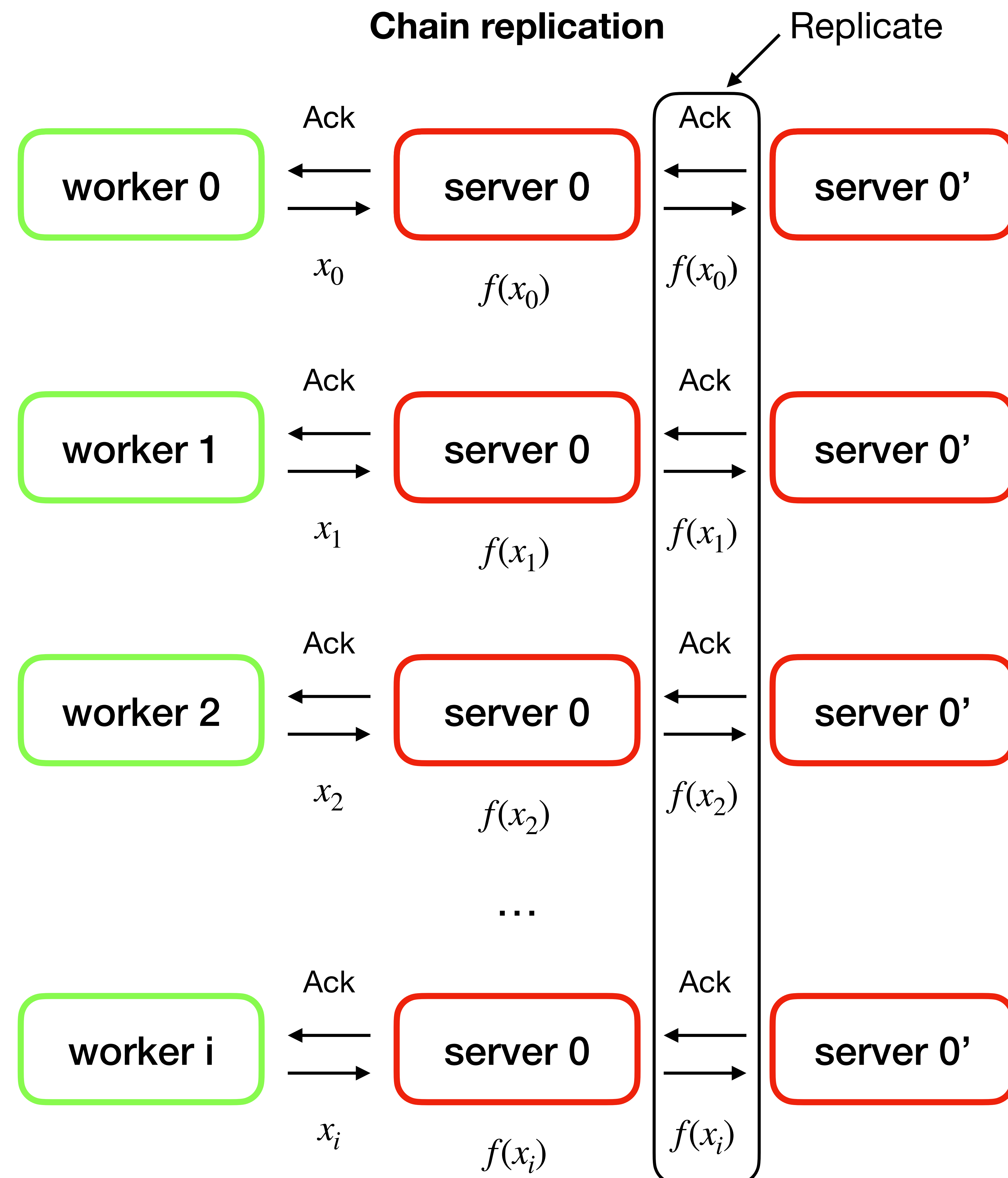
- Duplicate server nodes
- Consistent Hashing
- Only after aggregation information matters for DL update

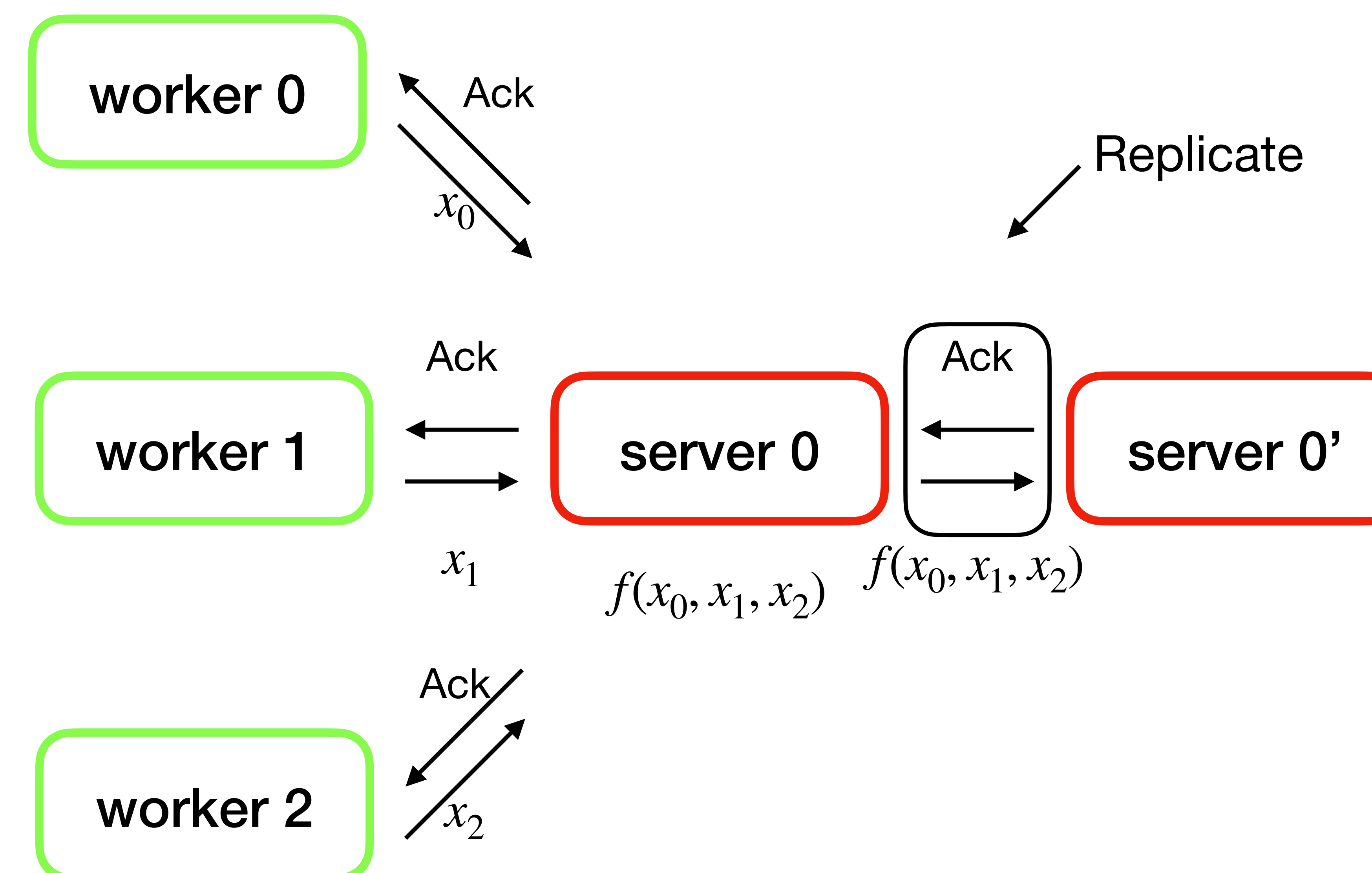
Solutions

- Uses Consistent Hashing for managing key-value pairs on different server nodes (could dynamically add or remove nodes more efficiently)
- Replicate after aggregation instead of chain replication

Methodology

Replicate after aggregation





Reduce a factor of worker numbers

Methodology

System flexibility

- User defined functions on server side (eg. calculating regularization terms)
- User defined message filtering (KTT, gradient thresholding)

Summary

Paratemeter messaging

- Range timestamps
- Cache and hash Keys
- Sparse weight matrix cleaning

Asynchronisation vs Synchronisation

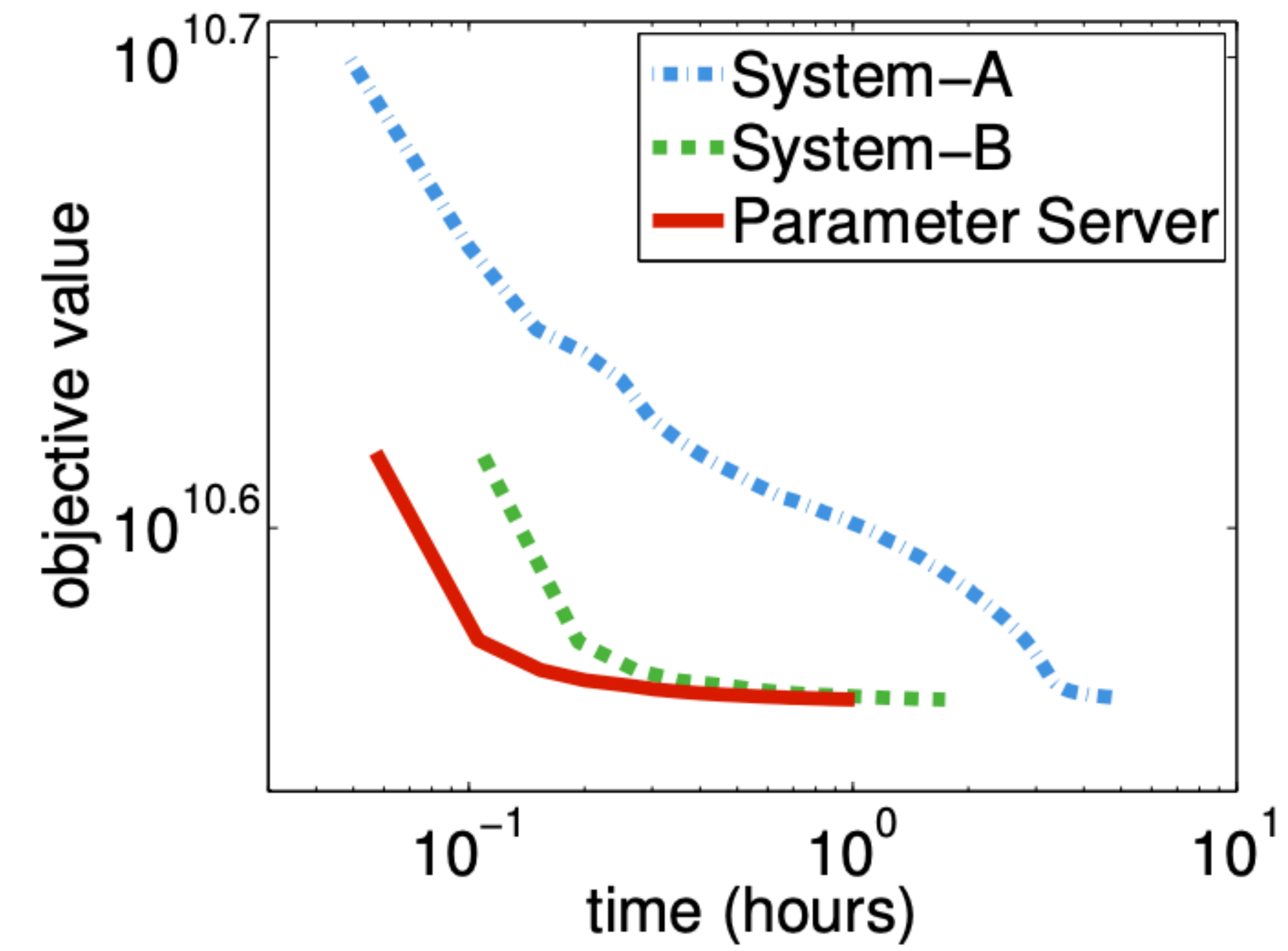
- Uses a hyper-parameter in bounded delay to balance them off

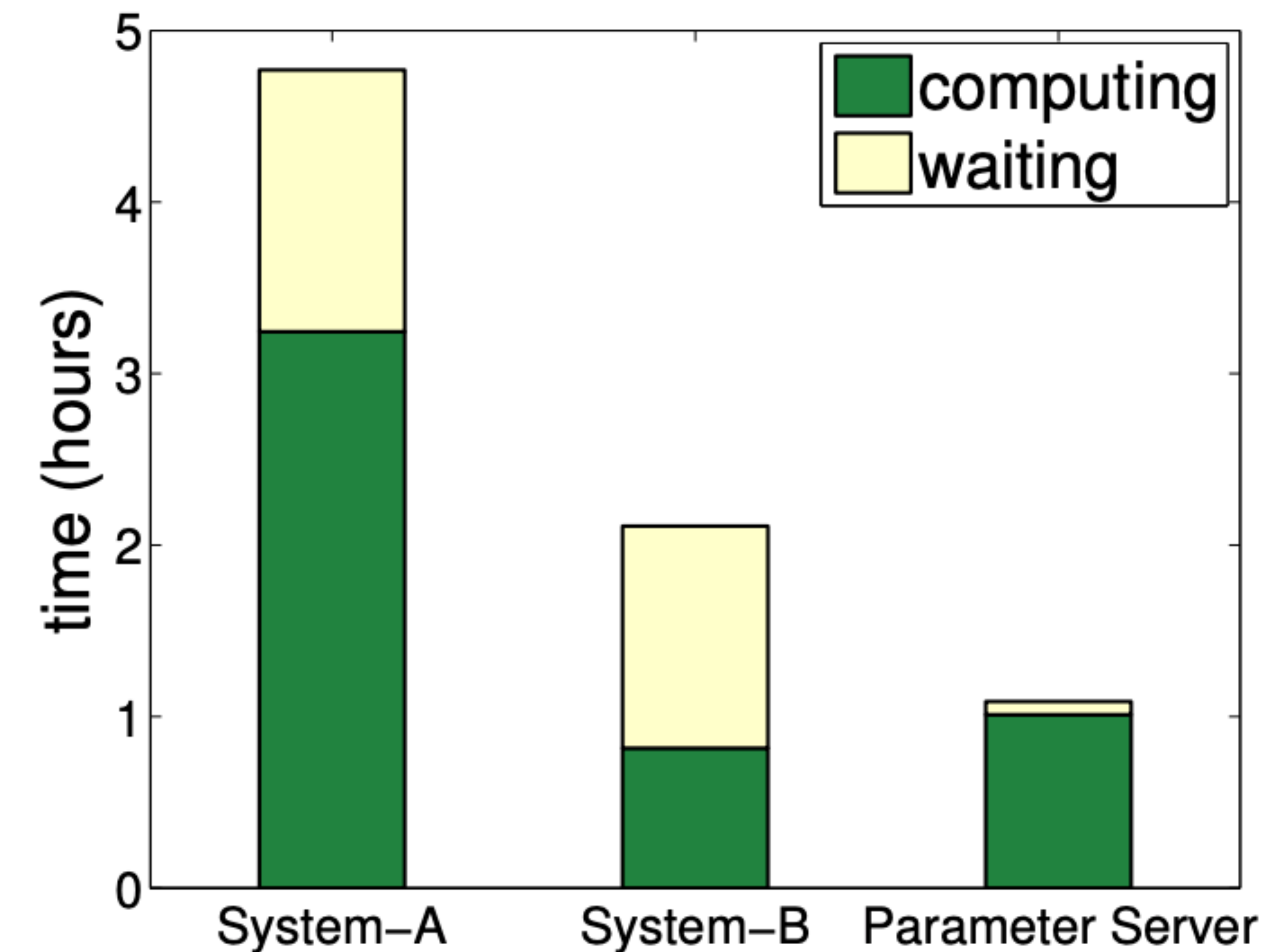
Fault tolerance and flexibility

- Consistent Hashing
- Replicate after aggregation

Discussion

	Method	Consistency	LOC
System A	L-BFGS	Sequential	10,000
System B	Block PG	Sequential	30,000
Parameter Server	Block PG	Bounded Delay KKT Filter	300





Questions

- Why calculating gradients for regularization terms can be handed over to server side?
- In which cases do asynchronisation might introduce duplicate efforts and inconsistencies?

XGBoost: A Scalable Tree Boosting System

Tianqi Chen, Carlos Guestrin

Presented by: Giulio Zhou

Overview of XGBoost

dmlc XGBoost

Problem: How do we efficiently train gradient boosted decision trees (GBDTs) on large tabular datasets?

Challenges:

- Existing implementations (pre-2015) were mostly single-core and in-memory, and could scale poorly to datasets with large numbers of features or instances.
- Data in tabular settings may be highly sparse or missing.
- Sorting and quantile sketch operations are not an obvious fit for accelerators focused on linear algebra operations.

Proposed Solution: XGBoost is a system for training GBDTs that supports parallel and approximate tree learning, as well as cache-aware and out-of-core computation.

Background: Data and Models in Machine Learning

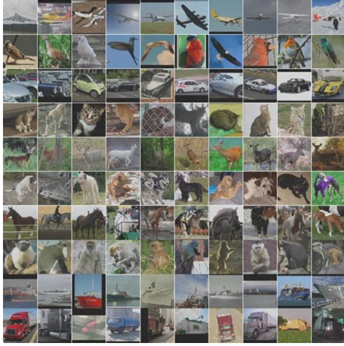
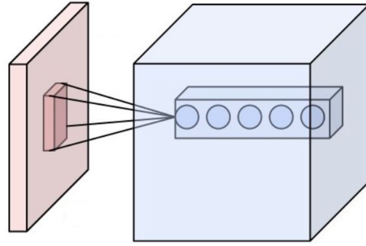
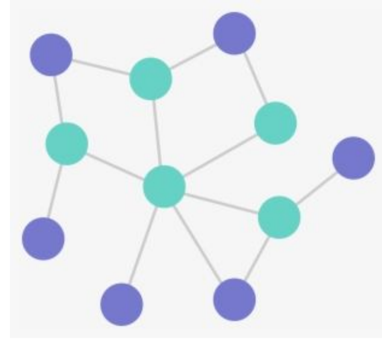


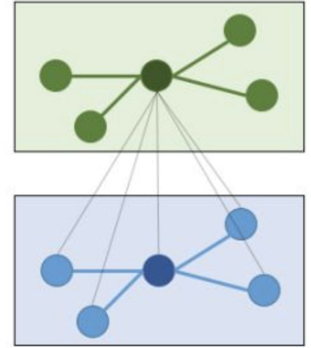
Image Data



CNN, MLP

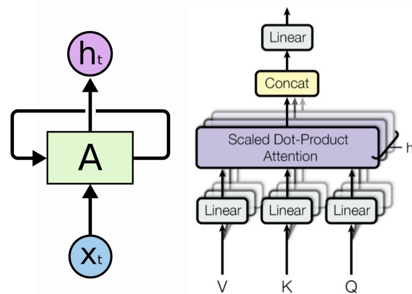


Graph Data



GNN

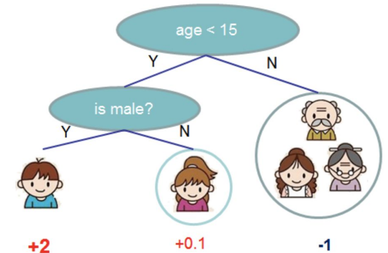
Language Data



RNN, Transformer

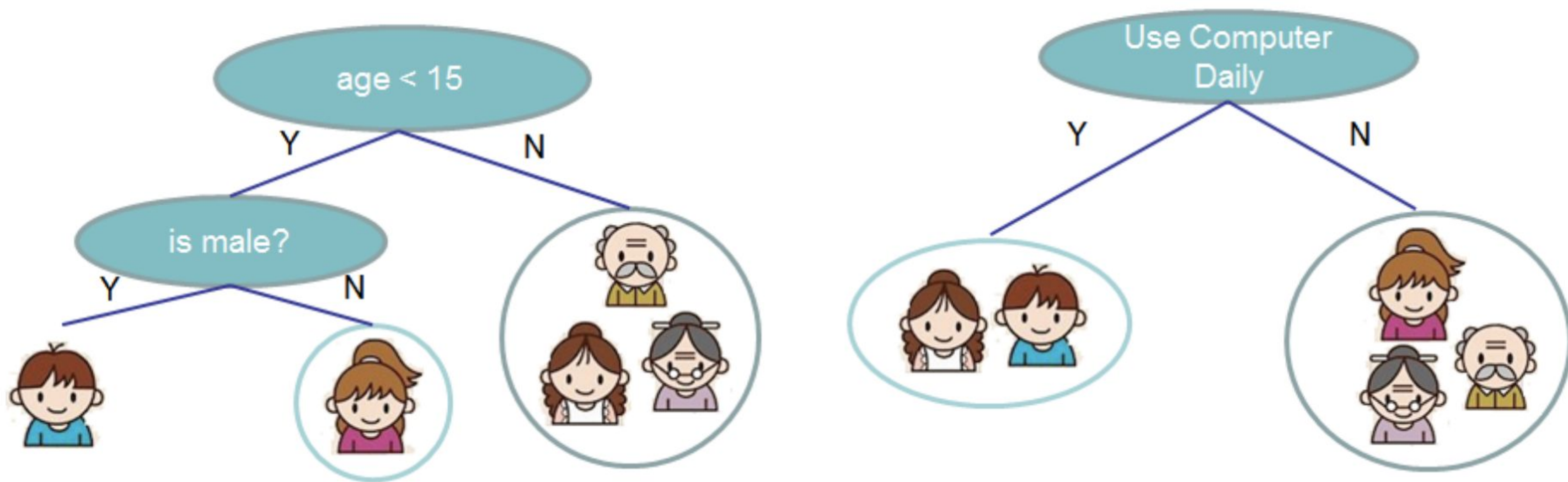
	A	B	C	D
1	**Region**	**Salesman**	**Date**	**Revenue**
2	North	Rob	10/21/2017	$2,059
3	South	Joe	10/9/2017	$1,908
4	North	Rikki	9/27/2017	$1,429
5	East	Chris	9/15/2017	$2,588
6	North	Rikki	9/3/2017	$2,085
7	East	Chris	8/22/2017	$1,996
8	South	Joe	8/10/2017	$1,718
9	North	Rikki	7/29/2017	$2,851
10	North	Rikki	7/17/2017	$2,735
11	East	Chris	7/5/2017	$2,864

Tabular Data



GBDT

Background: Decision Trees



Source: XGBoost Paper

Background: Decision Trees and Boosting

Objective is convex loss over sum of tree function outputs + regularization.

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$
$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

Idea behind boosting: Optimize for the objective *in the space of functions*, i.e. by learning tree functions and their associated leaf weights.

$$\hat{y}_i = \phi(\mathbf{x}_i) = \sum_{k=1}^{K} f_k(\mathbf{x}_i)$$

Learning all K trees of an ensemble jointly is intractable → learn trees *sequentially* to *greedily optimize* the objective function.

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} [g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t)$$

$$w_j^* = -\frac{\sum_{i\in I_j} g_i}{\sum_{i\in I_j} h_i + \lambda},$$

Stopping criteria may be manually specified (e.g. number of levels) or based on loss convergence.

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{T} \frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T.$$

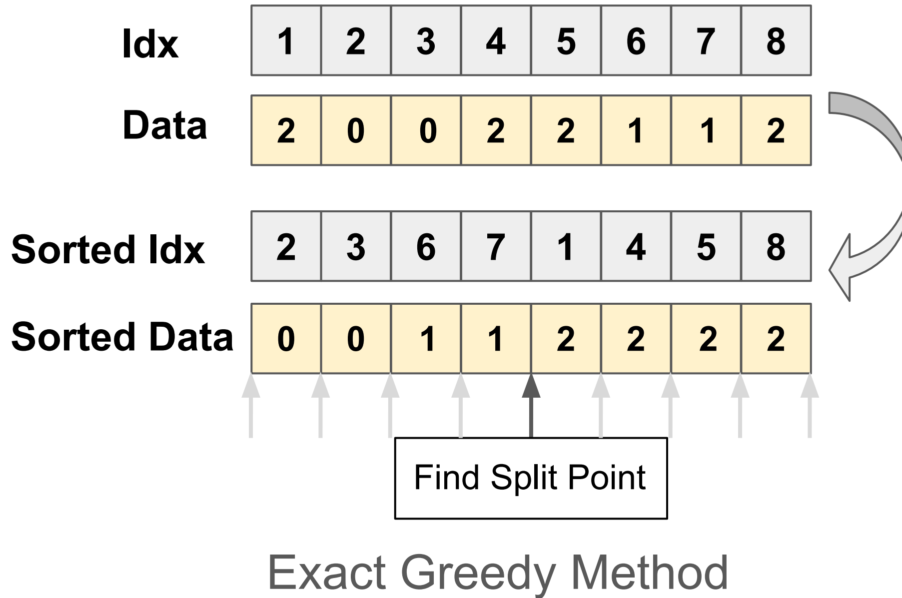
XGBoost

- ***Core algorithm:*** (sequentially) learn an ensemble of gradient-boosted decision trees that greedily minimizes convex loss function at each timestep.
- **Notable modeling features:** second-order optimization, column subsampling, weight shrinkage, fast tree splitting methods (*below*).
    - *Weighted Quantile Sketches*: Even splits via fast 1D gradient-weighted sorting.
    - *Sparsity-Aware Split Finding*: Fill NULLs with automatically learned default values.
- **System-level optimizations:** column block for parallel learning, cache-aware access, blocking for out-of-core computation.

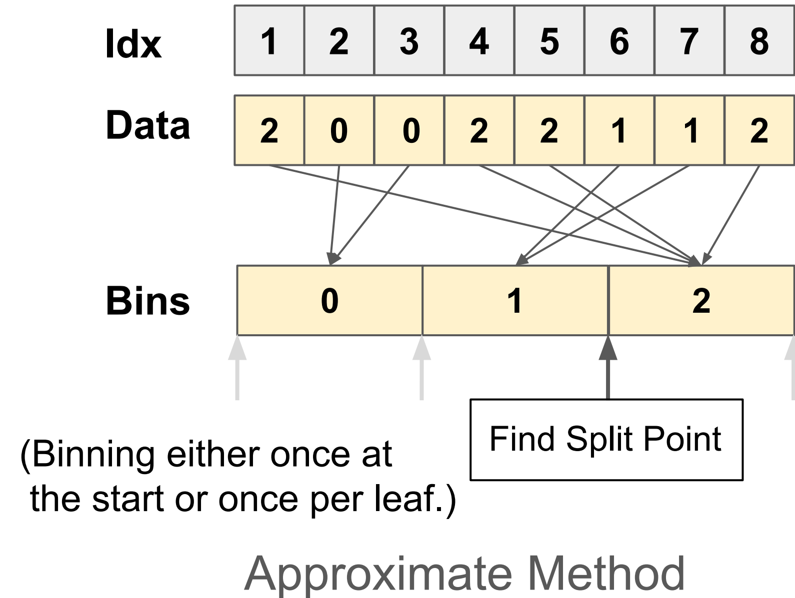
With these contributions, XGBoost helps make gradient-boosted decision trees more practical in the modern context of large datasets and parallel computation.

Exact Greedy and Approximate Splitting Methods

For each feature:



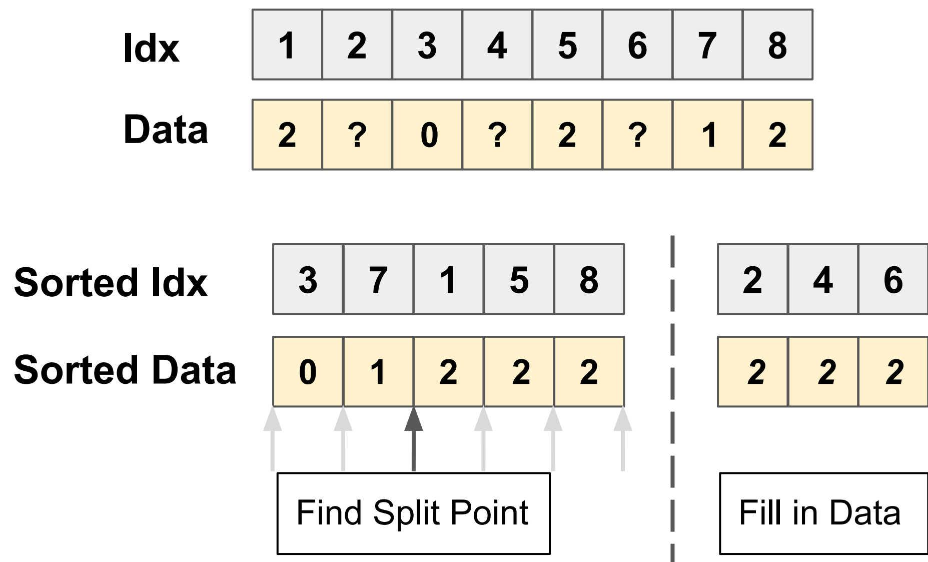
Exact Greedy Method



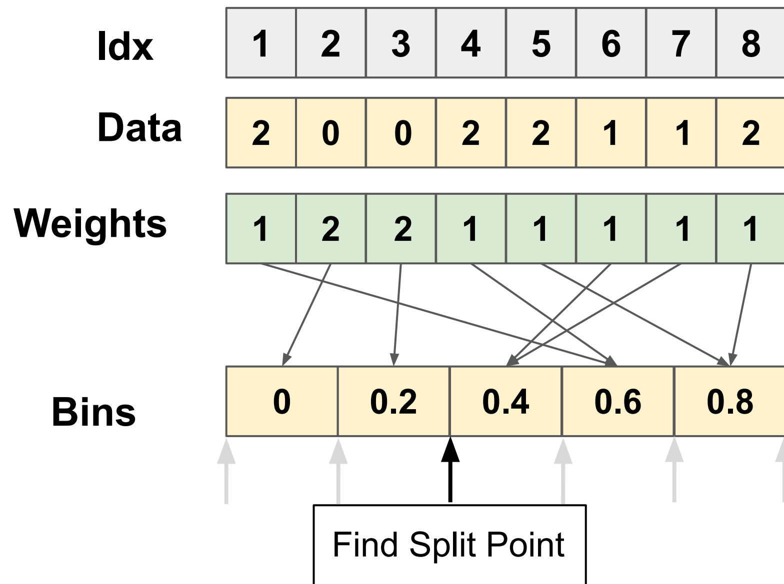
Approximate Method

Then, choose the point that reduces the loss by the largest amount.

Algorithms for Optimized Splitting (for tree learning)



Sparsity-Aware Split Finding
(Exact Greedy)



Weighted Quantile Sketches for Split Sorting
(Approximate)

System-level Optimizations

Column block for parallel learning — Columns are first sorted locally and the resulting sorted indices (per-column) are organized in blocks. Allows distribution of blocks to other machines to perform the sorting operations in parallel.

Cache-aware access — For exact greedy, rows can be pre-fetched to avoid non-contiguous memory accesses. For approximate, choose a cache block size to balance parallelization v.s. cache misses.

Blocking for out-of-core computation — Data divided into blocks stored on using *block compression* by column to reduce space and *block sharding* to increase disk throughput.

XGBoost addresses most of the stated challenges

Challenges:

- ❖ Existing implementations (pre-2015) were mostly single-core and in-memory.
  - ➢ Enables parallel, cache-aware, and out-of-core computation.
- ❖ Data in tabular settings may be highly sparse or missing.
  - ➢ Handles this issue for the exact match approach, but not the approximate setting.
  - ➢ (In fact, because categorical variables are one-hot encoded, XGBoost may increase the sparsity of the data.) [for the version of XGBoost in the paper]
- ❖ Sorting and quantile sketch operations are not an obvious fit for accelerators focused on linear algebra operations.
  - ➢ Optimizes access locality and data movement/placement.
  - ➢ Proposes algorithmic improvements such as weighted quantile sketch.

Discussion

1. XGBoost handles categorical features by one-hot encoding them. How might this affect the properties or quality of the learned trees?
2. XGBoost can handle sparseness efficiently when using exact greedy learning. Why is the histogram-based learning approach potentially inefficient for sparse data?