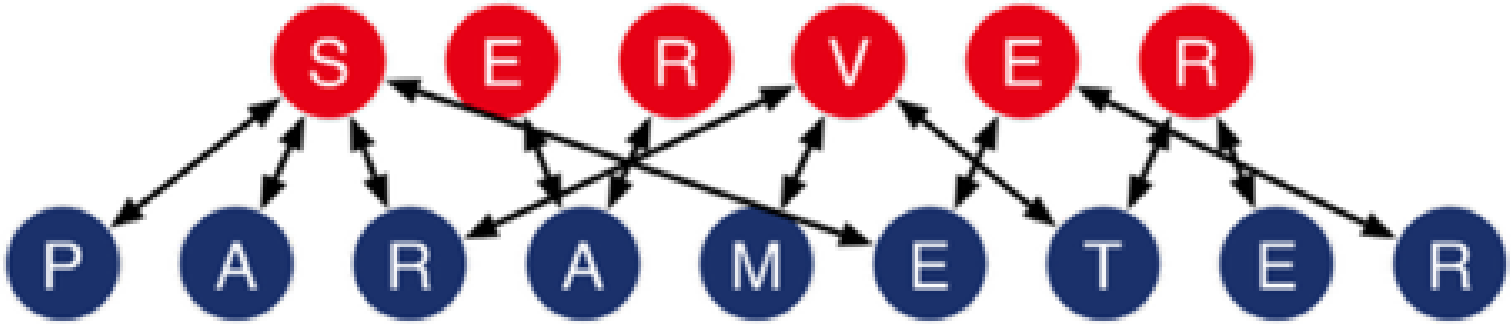
Scaling Distributed Machine Learning with the



BASED ON THE **PAPER** AND **PRESENTATION**:

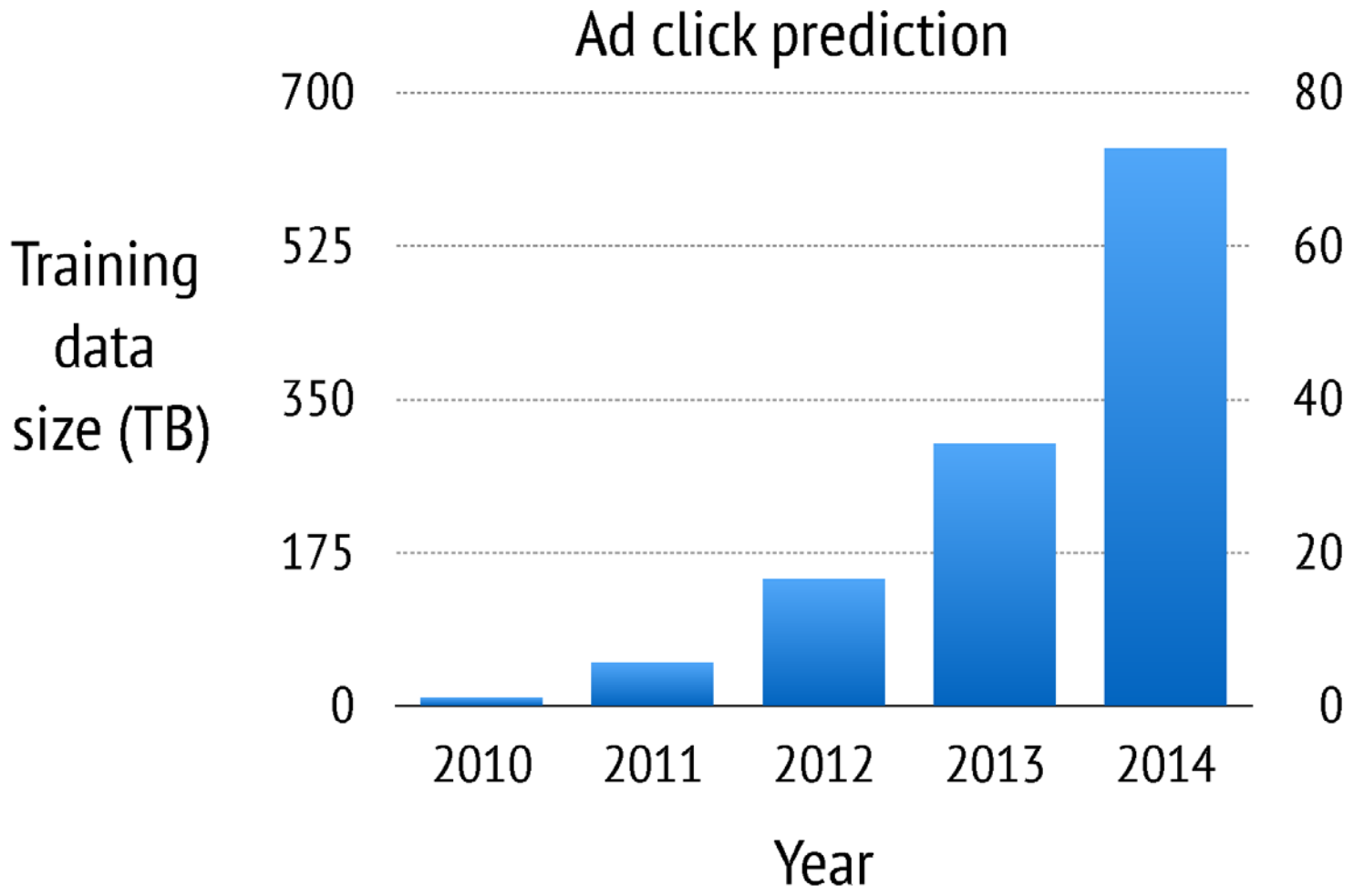
SCALING DISTRIBUTED MACHINE LEARNING WITH THE PARAMETER SERVER – **GOOGLE**, **BAIDU**, **CARNEGIE MELLON UNIVERSITY**

INCLUDED IN THE PROCEEDINGS AT **OSDI 14**

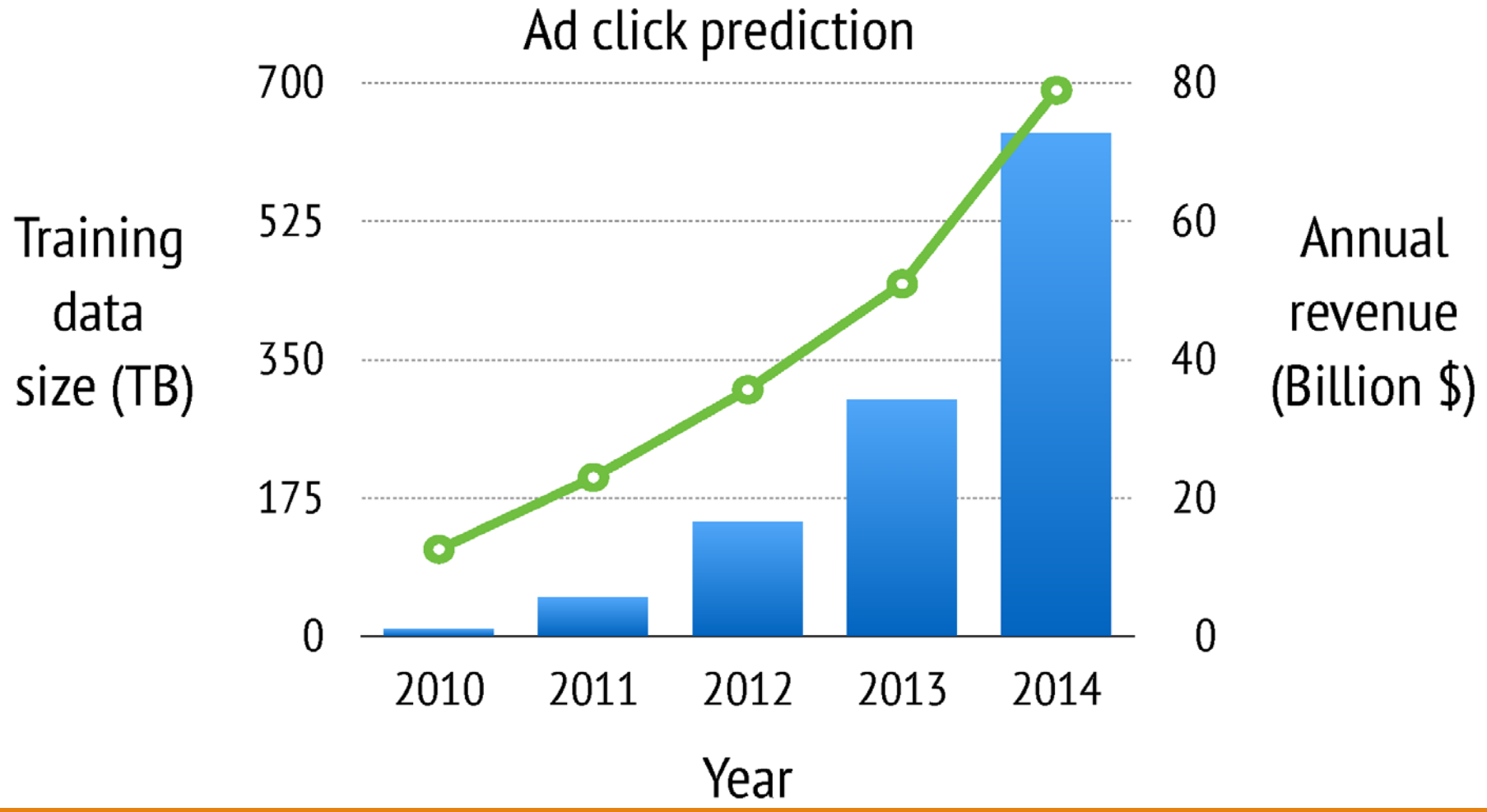
PRESENTATION BY: SANCHIT GUPTA

Machine learning is concerned with systems that can learn from data

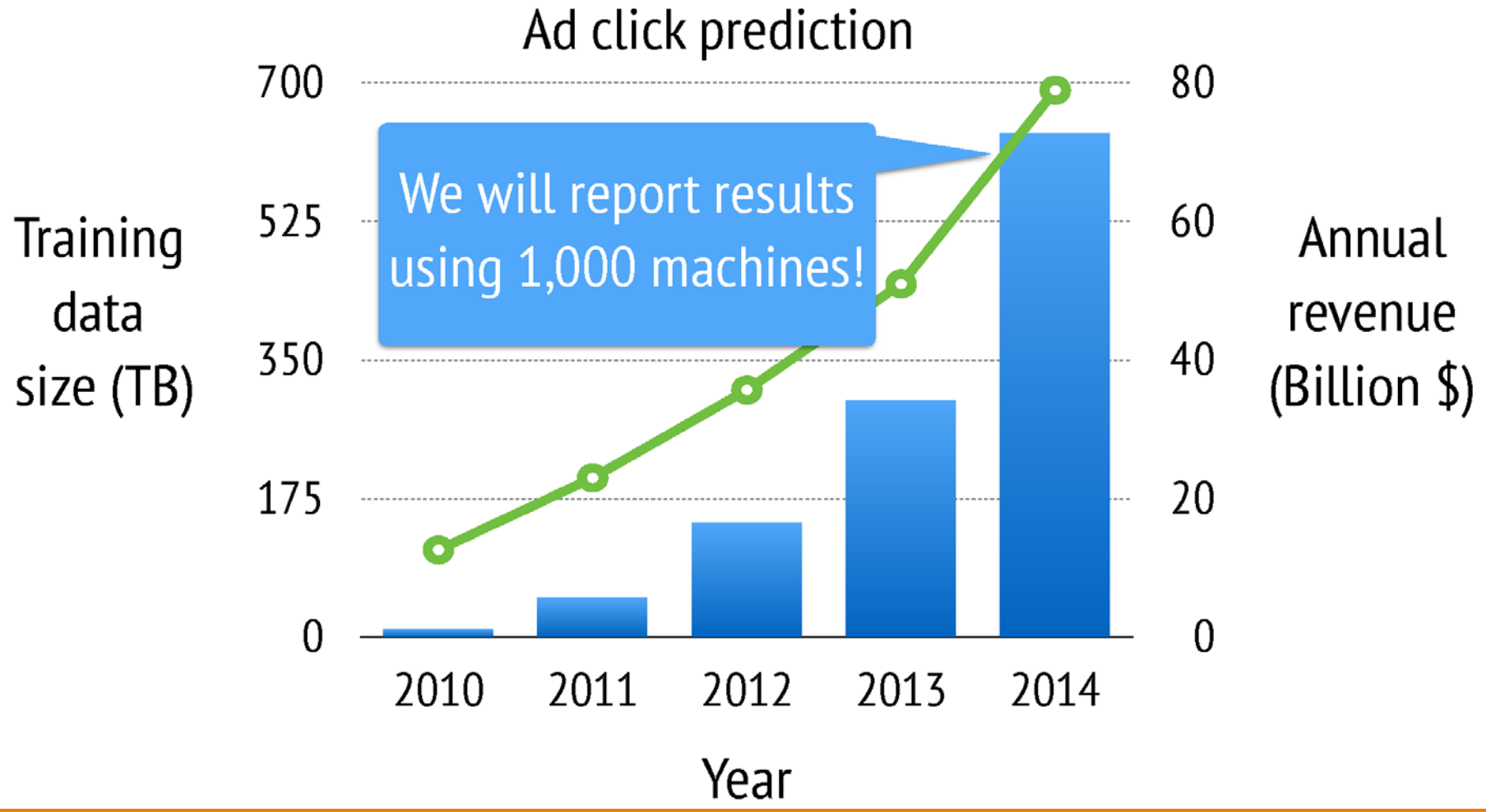
Machine learning is concerned with systems that can learn from data



Machine learning is concerned with systems that can learn from data

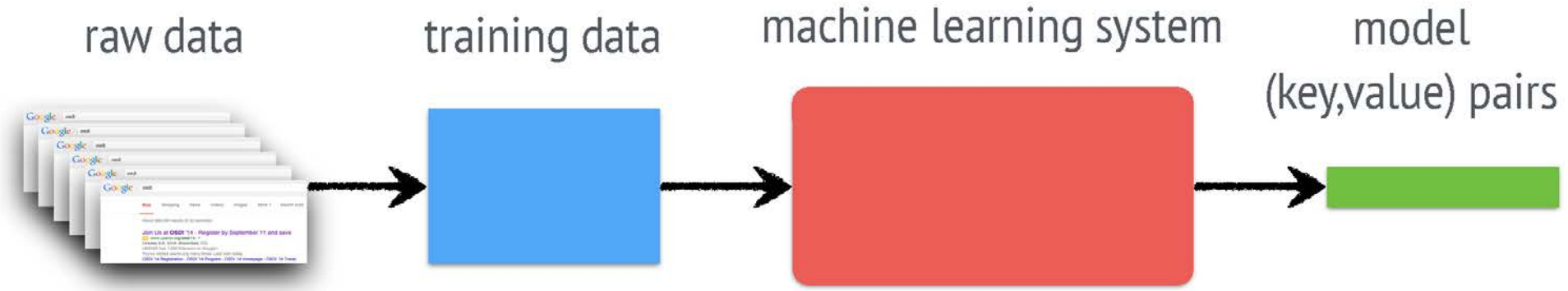


Machine learning is concerned with systems that can learn from data

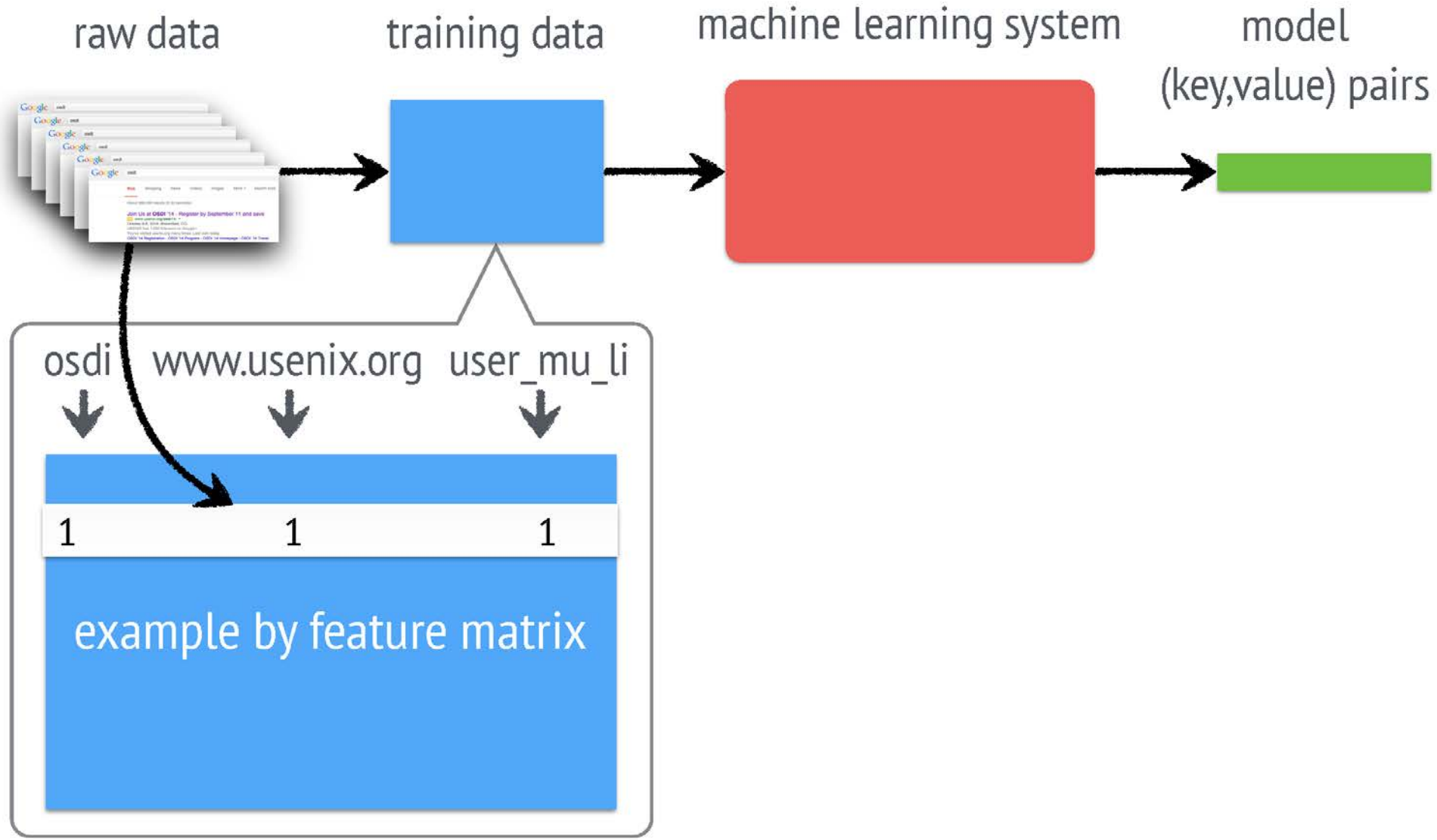


Machine learning is concerned with systems that can learn from data

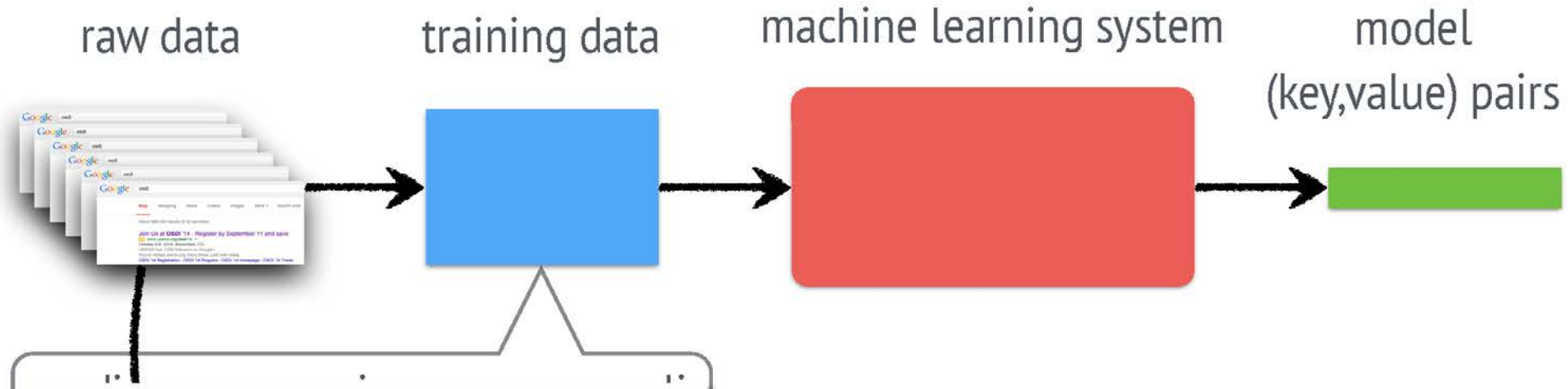
Overview of machine learning



Overview of machine learning



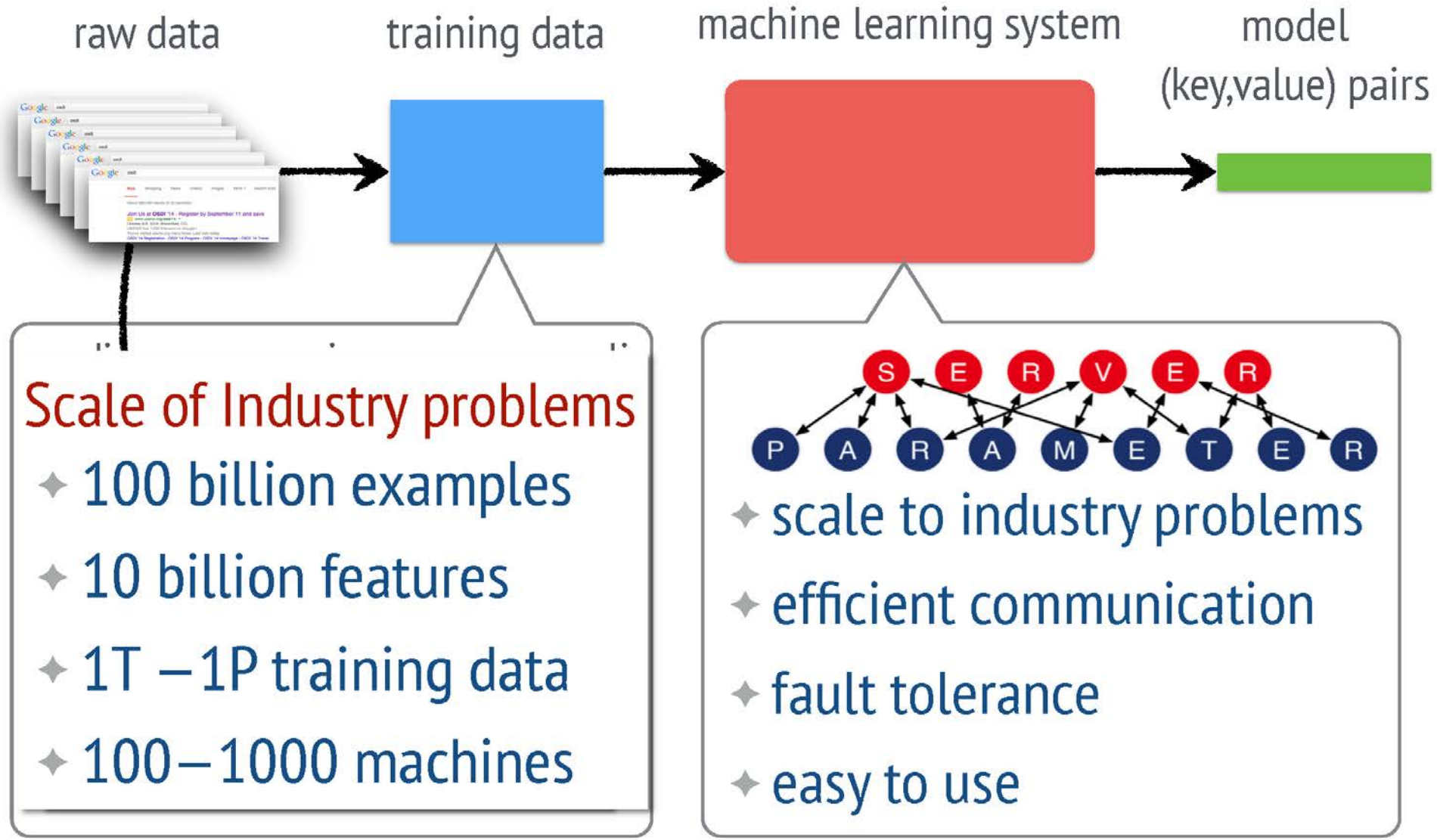
Overview of machine learning



Scale of Industry problems

- 100 billion examples
- 10 billion features
- 1T – 1P training data
- 100 – 1000 machines

Overview of machine learning



Characteristics & Challenges of ML jobs

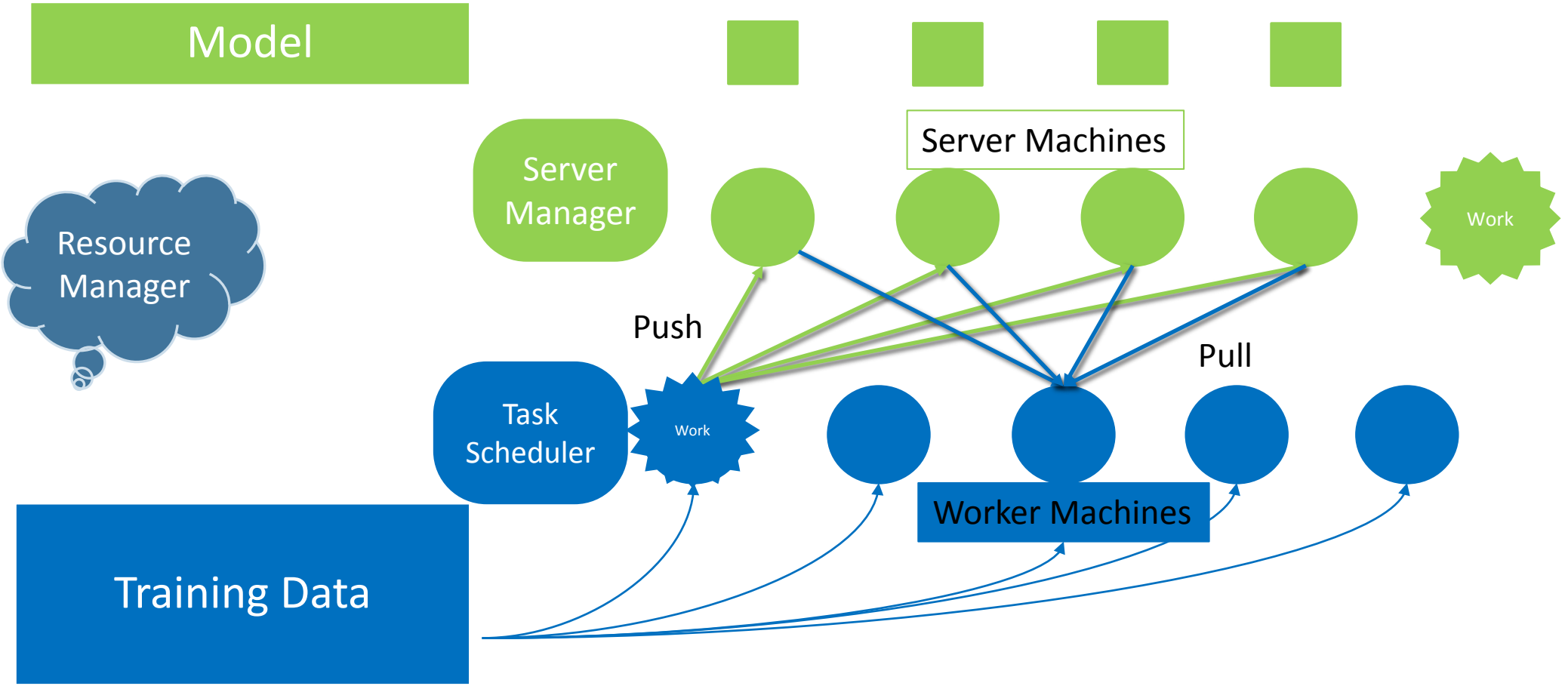
- Training Data is **Large** – 1TB to 1PB
- Complex Models with **Billions** and **Trillions** of Parameters
- Parameters are shared globally among worker nodes:
  - Accessing them incurs large **Network costs**
  - Sequential ML jobs require **barriers** and hurt performance by blocking
  - At scale, **Fault Tolerance** is required as these jobs run in a cloud environment where machines are unreliable and jobs can be preempted

Key Goals and Features of Design

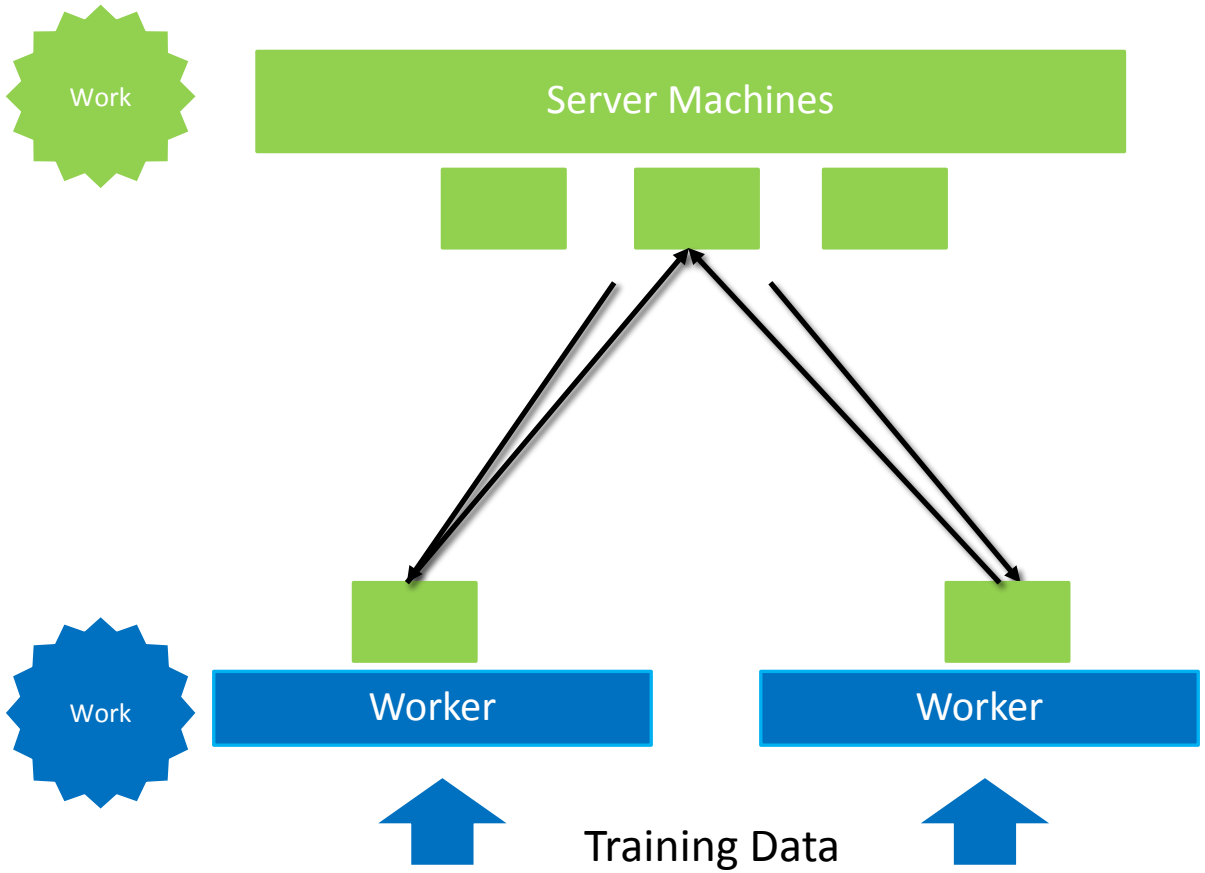
- **Efficient Communication**: asynchronous communication model (does not block computation)
- **Flexible Consistency Models**: Algorithm designer can balance algorithmic convergence and system efficiency
- **Elastic Scalability** : New nodes can be added without restarting framework
- **Fault Tolerance and Durability**: Recovery from and repair in 1 sec.
- **Ease of Use**: easy for users to write programs

Architecture & Design Details

Architecture: Data and Model



Example: Distributed gradient Descent



- Workers get the Assigned training data
- Workers **Pull** the Working set of Model
- Iterate until Stop:
  - Workers **Compute** Gradients
  - Workers **Push** Gradients
  - Servers **Aggregate** into current model
  - Workers **Pull** updated model

Architecture: Parameter Key-Value

- Model Parameters are represented as Key – Value pairs
- Parameter Server approach models the Key-Value pairs as sparse Linear Algebra Objects.
- **Batch** several key-value pairs required to compute a vector/matrix instead of sending them one by one
- **Easy to Program**! – Lets us treat the parameters as key-values while endowing them with matrix semantics

(Key, value) vectors for the shared parameters

(Key, value) vectors for the shared parameters

- Good for programmers: Matches mental model
- Good for system: Expose optimizations based upon structure of data

(Key, value) vectors for the shared parameters

- Good for programmers: Matches mental model
- Good for system: Expose optimizations based upon structure of data

Example: computing gradient

$$\text{gradient} = \text{data}^T \times (- \text{label} \times 1 / (1 + \exp (\text{label} \times \text{data} \times \text{model}))$$

Architecture: Range Push and Pull

- Data is sent between Workers and Servers using *PUSH* and *PUSH* operations.

- Parameter Server optimizes updates **communication** by using RANGE based *PUSH* and *PULL.*

- Example: Let ***w*** denote parameters of some model
  - *w.push(**Range,** dest)*
  - *w.pull**(Range**, dest**)***
  - These methods will send/receive all existing entries of ***w*** with ***keys*** in ***Range***

Architecture: Asynchronous tasks and Dependency

- Challenges for Data Synchronization:
  - There is a MASSIVE communication traffic due to frequent access of Shared Model
  - Global barriers between iterations – leads to:
    - idle workers waiting for other computation to finish
    - High total finish time

Task

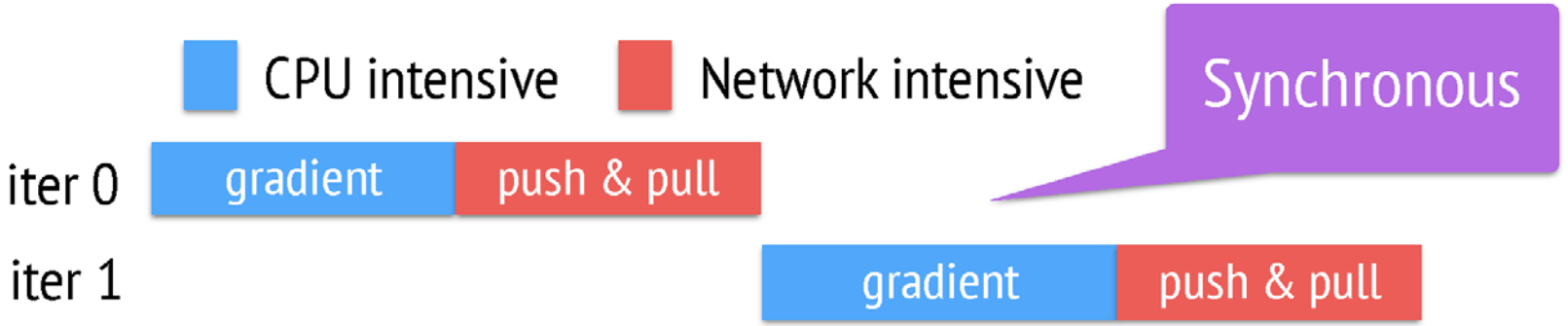
- a push / pull / user defined function (an iteration)

Task

- a push / pull / user defined function (an iteration)
- “execute-after-finished” dependency

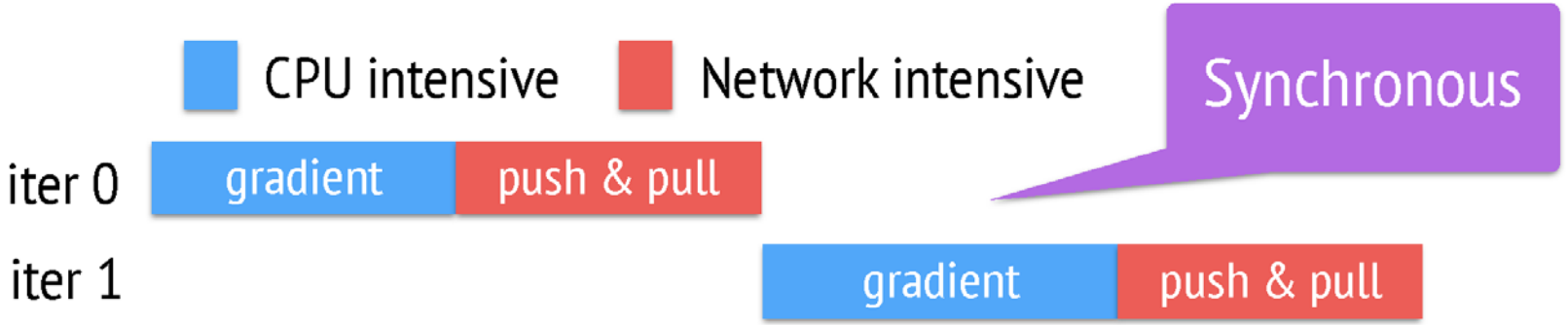
Task

- a push / pull / user defined function (an iteration)
- “execute-after-finished” dependency



Task

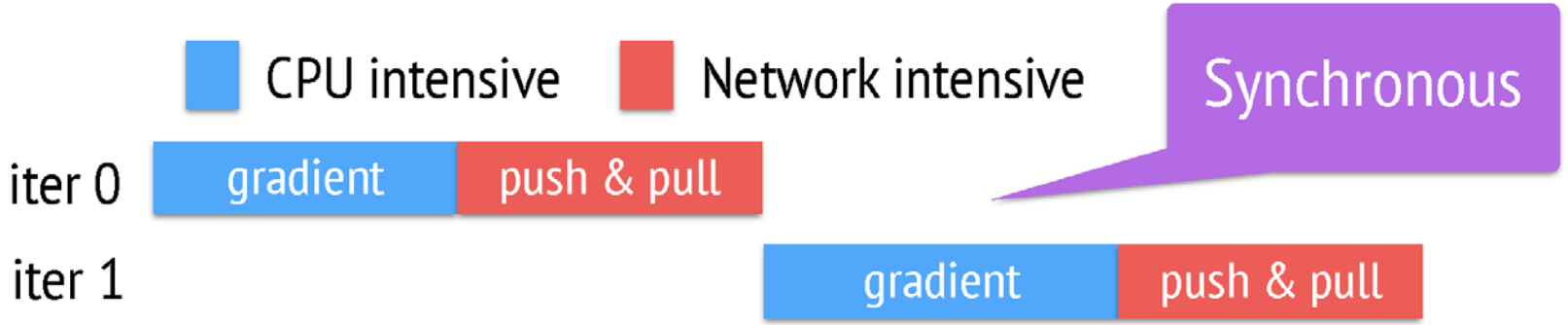
- a push / pull / user defined function (an iteration)
- “execute-after-finished” dependency



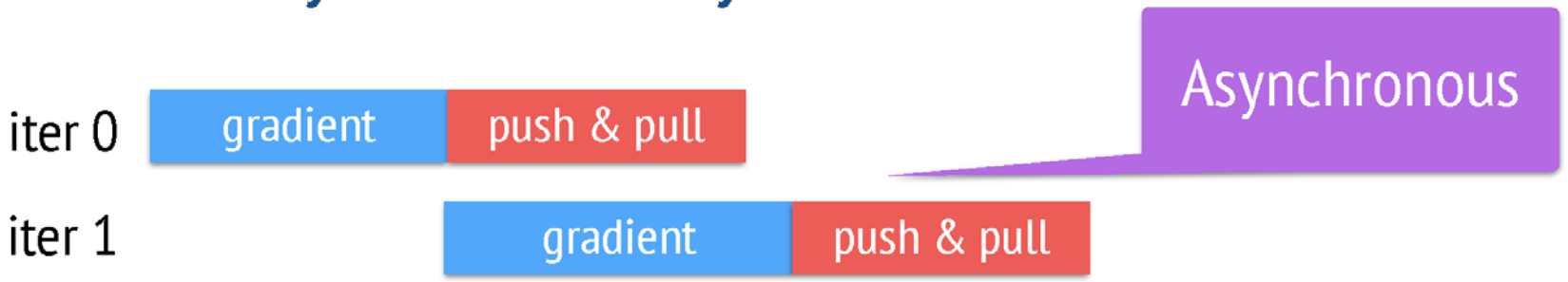
- executed asynchronously

Task

- a push / pull / user defined function (an iteration)
- “execute-after-finished” dependency

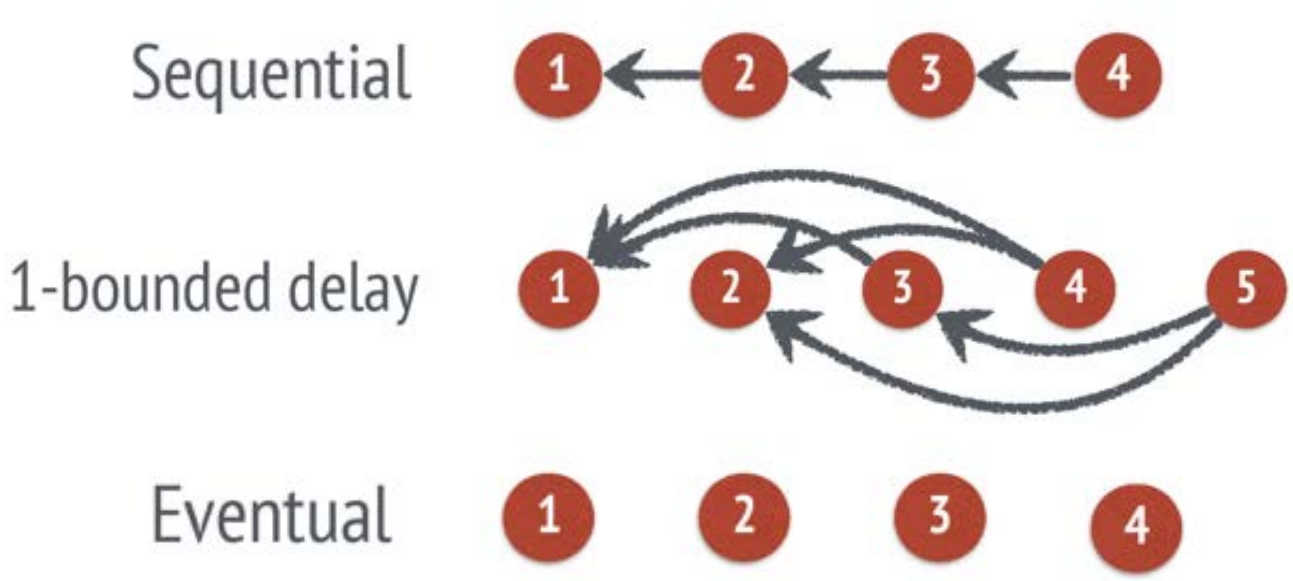


- executed asynchronously



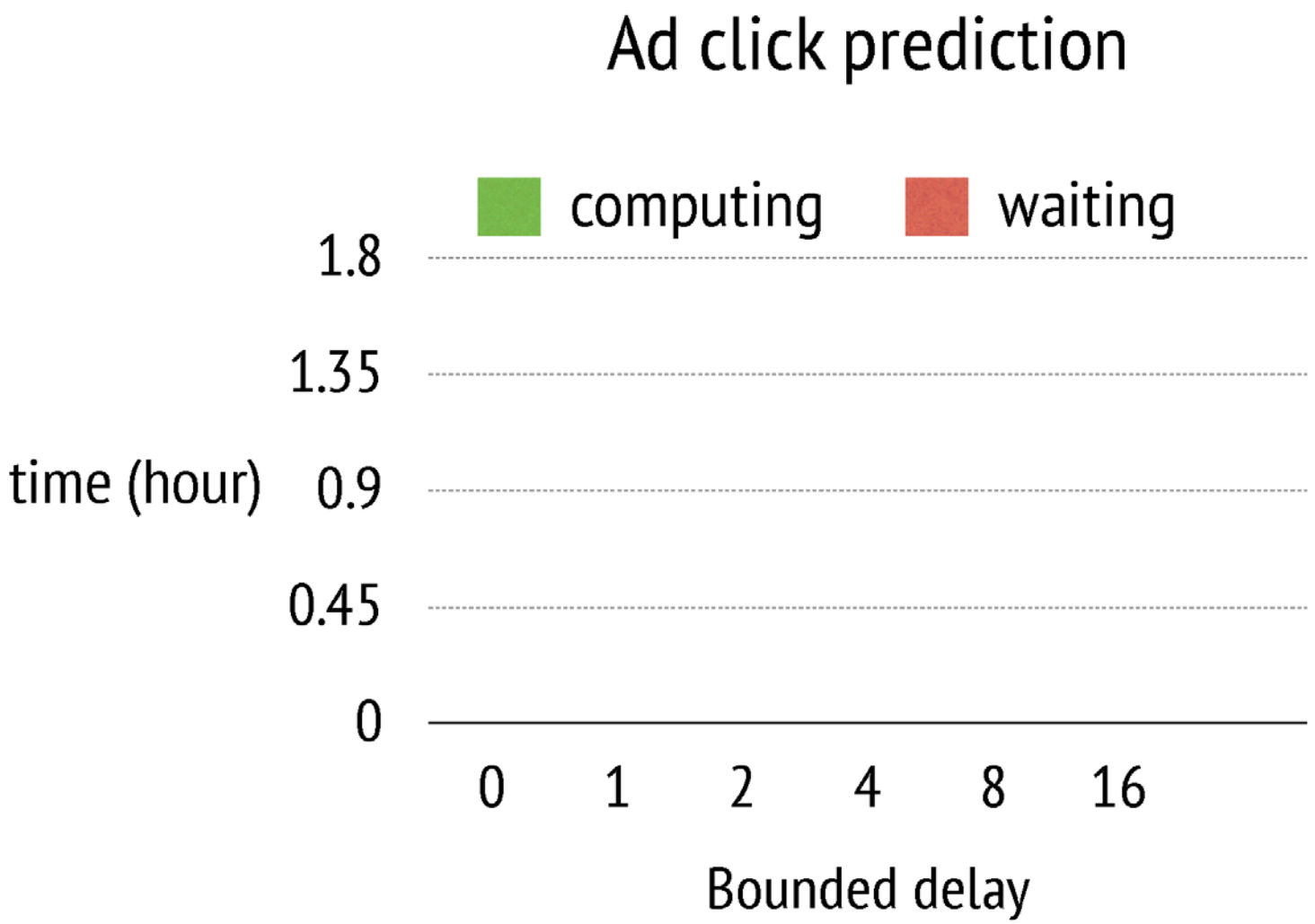
Architecture: Flexible Consistency

- Can change the consistency model for the system, as per the requirements of the job
- Up to the algorithm designer to choose the flexible consistency model
- Trade-off between **Algorithm Efficiency** and **System** Performance

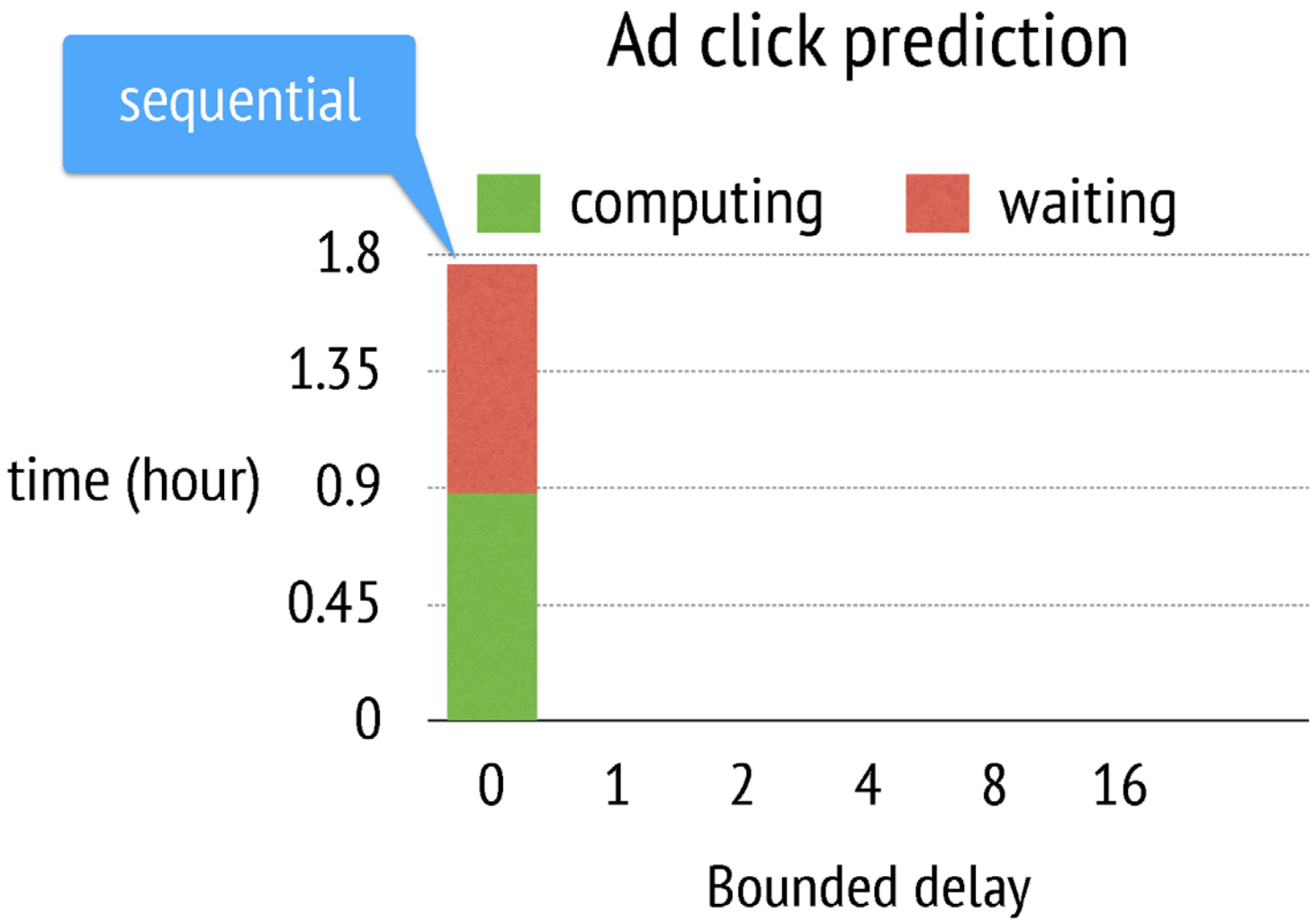


Results for bounded delay

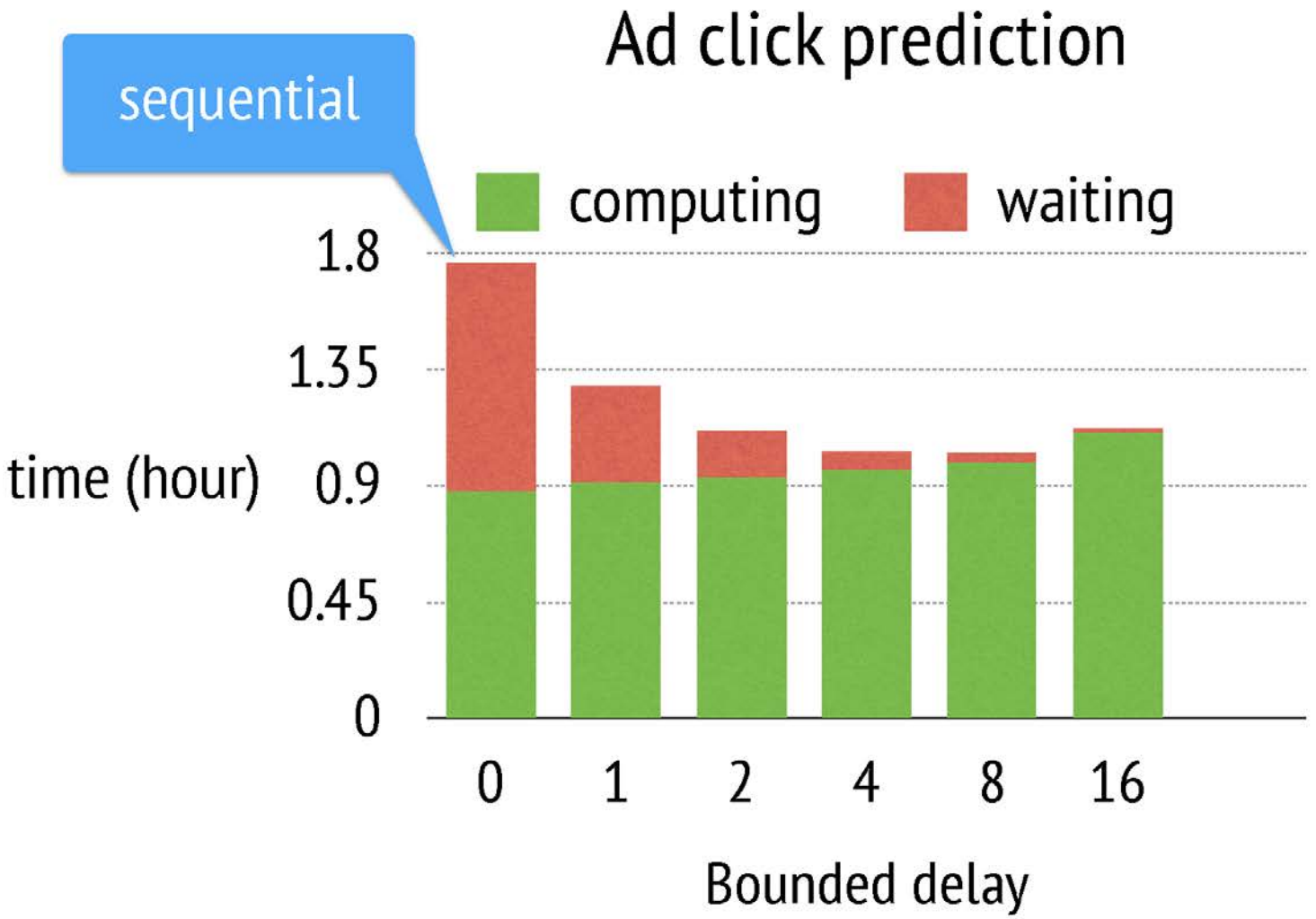
Results for bounded delay



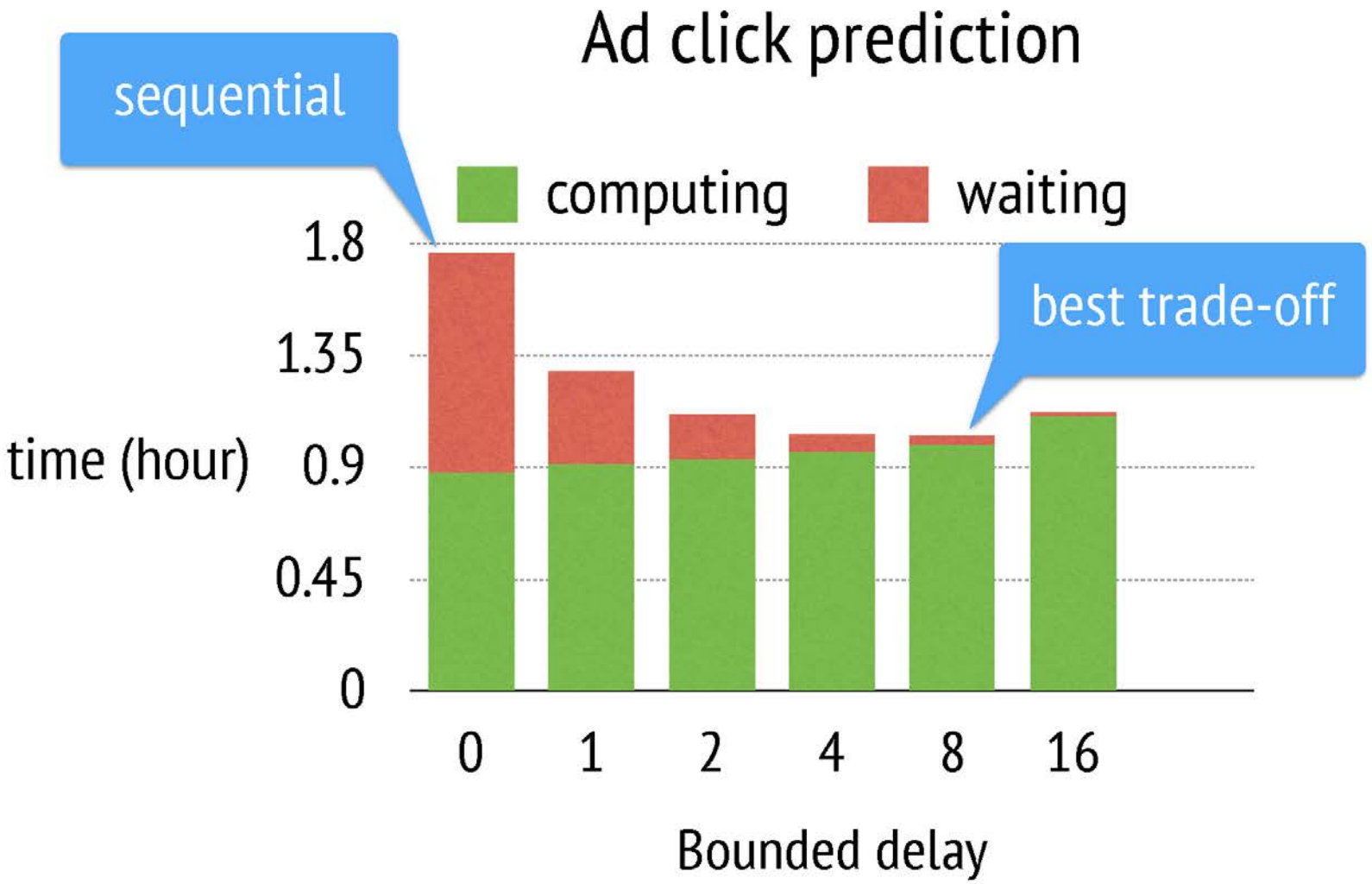
Results for bounded delay



Results for bounded delay



Results for bounded delay



Architecture: User Defined Filters

- Selectively Synchronize (key, value) pairs.
- Filters can be placed at either or both the *Server* machines and *Worker* machines.
- Allows for fine-grained consistency control within a task
- Example: *Significantly modified filter*: Only pushes entries that have changed for more than an amount.

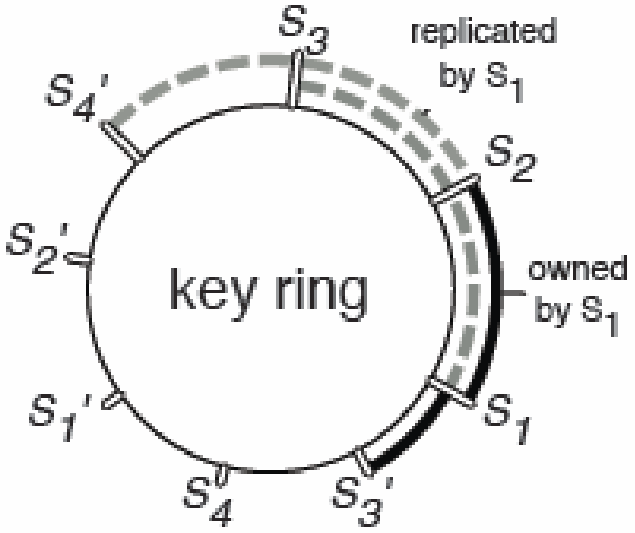
Implementation: Vector Clocks & Messaging

- Vector Clocks are attached for each (Key, value) pairs for several purposes:
  - Tracking Aggregation Status
  - Rejecting doubly sent data
  - Recovery from Failure
- As many (key, value) pairs get updated at the same time during one iteration, they can share the same clock stamps. This reduces the space requirements.
- Messages are sent in *Ranges* for efficient lookup and transfers.
- Messages are compressed using Google's Snappy compression library.

Implementation: Consistent Hashing & Replication

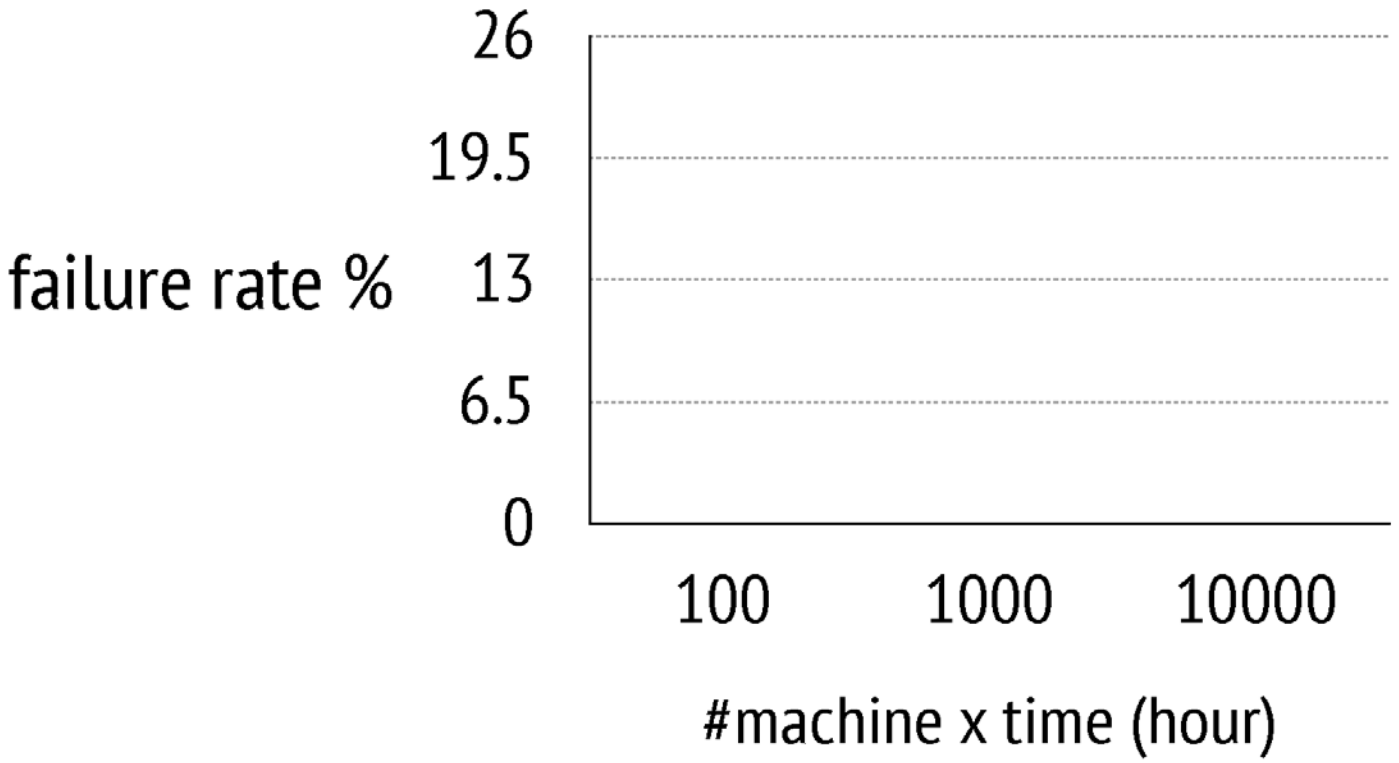
- The parameter server partitions the keys on to the Servers using *Ranged Partitioning*.

- The *Servers* are themselves hashed to a virtual ring similar to Chord.
- Server nodes store a replica of (Key, value) pairs in *k* nodes counter clockwise to it.

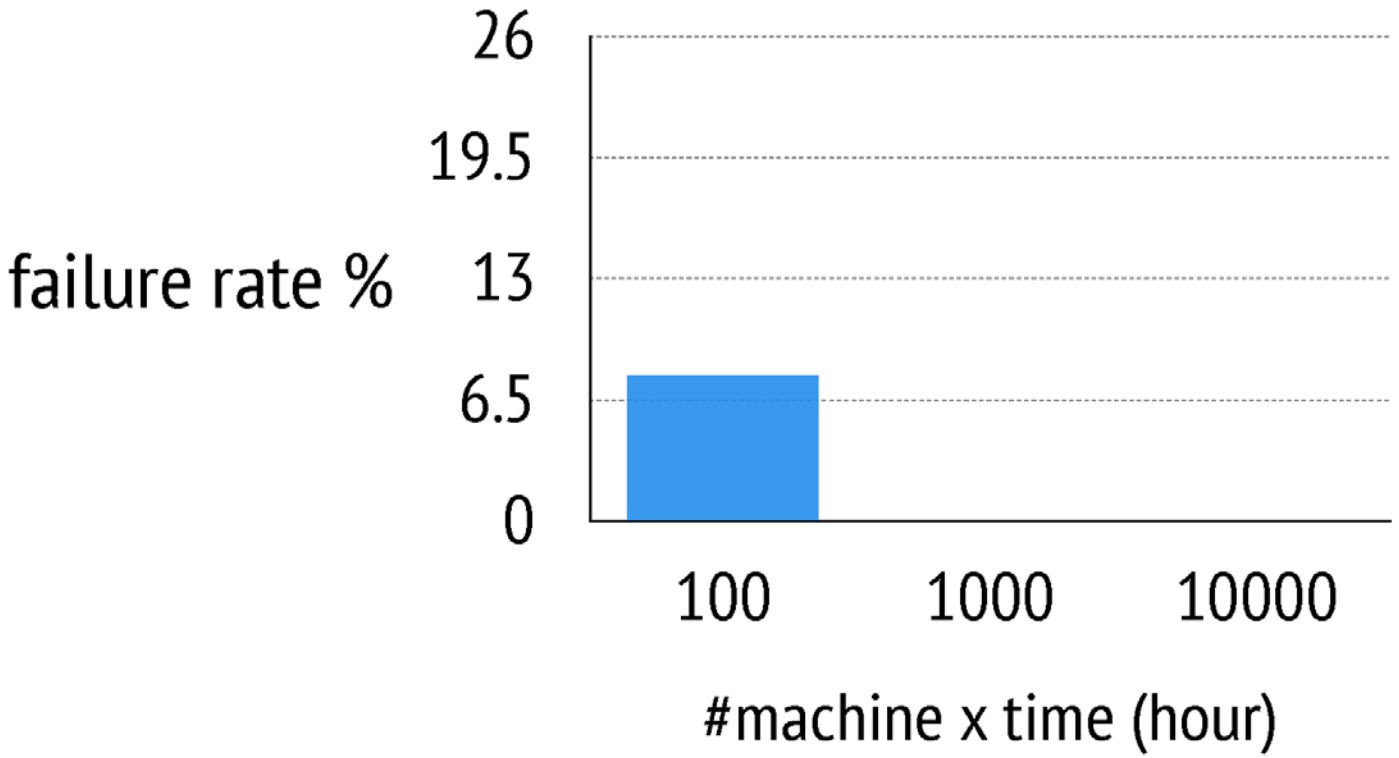


Machine learning job logs in a three-month period:

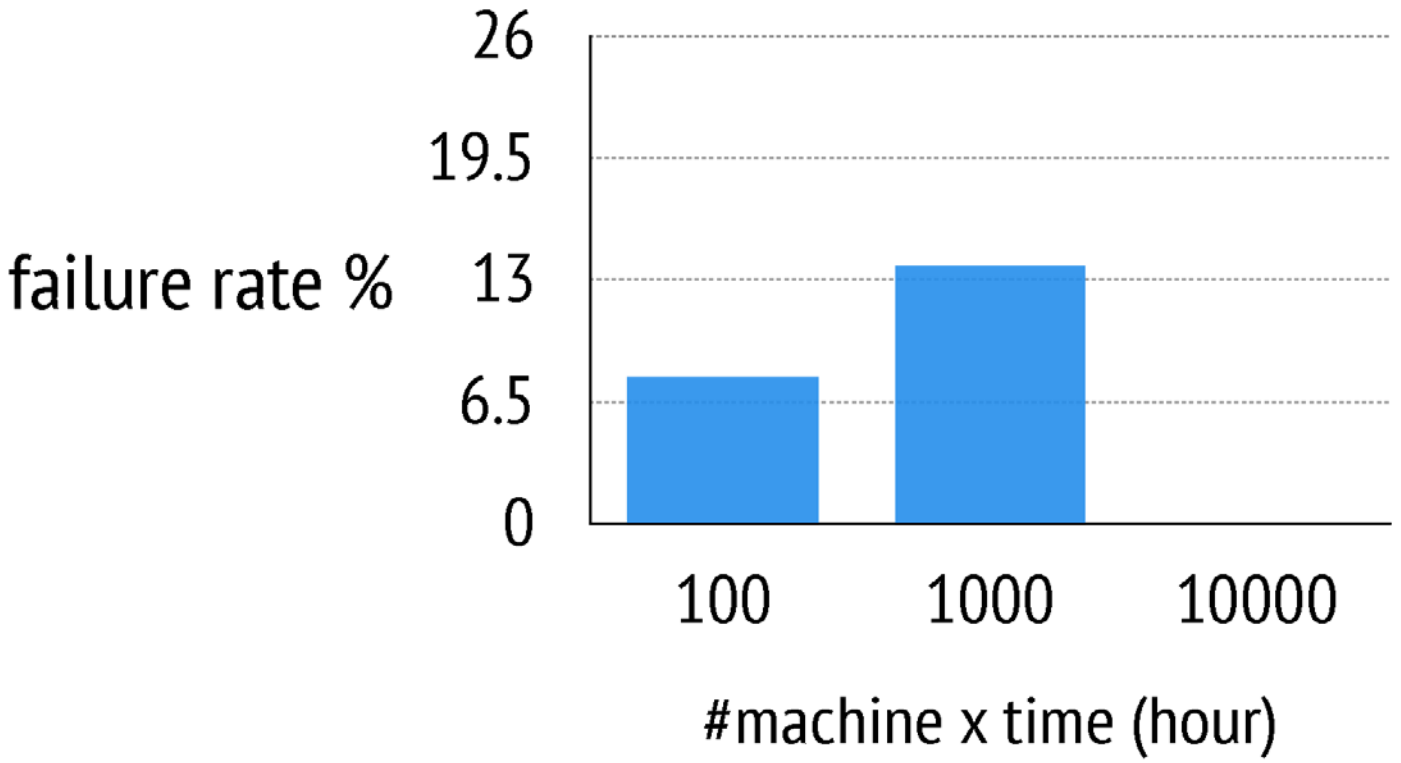
Machine learning job logs in a three-month period:



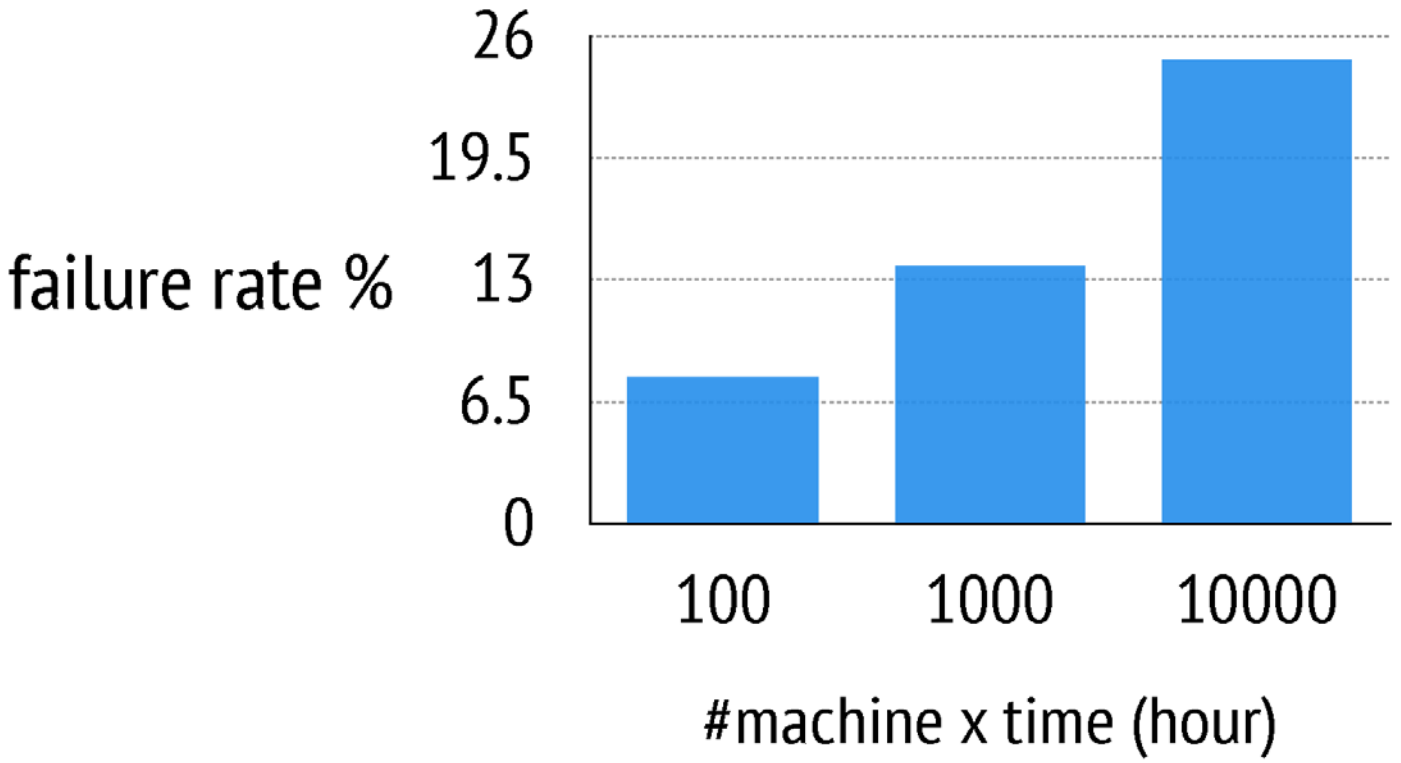
Machine learning job logs in a three-month period:



Machine learning job logs in a three-month period:



Machine learning job logs in a three-month period:



Fault tolerance

Fault tolerance

- Model is partitioned by consistent hashing

Fault tolerance

- Model is partitioned by consistent hashing
- Default replication: Chain replication (consistent, safe)

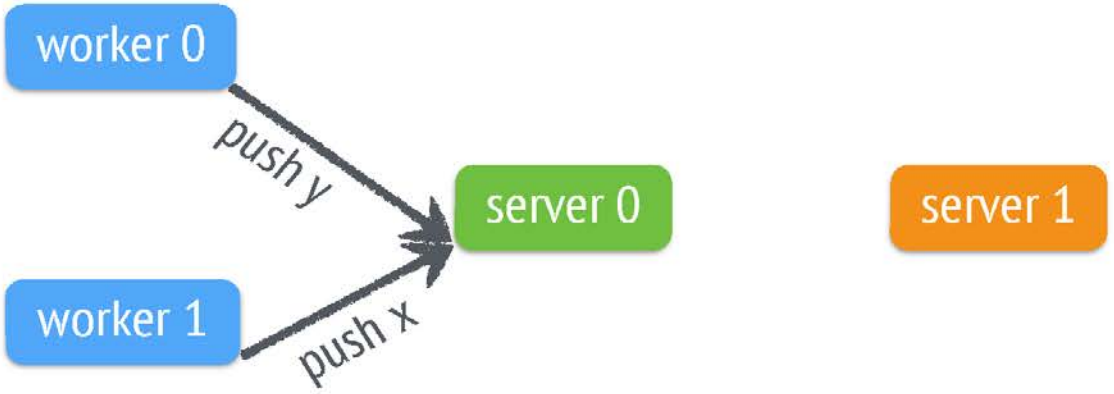
Fault tolerance

- Model is partitioned by consistent hashing
- Default replication: Chain replication (consistent, safe)

Fault tolerance

- Model is partitioned by consistent hashing
- Default replication: Chain replication (consistent, safe)

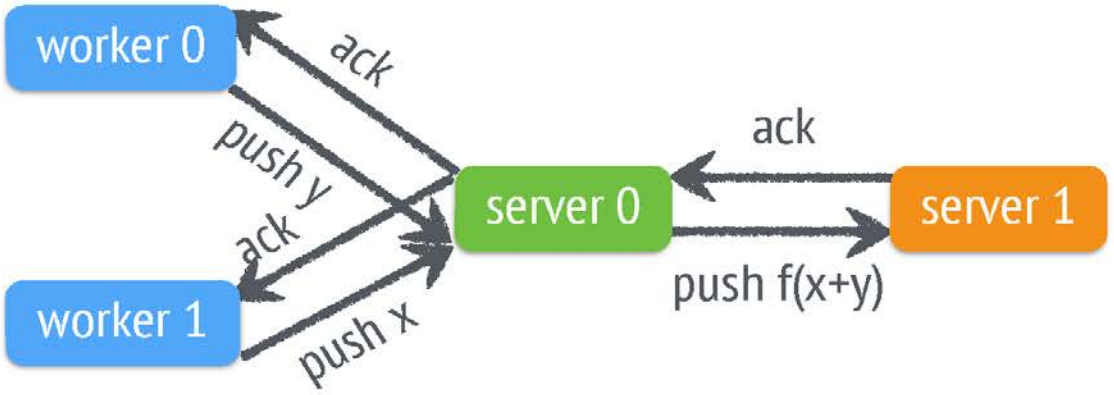
- Option: Aggregation reduces backup traffic (algo specific)



Fault tolerance

- Model is partitioned by consistent hashing
- Default replication: Chain replication (consistent, safe)

- Option: Aggregation reduces backup traffic (algo specific)

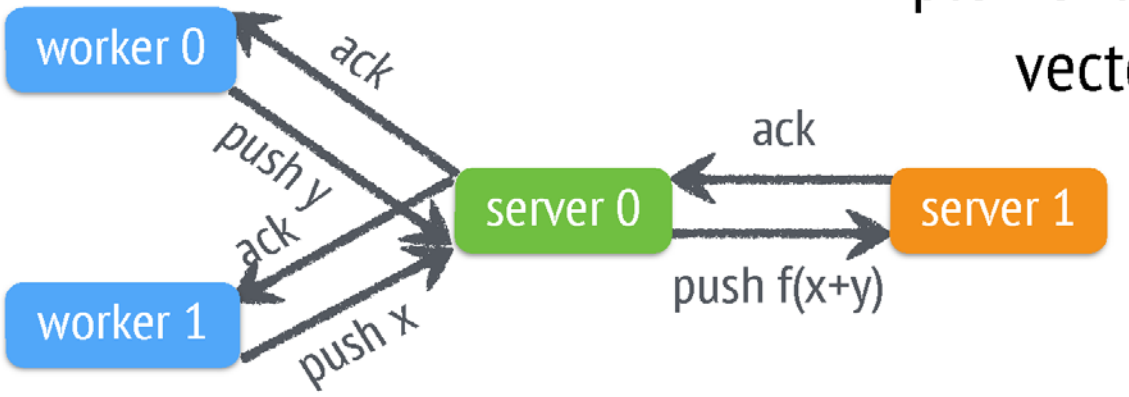


Fault tolerance

- Model is partitioned by consistent hashing
- Default replication: Chain replication (consistent, safe)

- Option: Aggregation reduces backup traffic (algo specific)

implemented by efficient vector clock



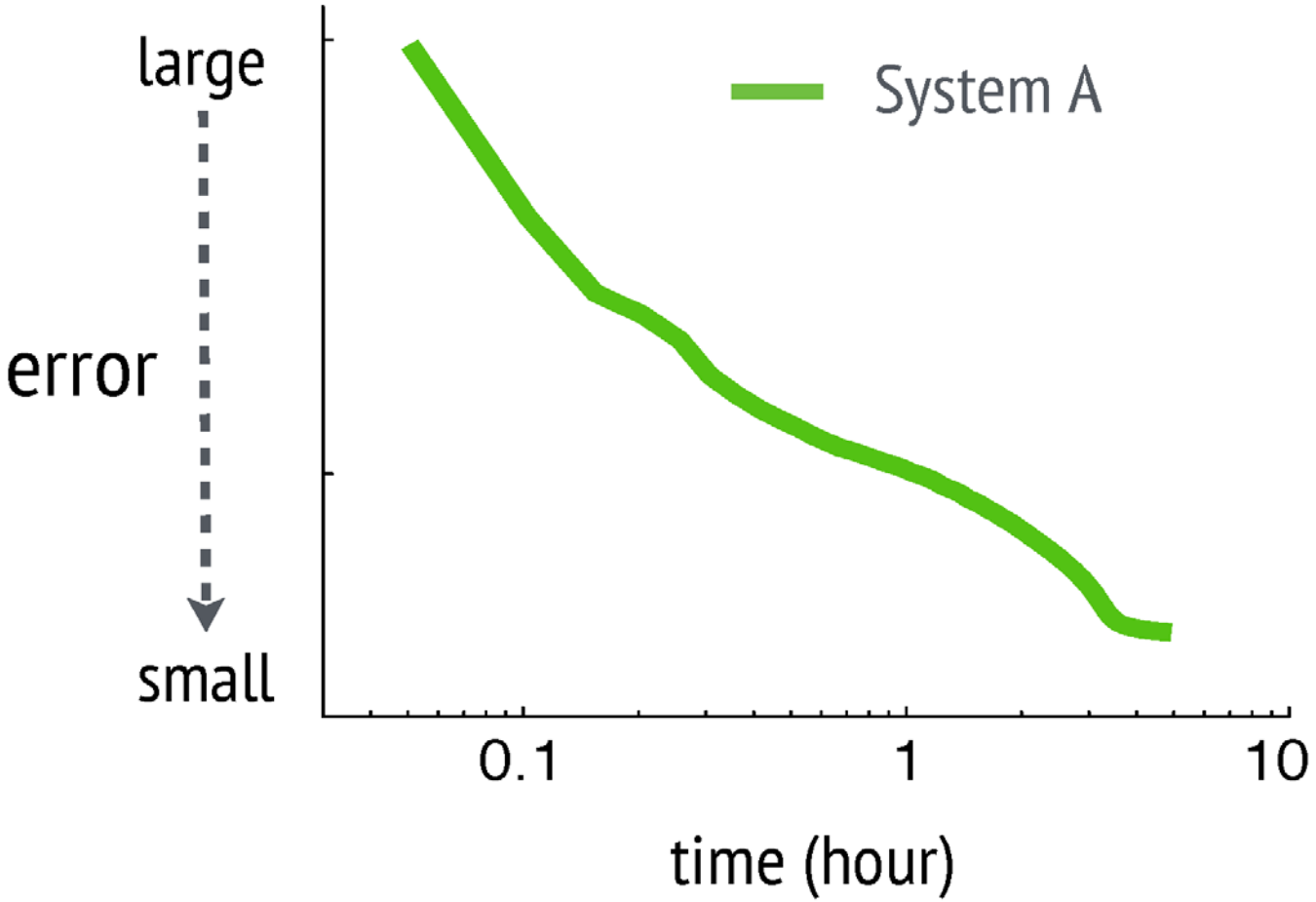
Evaluation

Evaluation: Sparse Logistic Regression

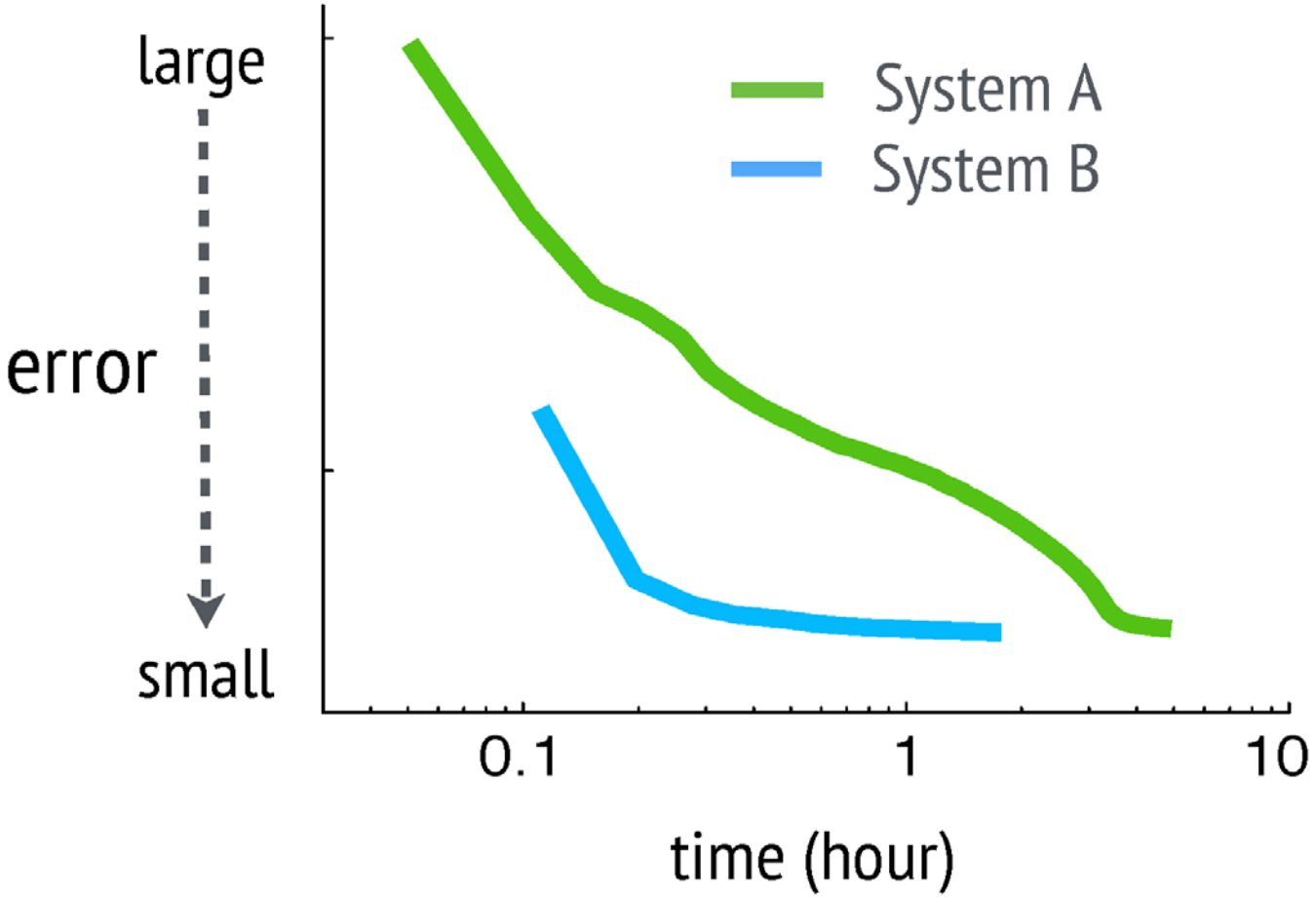
- One of the most popular large scale Risk Minimization algorithm.
- For example in the case of ads prediction, we want to predict the revenue an ad will generate.
- It can be done by running a logistic regression on the available data for ads which are 'close to' the ad we want to post.
- The experiment was run with:
  - 170 billion examples
  - 65 billion unique features
  - 636 TB of data in total
  - 1000 machines: 800 workers & 200 servers
  - Machines: 16 cores, 192 GB DRAM, and 10 Gb Ethernet links

	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

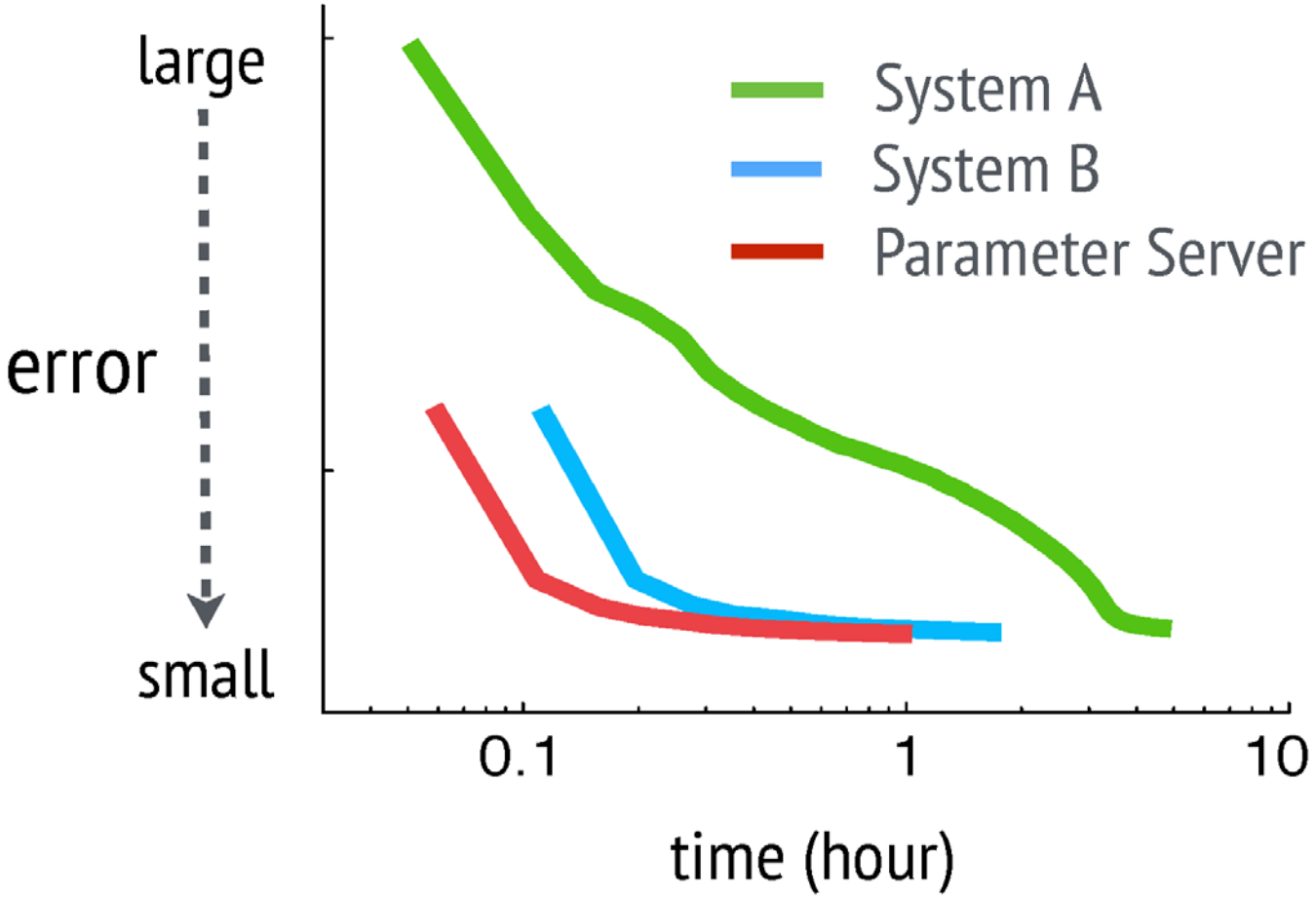
Progress



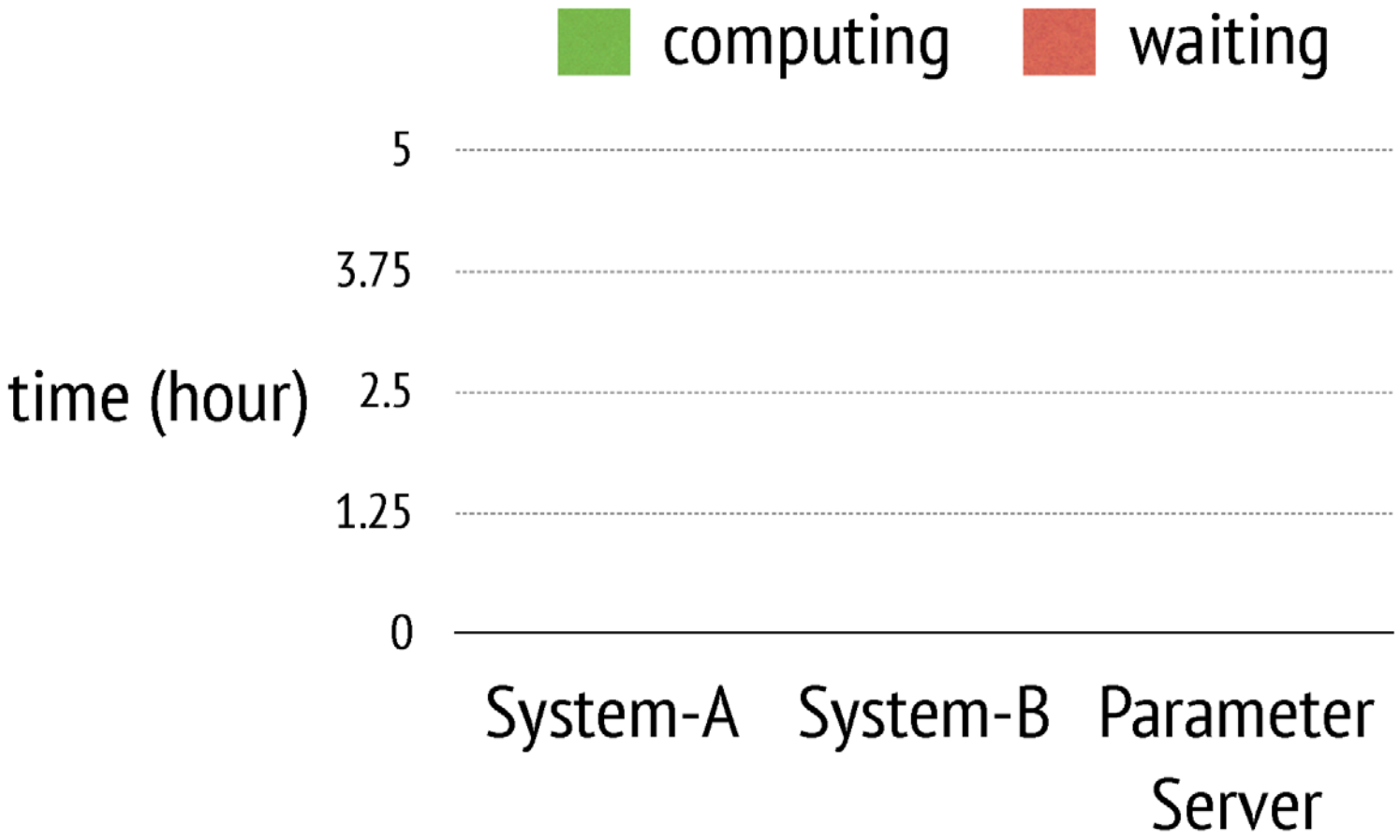
Progress



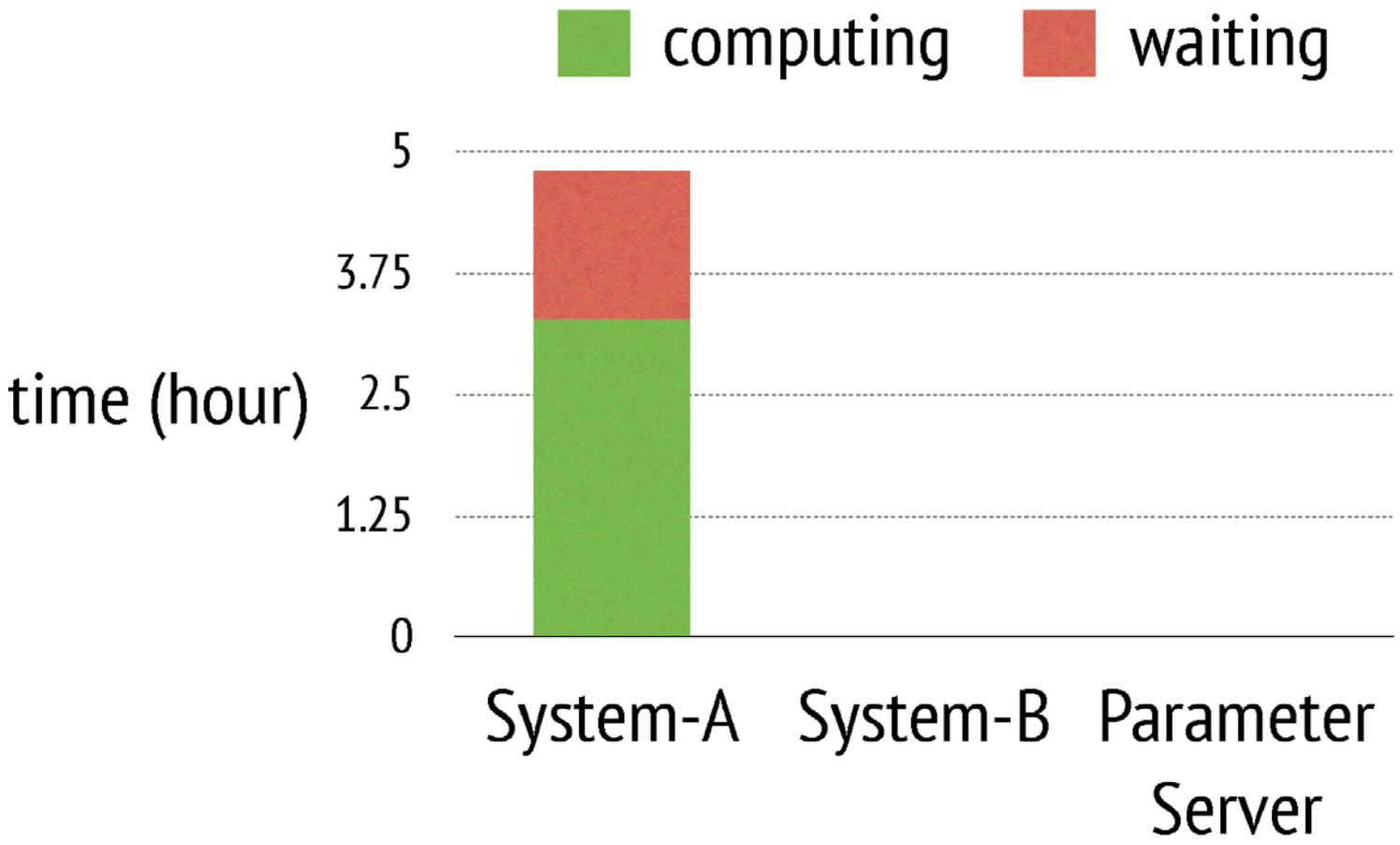
Progress


large
error
small
System A
System B
Parameter Server
0.1
1
10
time (hour)

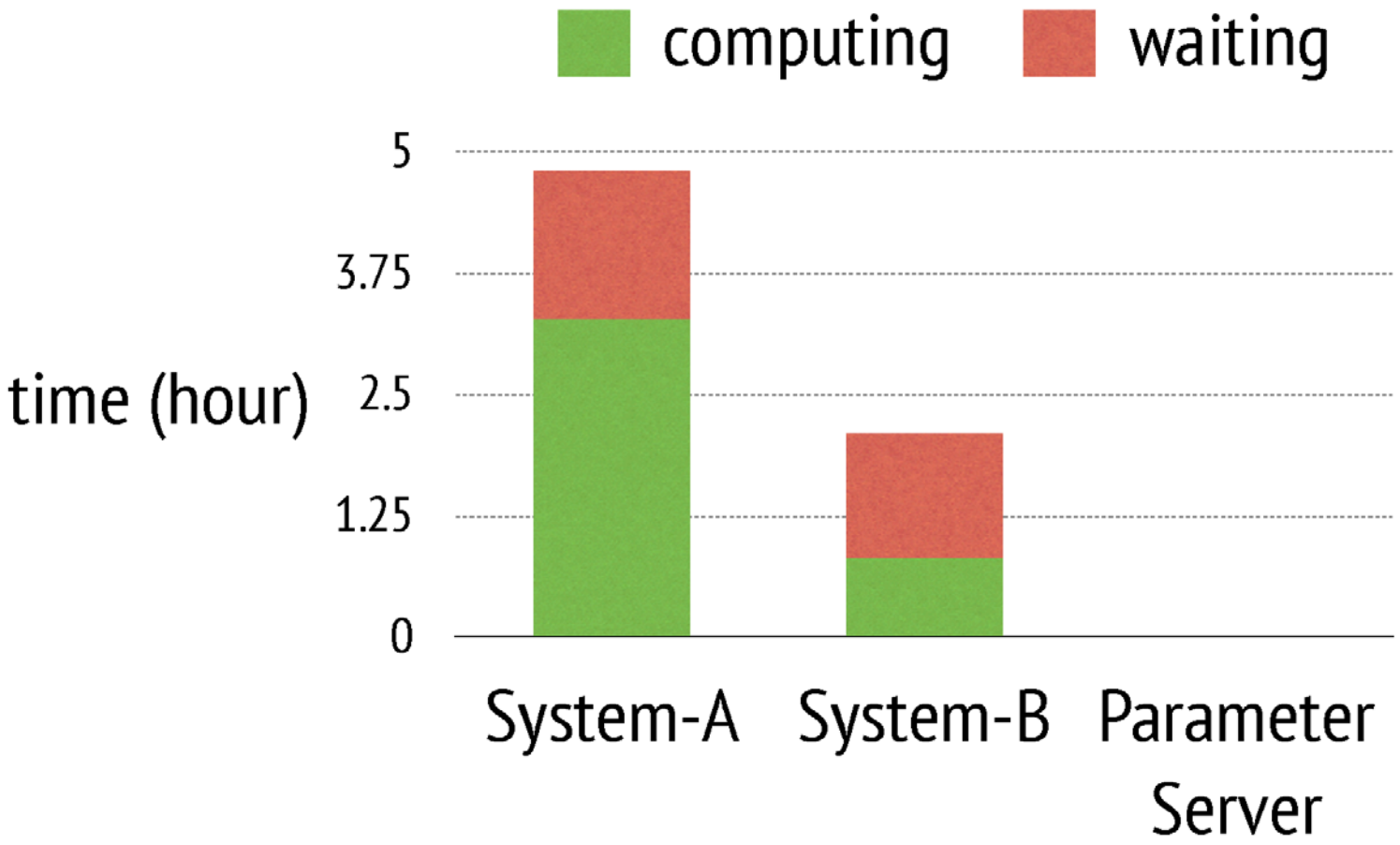
Time decomposition



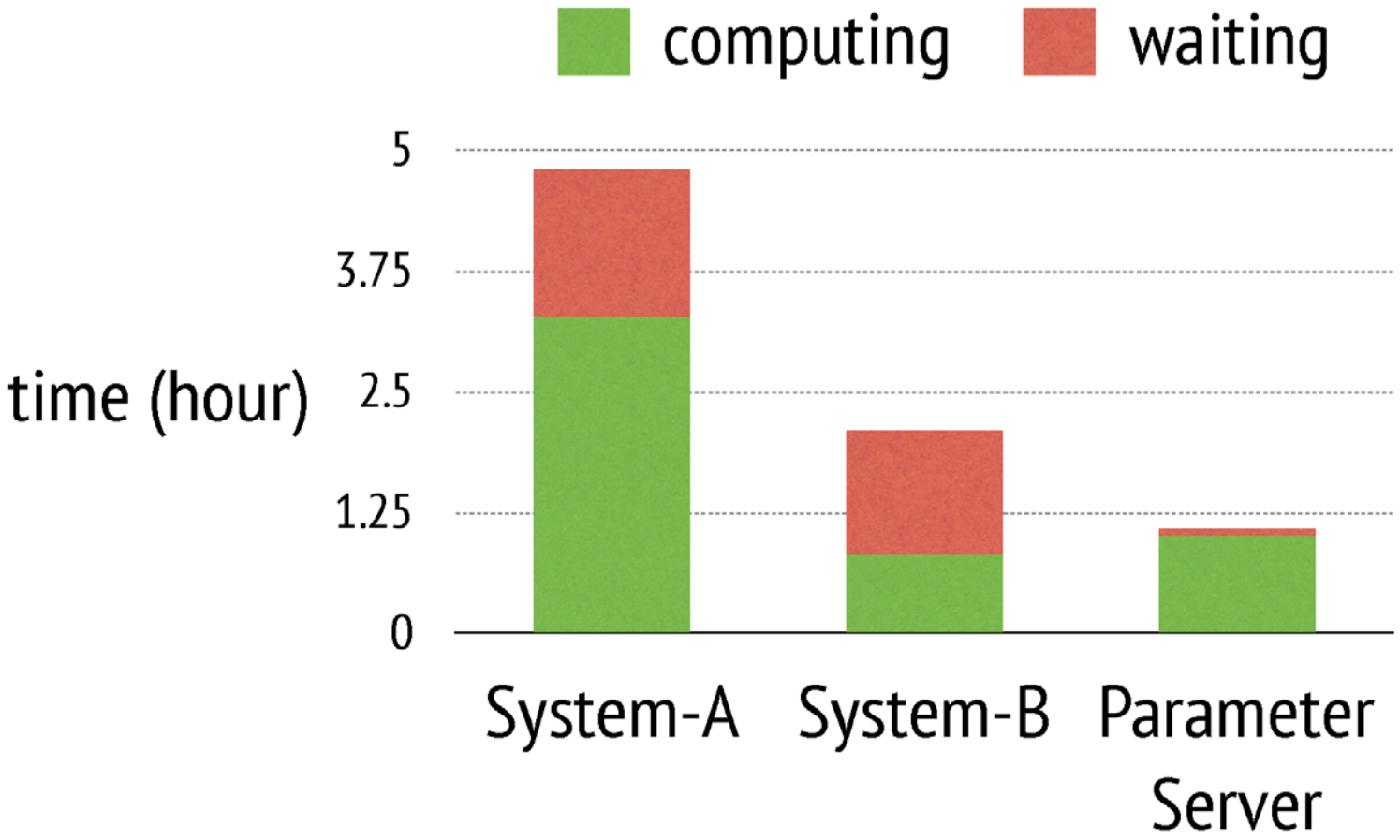
Time decomposition



Time decomposition



Time decomposition



Summary and Discussion

Summary: Pros

- **Efficient Communication**:
  - Batching (key,value) pairs in Linear Algebra objects
  - Filters to reduce unnecessary communication & message compression
  - Caching keys at worker and server nodes for local access
- **Flexible Consistency Models**:
  - Can choose between Sequential, Eventual, and Bounded delay consistency models
  - Allows for tradeoffs between System Performance and Algorithmic Convergence
- **Fault Tolerance and Durability**:
  - Replication of data in Servers
  - Failed workers can restart at the point of failure by using vector clocks
- **Ease of Use**:
  - Linear Algebra objects allow for intuitive implementation of tasks

Summary: Cons & Further Discussion

- What are System A and System B? No insight into design differences.
- *Server Manager Failures* and *Task Scheduler* failures are not discussed.
- No experiments on the other two systems with *Bounded delay* model. System B's waiting time may reduce if implemented with a Bounded Delay model.