High Performance Computing

10th Lecture

NOVEMBER 4, 2016
RIO YOKOTA LAB.
HIROKI NAGANUMA

Selected Paper

2014 IEEE International Conference on Big Data

Large-scale Logistic Regression and Linear Support Vector Machines Using Spark

Chieh-Yen Lin
Dept. of Computer Science
National Taiwan Univ., Taiwan
r01944006@csie.ntu.edu.tw

Cheng-Hao Tsai
Dept. of Computer Science
National Taiwan Univ., Taiwan
r01922025@csie.ntu.edu.tw

Ching-Pei Lee *
Dept. of Computer Science
Univ. of Illinois, USA
clee149@illinois.edu

Chih-Jen Lin
Dept. of Computer Science
National Taiwan Univ., Taiwan
cjlin@csie.ntu.edu.tw

- Published in:
  IEEE International Conference on Big Data, 2014

- Date of Conferrence:
  October 27-30, 2014

- http://www.csie.ntu.edu.tw/~cjlin/papers/spark-liblinear/spark-liblinear.pdf

Abstract

- Logistic regression and linear SVM are useful methods for large-scale classification.
  However, their distributed implementations have not been well studied.

Abstract

- Logistic regression and linear SVM are useful methods for large-scale classification.
  However, their distributed implementations have not been well studied.

- Recently, because of the inefficiency of the MapReduce framework on iterative algorithms, Spark, an in-memory cluster-computing platform, has been proposed.It has emerged as a popular framework for large-scale data processing and analytics.

Abstract

- Logistic regression and linear SVM are useful methods for large-scale classification.However, their distributed implementations have not been well studied.

- Recently, because of the inefficiency of the MapReduce framework on iterative algorithms, Spark, an in-memory cluster-computing platform, has been proposed.It has emerged as a popular framework for large-scale data processing and analytics.

- In this work,consider a distributed Newton method for solving logistic regression as well linear SVM and implement it on Spark.

Outline

1. Introduction
2. Approach
3. Implementation design
4. Related Works
5. Discussions and Conclusions

Outline

1. Introduction

2. Approach

3. Implementation design

4. Related Works

5. Discussions and Conclusions

Linear Classification on One Computer

- Linear classification on one machine is a mature technique: millions of data can be trained in a few seconds.

Linear Classification on One Computer

- Linear classification on one machine is a mature technique: millions of data can be trained in a few seconds.

- What if the data are even bigger than the capacity of our machine?

Linear Classification on One Computer

- Linear classification on one machine is a mature technique: millions of data can be trained in a few seconds.

- What if the data are even bigger than the capacity of our machine?

Solution 1: get a machine with larger memory/disk.
-> The data loading time would be too lengthy.

Linear Classification on One Computer

- Linear classification on one machine is a mature technique: millions of data can be trained in a few seconds.

- What if the data are even bigger than the capacity of our machine?

Solution 1: get a machine with larger memory/disk.
-> The data loading time would be too lengthy.

Solution 2: distributed training.

Distributed Linear Classification

- In distributed training, data loaded in parallel to reduce the I/O time. (e.g. HDFS)

Distributed Linear Classification

- In distributed training, data loaded in parallel to reduce the I/O time. (e.g. HDFS)

- With more machines, computation is faster.

Distributed Linear Classification

- In distributed training, data loaded in parallel to reduce the I/O time. (e.g. HDFS)

- With more machines, computation is faster.

- But communication and synchronization cost become significant.

Distributed Linear Classification

- In distributed training, data loaded in parallel to reduce the I/O time. (e.g. HDFS)

- With more machines, computation is faster.

- But communication and synchronization cost become significant.

- To keep the training efficiency, we need to consider algorithms with less communication cost, and examine implementation details carefully.

Distributed Linear Classification on Apache Spark

- Train logistic regression (LR) and L2-loss linear support vector machine (SVM) models on Apache Spark (Zaharia et al., 2010).

Distributed Linear Classification on Apache Spark

- Train logistic regression (LR) and L2-loss linear support vector machine (SVM) models on Apache Spark (Zaharia et al., 2010).
- Why Spark?

> MPI (Snir and Otto, 1998) is efficient, but does not support fault tolerance.

Distributed Linear Classification on Apache Spark

- Train logistic regression (LR) and L2-loss linear support vector machine (SVM) models on Apache Spark (Zaharia et al., 2010).
- Why Spark?

> MPI (Snir and Otto, 1998) is efficient, but does not support fault tolerance.

> MapReduce (Dean and Ghemawat, 2008) supports fault tolerance, but is slow in communication.

Distributed Linear Classification on Apache Spark

- Why Spark?

MPI (Snir and Otto, 1998) is efficient, but does not support fault tolerance.

MapReduce (Dean and Ghemawat, 2008) supports fault tolerance, but is slow in communication.

Spark combines advantages of both frameworks.

Distributed Linear Classification on Apache Spark

- Why Spark?

MPI (Snir and Otto, 1998) is e cient, but does not support fault tolerance.

MapReduce (Dean and Ghemawat, 2008) supports fault tolerance, but is slow in communication.

Spark combines advantages of both frameworks.

Communications conducted in-memory.

Distributed Linear Classification on Apache Spark

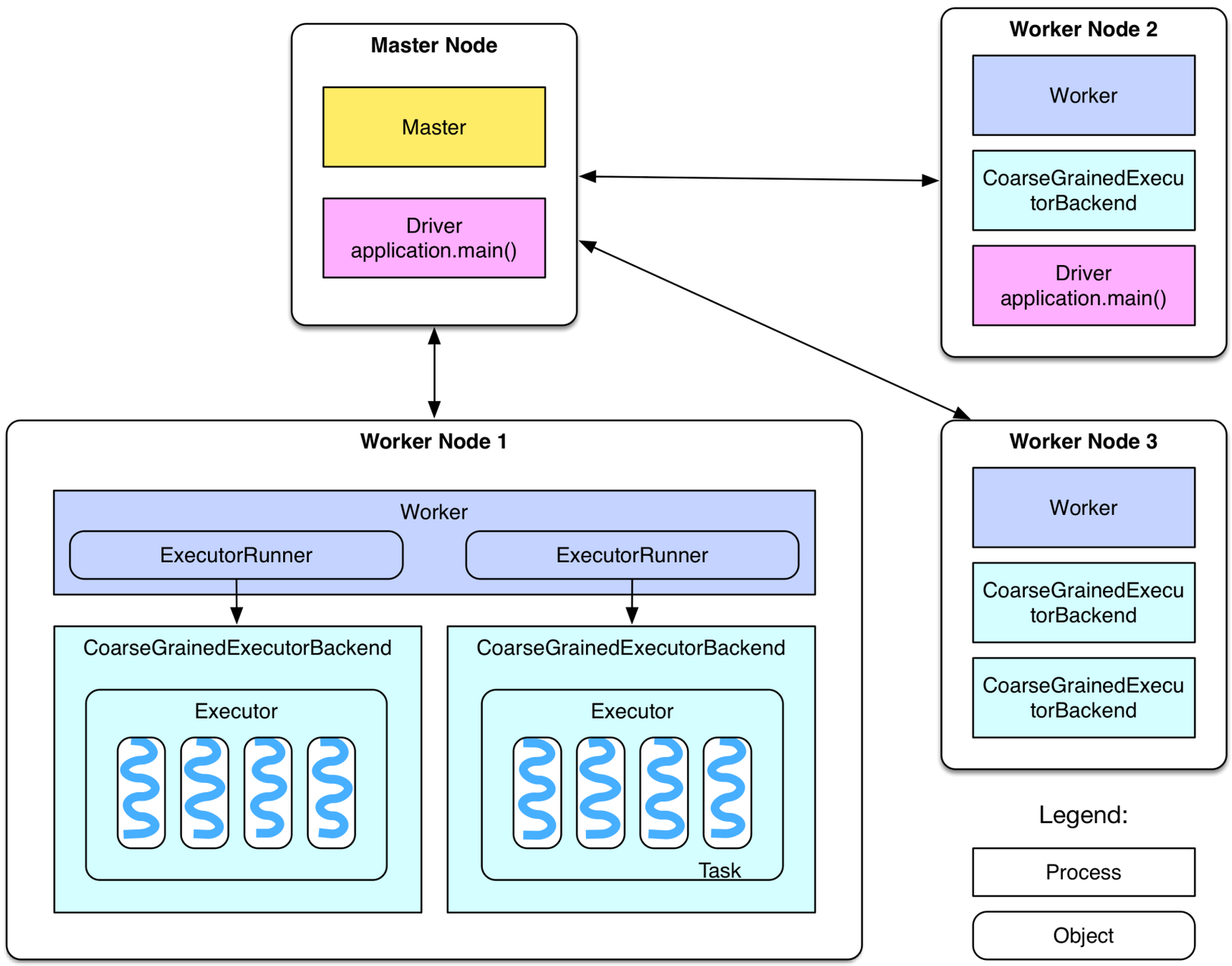
- Why Spark?

Spark combines advantages of both frameworks.
Communications conducted in-memory.
Supports fault tolerance.

- However, Spark is new and still under development. (2014)
- Therefore it is necessary to examine important implementation issues to ensure efficiency.

Apache Spark

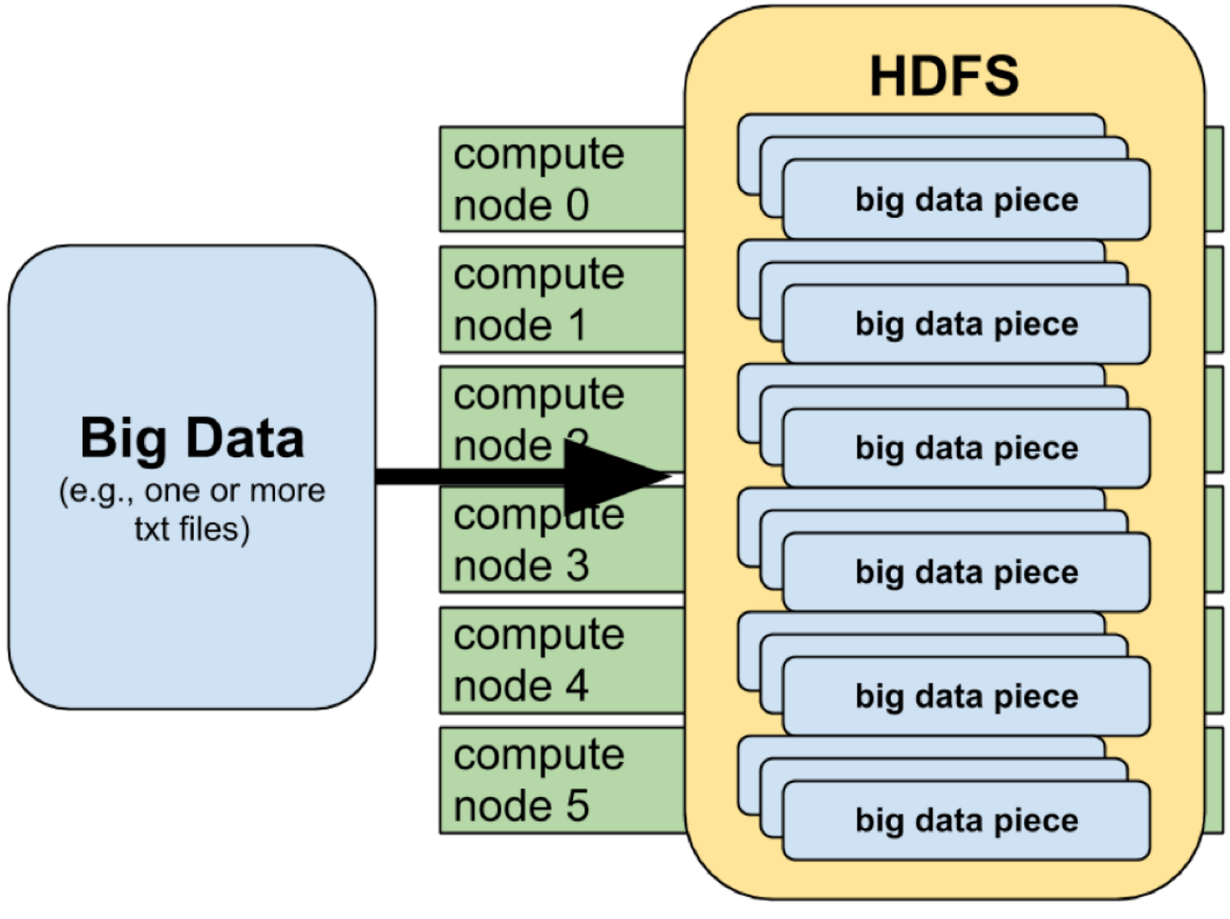
- Only the master-slave framework.



Apache Spark

- Only the master-slave framework.

- Data fault tolerance: Hadoop Distributed File System (Borthakur, 2008).



Apache Spark

- Only the master-slave framework.

- Data fault tolerance: Hadoop Distributed File System (Borthakur, 2008).

- Computation fault tolerance:

Apache Spark

- Only the master-slave framework.

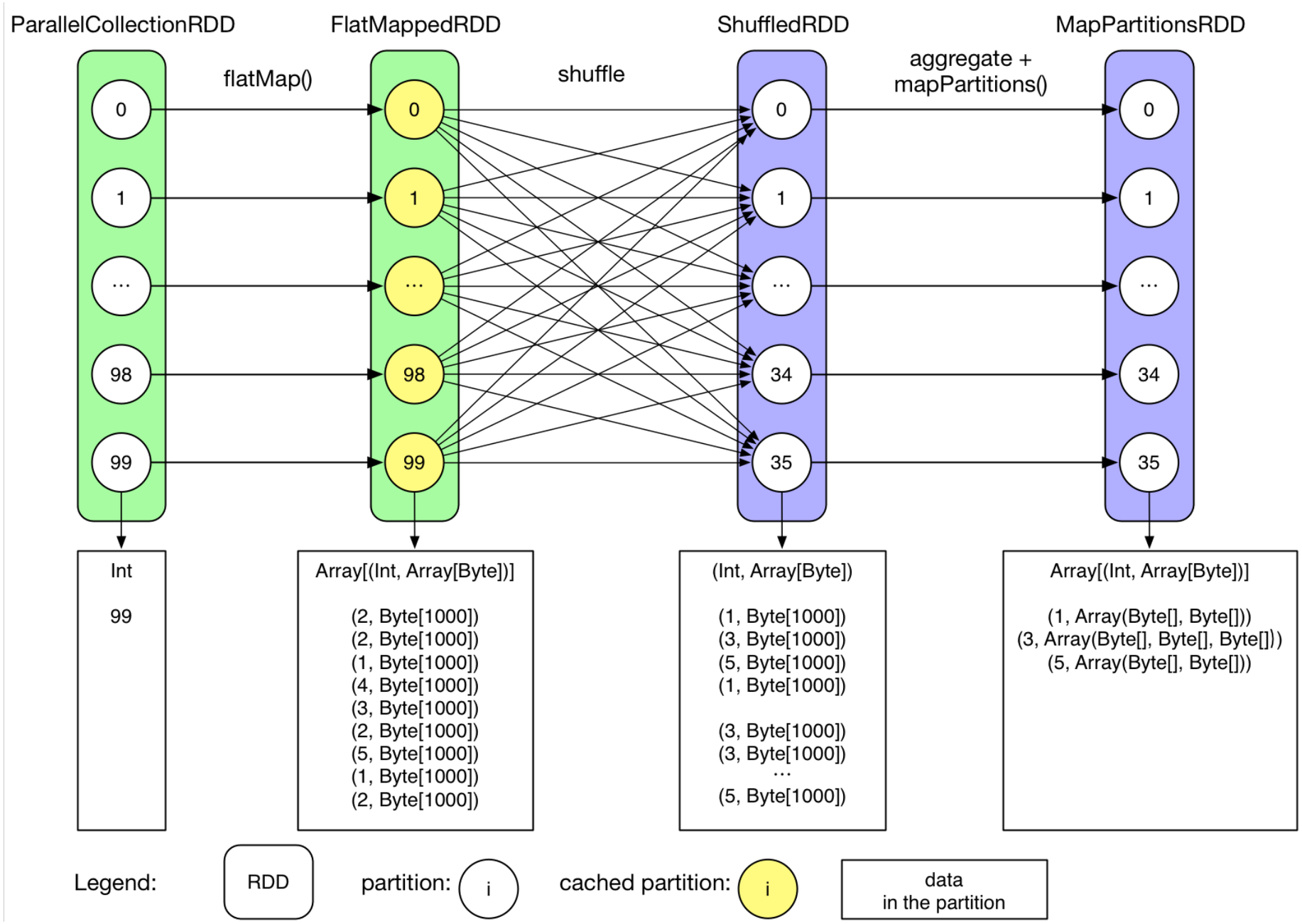
- Data fault tolerance: Hadoop Distributed File System (Borthakur, 2008).

- Computation fault tolerance: Read-only Resilient Distributed Datasets (RDD) and lineage (Zaharia et al., 2012).
  Basic idea: reconduct operations recorded in lineage on immutable RDDs.

Apache Spark

- Lineage (Zaharia et al., 2012).

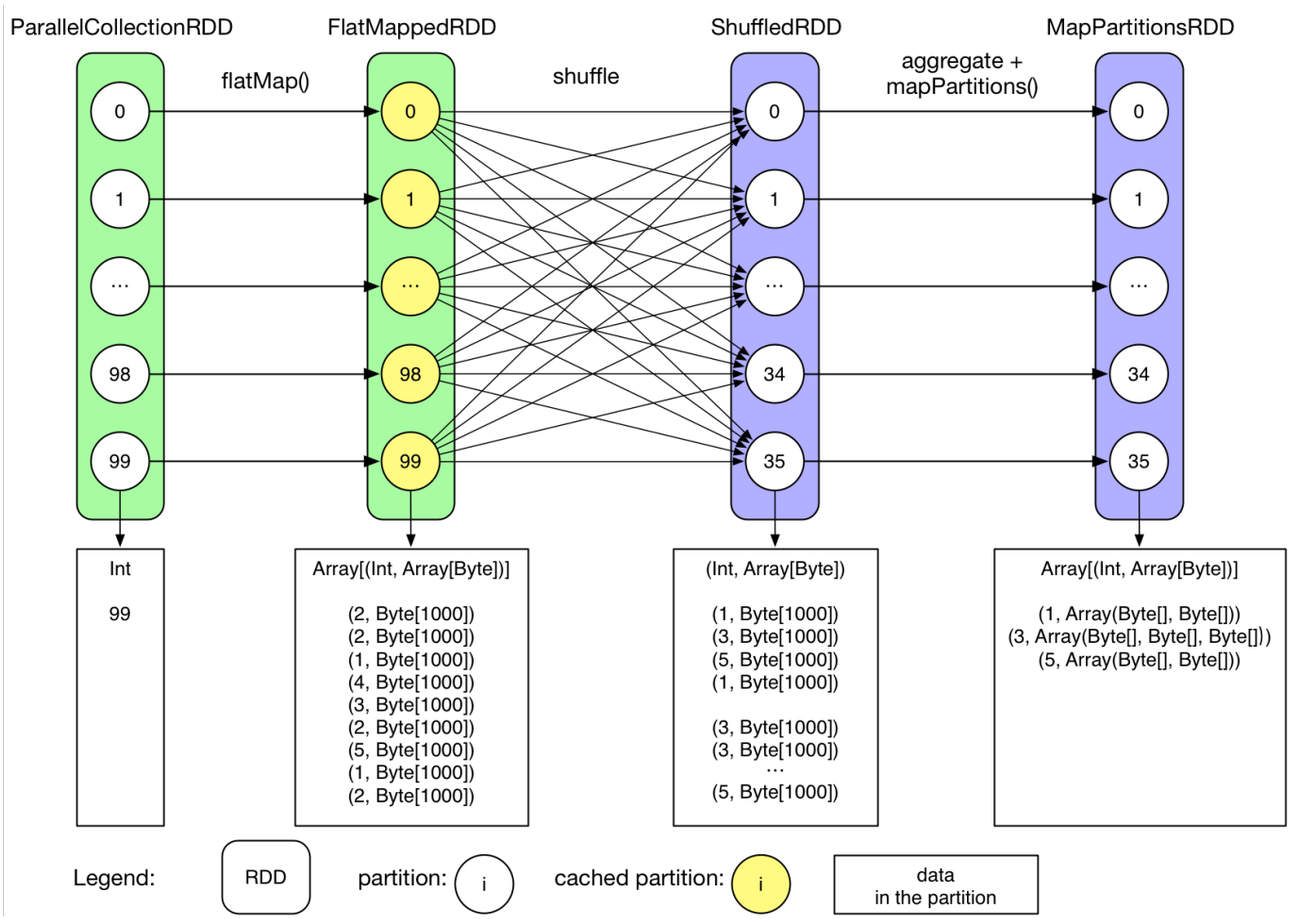
Spark firstly creates a logical plan (namely data dependency graph) for each application.

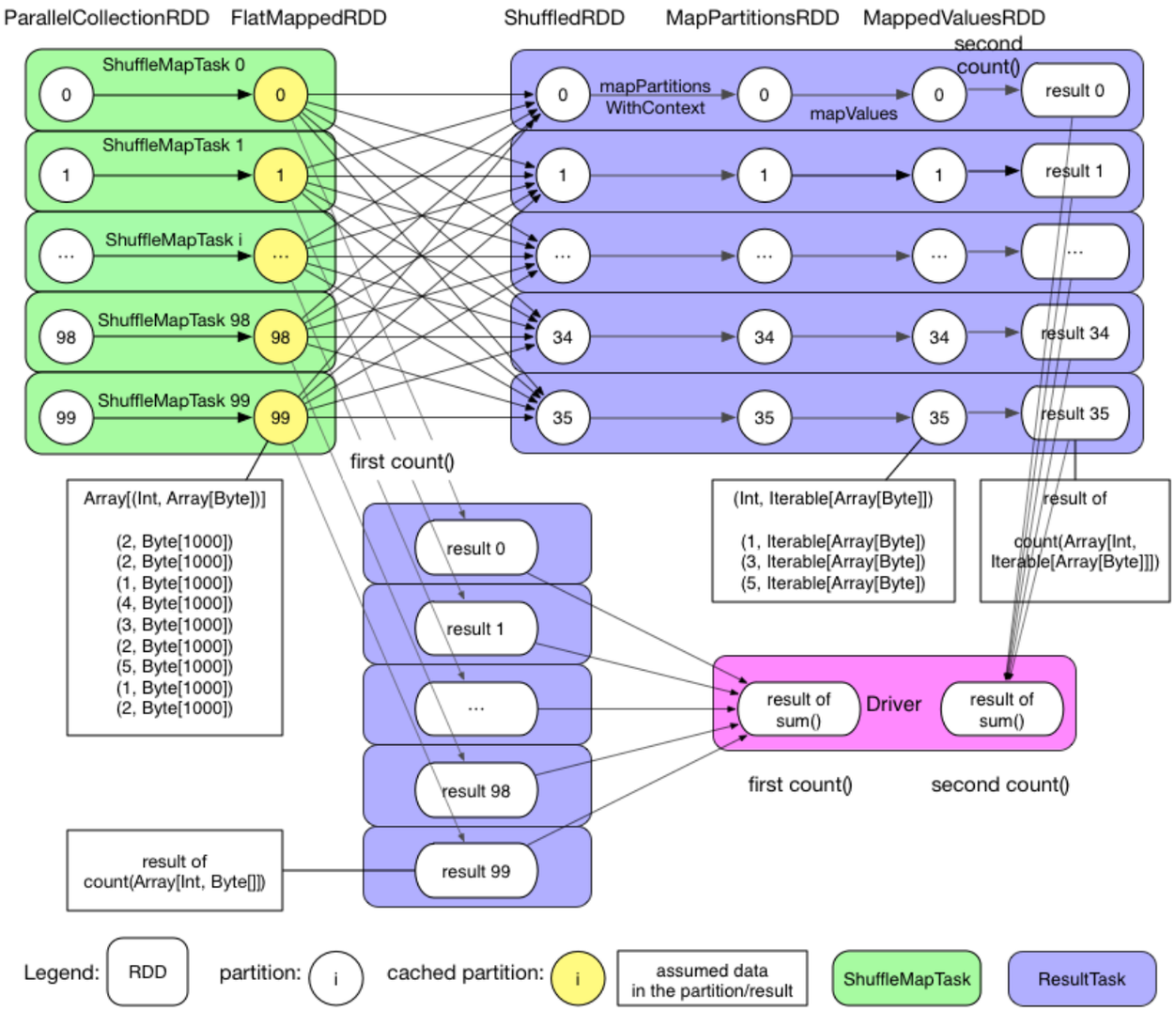


Apache Spark

- Lineage (Zaharia et al., 2012).

Then it transforms the logical plan into a physical plan (a DAG graph of map/reduce stages and map/reduce tasks).

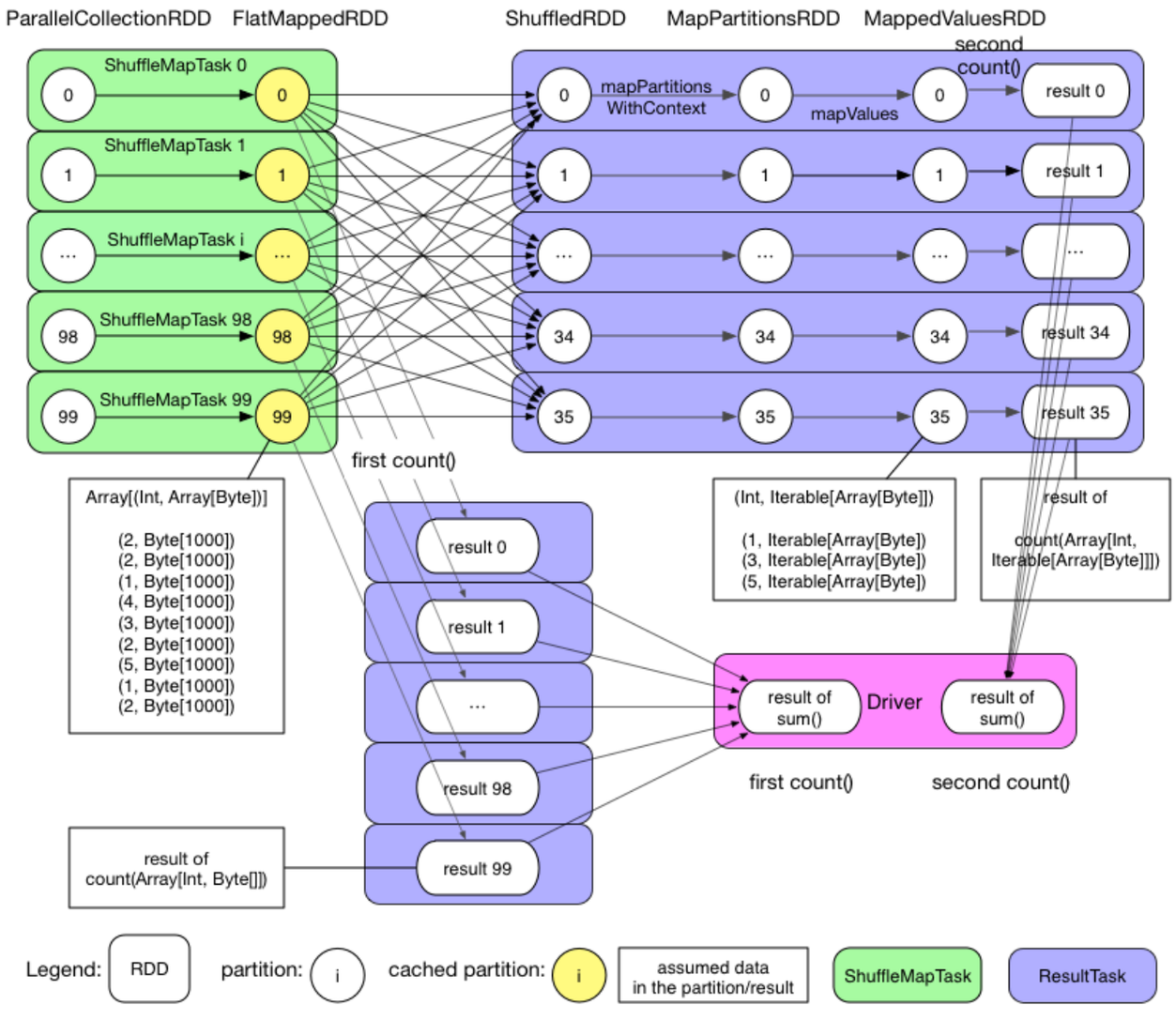


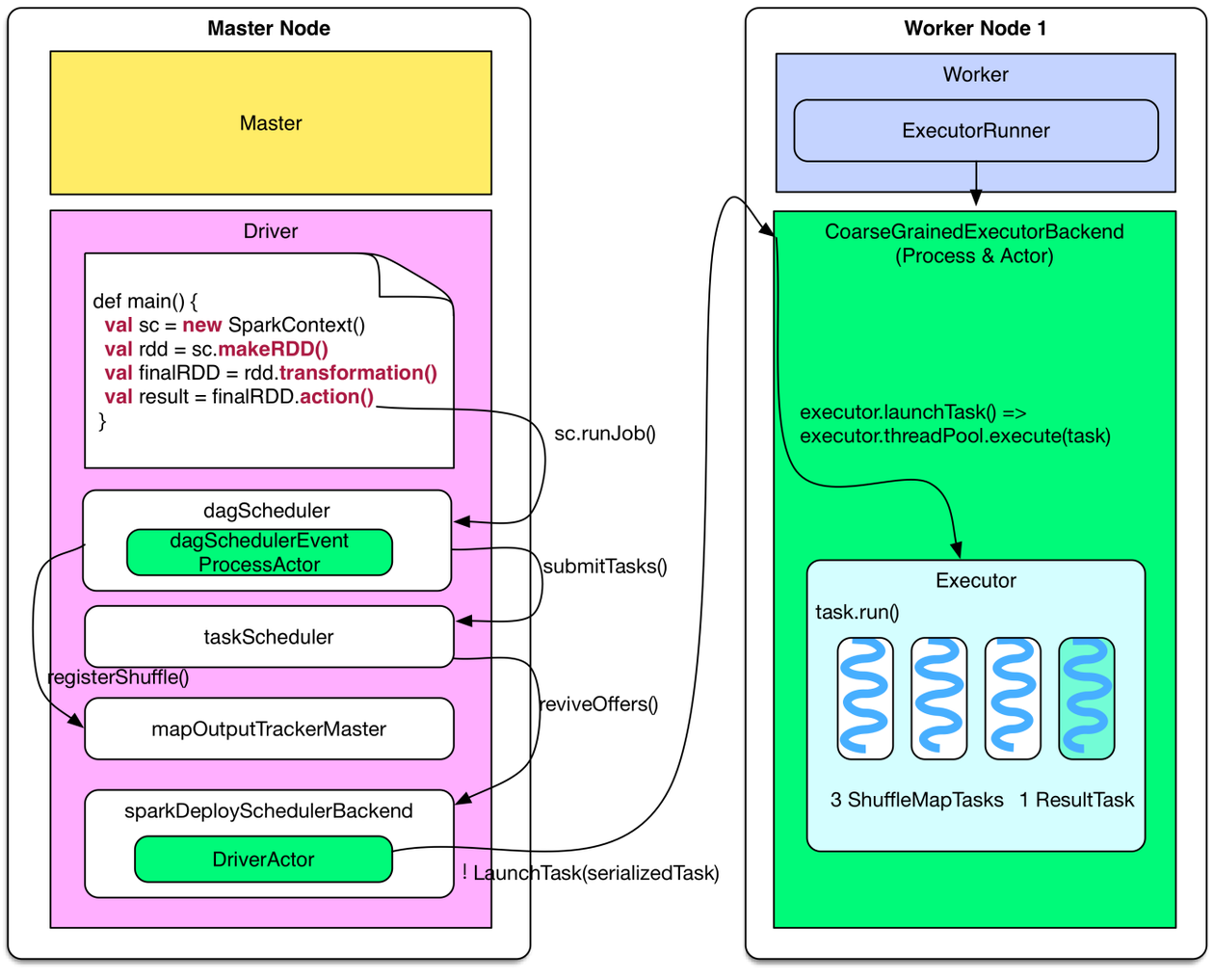


Apache Spark

- Lineage (Zaharia et al., 2012).

After that, concrete map/reduce tasks will be lanuched to process the input data.

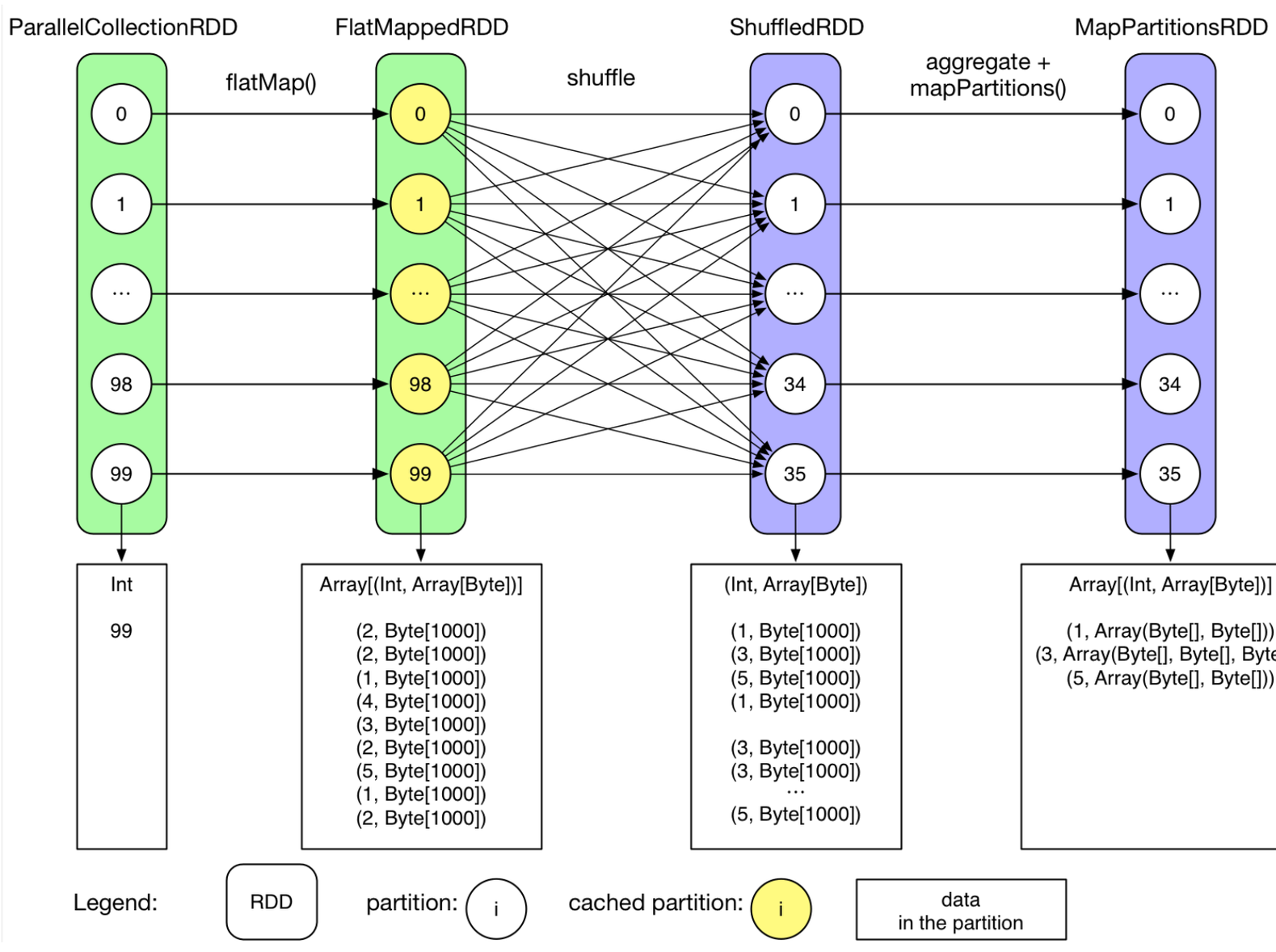




Apache Spark (skip)

- Lineage (Zaharia et al., 2012).

  spark code example



GroupByTest.scala

```
package org.apache.spark.examples
import java.util.Random
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._
/**
  * Usage: GroupByTest [numMappers] [numKVPairs]
[valSize] [numReducers]
  */
object GroupByTest {
  def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("GroupBy
Test")
    var numMappers = 100
    var numKVPairs = 10000
    var valSize = 1000
    var numReducers = 36
    val sc = new SparkContext(sparkConf)
    val pairs1 = sc.parallelize(0 until numMappers,
numMappers).flatMap { p =>
      val ranGen = new Random
      var arr1 = new Array[(Int, Array[Byte])]
(numKVPairs)
      for (i <- 0 until numKVPairs) {
        val byteArr = new Array[Byte](valSize)
        ranGen.nextBytes(byteArr)
        arr1(i) = (ranGen.nextInt(Int.MaxValue),
byteArr)
      }
      arr1
    }.cache
    // Enforce that everything has been calculated and
in cache
    pairs1.count
    println(pairs1.groupByKey(numReducers).count)
    sc.stop()
  }
}
```

Apache Spark (skip)

- Lineage (Zaharia et al., 2012).

  spark code explanation

[1]. Initialize SparkConf.
[2]. Initialize numMappers=100, numKVPairs=10,000, valSize=1000, numReducers=36.
[3]. Initialize SparkContext, which creates the necessary objects and actors for the driver.
[4].Each mapper creats an arr1: Array[(Int, Byte[])], which has numKVPairs elements. Each Int is a random integer, and each byte array's size is valSize. We can estimate Size(arr1) = numKVPairs * (4 + valSize) = 10MB, so that Size(pairs1) = numMappers * Size(arr1) =1000MB.

[5].Each mapper is instructed to cache its arr1 array into the memory.
[6].The action count() is applied to sum the number of elements in arr1 in all mappers, the result is numMappers * numKVPairs = 1,000,000. This action triggers the caching of arr1s.
[7].groupByKey operation is performed on cached pairs1. The reducer number (a.k.a., partition number) is numReducers. Theoretically, if hash(key) is evenly distributed, each reducer will receive numMappers * numKVPairs / numReducer = 27,777 pairs of (Int, Array[Byte]), with a size of Size(pairs1) / numReducer = 27MB.
[8].Reducer aggregates the records with the same Int key, the result is (Int, List(Byte[], Byte[], ..., Byte[])).
[9].Finally, a count() action sums up the record number in each reducer, the final result is actually the number of distinct integers in pairs1.

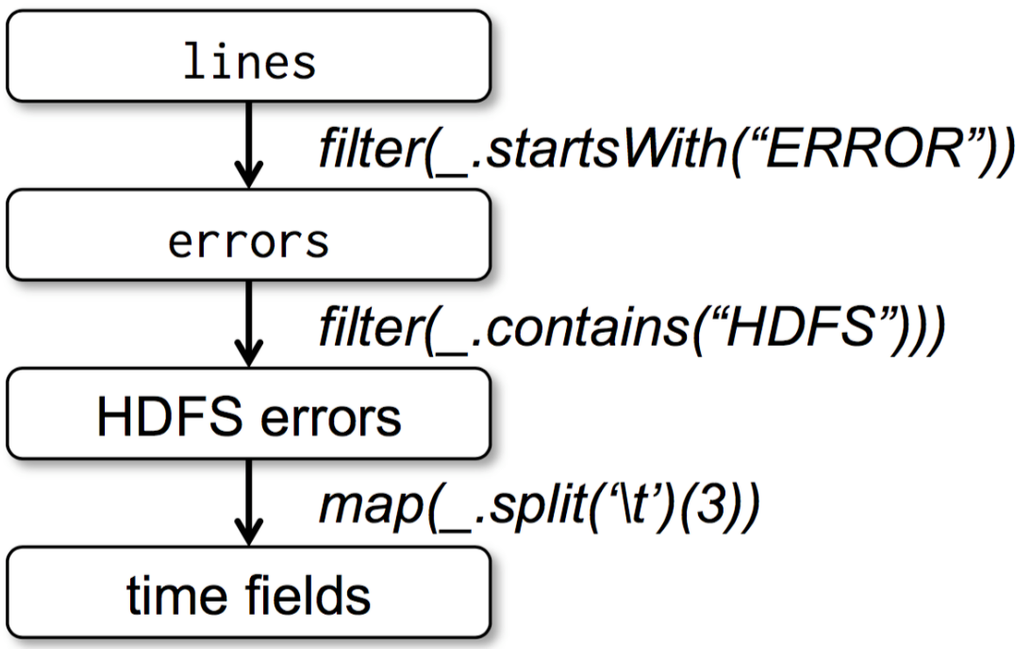
GroupByTest.scala

```
package org.apache.spark.examples
import java.util.Random
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.SparkContext._
/**
  * Usage: GroupByTest [numMappers] [numKVPairs]
[valSize] [numReducers]
  */
object GroupByTest {
  def main(args: Array[String]) {
    val sparkConf = new SparkConf().setAppName("GroupBy
Test")
    var numMappers = 100
    var numKVPairs = 10000
    var valSize = 1000
    var numReducers = 36
    val sc = new SparkContext(sparkConf)
    val pairs1 = sc.parallelize(0 until numMappers,
numMappers).flatMap { p =>
      val ranGen = new Random
      var arr1 = new Array[(Int, Array[Byte])]
(numKVPairs)
      for (i <- 0 until numKVPairs) {
        val byteArr = new Array[Byte](valSize)
        ranGen.nextBytes(byteArr)
        arr1(i) = (ranGen.nextInt(Int.MaxValue),
byteArr)
      }
      arr1
    }.cache
    // Enforce that everything has been calculated and
in cache
    pairs1.count
    println(pairs1.groupByKey(numReducers).count)
    sc.stop()
  }
}
```

Apache Spark

- Read-only Resilient Distributed Datasets (RDD)

RDD is a mechanism capable of holding the data to be repeatedly used in the memory. MapReduce of Hadoop had been held, fault tolerance, data locality, scalability has taken over as it is.



1. an RDD is a read-only, partitioned collection of records.
2. an RDD has enough information about how it was derived from other datasets (its lineage)
3. persistence and partitioning
4. lazy operations

Apache Spark

- Read-only Resilient Distributed Datasets (RDD)

  reference

  - Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing
    http://www-bcf.usc.edu/~minlanyu/teach/csci599-fall12/papers/nsdi_spark.pdf

  - My article of Qiita
    http://qiita.com/Hiroki11x/items/4f5129094da4c91955bc

Outline

1. Introduction

2. Approach

3. Implementation design

4. Related Works

5. Discussions and Conclusions

Logistic Regression and Linear Support Vector Machine

Most linear classification models consider the following optimization problem

$$\min_{\boldsymbol{w}} \quad f(\boldsymbol{w}) \equiv \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\xi(\boldsymbol{w};\boldsymbol{x}_i,y_i), \qquad (1)$$

$\{(x_i,y_i)\}_{i=1}^{l}$, $x_i \in R^n$, $y_i \in \{-1,1\}$, $\forall i$, :Given a set of training label-instance pairs

$C > 0$: User-specified parameter

$\xi(w;x_i,y_i)$: loss function

$$\xi(\boldsymbol{w};\boldsymbol{x}_i,y_i) \equiv \begin{cases} \max(0,1-y_i\boldsymbol{w}^T\boldsymbol{x}_i), & (2) \\ \max(0,1-y_i\boldsymbol{w}^T\boldsymbol{x}_i)^2, \quad \text{and} & (3) \\ \log\left(1+\exp\left(-y_i\boldsymbol{w}^T\boldsymbol{x}_i\right)\right). & (4) \end{cases}$$

Logistic Regression and Linear Support Vector Machine

Most linear classification models consider the following optimization problem

$$\min_{\boldsymbol{w}} \quad f(\boldsymbol{w}) \equiv \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\xi(\boldsymbol{w};\boldsymbol{x}_i,y_i)$$

regularizer — loss function

The objective function f has two parts: the regularizer that controls the complexity of the model, and the loss that measures the error of the model on the training data. The loss function $\xi(w;x_i,y_i)$ is typically a convex function in $\mathbf{w}$.

Logistic Regression and Linear Support Vector Machine

Problem (1) is referred as L1-loss and L2-loss SVM if (2) and (3) is used, respectively.

$$\min_{\boldsymbol{w}} \quad f(\boldsymbol{w}) \equiv \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\xi(\boldsymbol{w};\boldsymbol{x}_i,y_i), \qquad (1)$$

regularizer

loss function

$$\xi(\boldsymbol{w};\boldsymbol{x}_i,y_i) \equiv \begin{cases} \max(0,1-y_i\boldsymbol{w}^T\boldsymbol{x}_i), & (2) \\ \max(0,1-y_i\boldsymbol{w}^T\boldsymbol{x}_i)^2, \quad \text{and} & (3) \\ \log\left(1+\exp\left(-y_i\boldsymbol{w}^T\boldsymbol{x}_i\right)\right). & (4) \end{cases}$$

It is known that (2) and (3) are differentiable while (2) is not and is thus more difficult to optimize. Therefore, we focus on solving LR and L2-loss SVM in the rest of the paper.

Logistic Regression and Linear Support Vector Machine

Problem (1) is referred as LR if (4) is used.

$$\min_{\boldsymbol{w}} \quad f(\boldsymbol{w}) \equiv \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\xi(\boldsymbol{w};\boldsymbol{x}_i,y_i), \qquad (1)$$

regularizer

loss function

$$\xi(\boldsymbol{w};\boldsymbol{x}_i,y_i) \equiv \begin{cases} \max(0,1-y_i\boldsymbol{w}^T\boldsymbol{x}_i), & (2) \\ \max(0,1-y_i\boldsymbol{w}^T\boldsymbol{x}_i)^2, \quad \text{and} & (3) \\ \log\left(1+\exp\left(-y_i\boldsymbol{w}^T\boldsymbol{x}_i\right)\right). & (4) \end{cases}$$

Logistic Regression and Linear Support Vector Machine

Most linear classification models consider the following optimization problem

$$\min_{\boldsymbol{w}} \quad f(\boldsymbol{w}) \equiv \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\xi(\boldsymbol{w};\boldsymbol{x}_i,y_i), \qquad (1)$$

$\{(x_i,y_i)\}_{i=1}^{l}$, $x_i \in R^n$, $y_i \in \{-1,1\}$, $\forall i$, :Given a set of training label-instance pairs

$C > 0$: User-specified parameter

$\xi(w;x_i,y_i)$: loss function

$$\xi(\boldsymbol{w};\boldsymbol{x}_i,y_i) \equiv \begin{cases} \max(0,1-y_i\boldsymbol{w}^T\boldsymbol{x}_i), & (2) \\ \max(0,1-y_i\boldsymbol{w}^T\boldsymbol{x}_i)^2, \quad \text{and} & (3) \\ \log\left(1+\exp\left(-y_i\boldsymbol{w}^T\boldsymbol{x}_i\right)\right). & (4) \end{cases}$$

Use a trust region Newton method to minimize f(w) (Lin and Mor´e, 1999).

Trust Region Newton Method

- Trust region Newton method is a type of truncated Newton approach.

- We consider the trust region method (Lin and Mor´e, 1999), which is a truncated Newton method to deal with general bound-constrained optimization problems (i.e., variables are in certain intervals).

- At each iteration of a trust region Newton method for minimizing f(w), we have an iterate wt, a size Δt of the trust region, and a quadratic model.

Most linear classification models consider the following optimization problem

$$\min_{\boldsymbol{w}} \quad f(\boldsymbol{w}) \equiv \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{i=1}^{l}\xi(\boldsymbol{w};\boldsymbol{x}_i,y_i), \qquad (1)$$

To discuss Newton methods, we need the Hessian of f(w):

Truncated Newton method(skip)

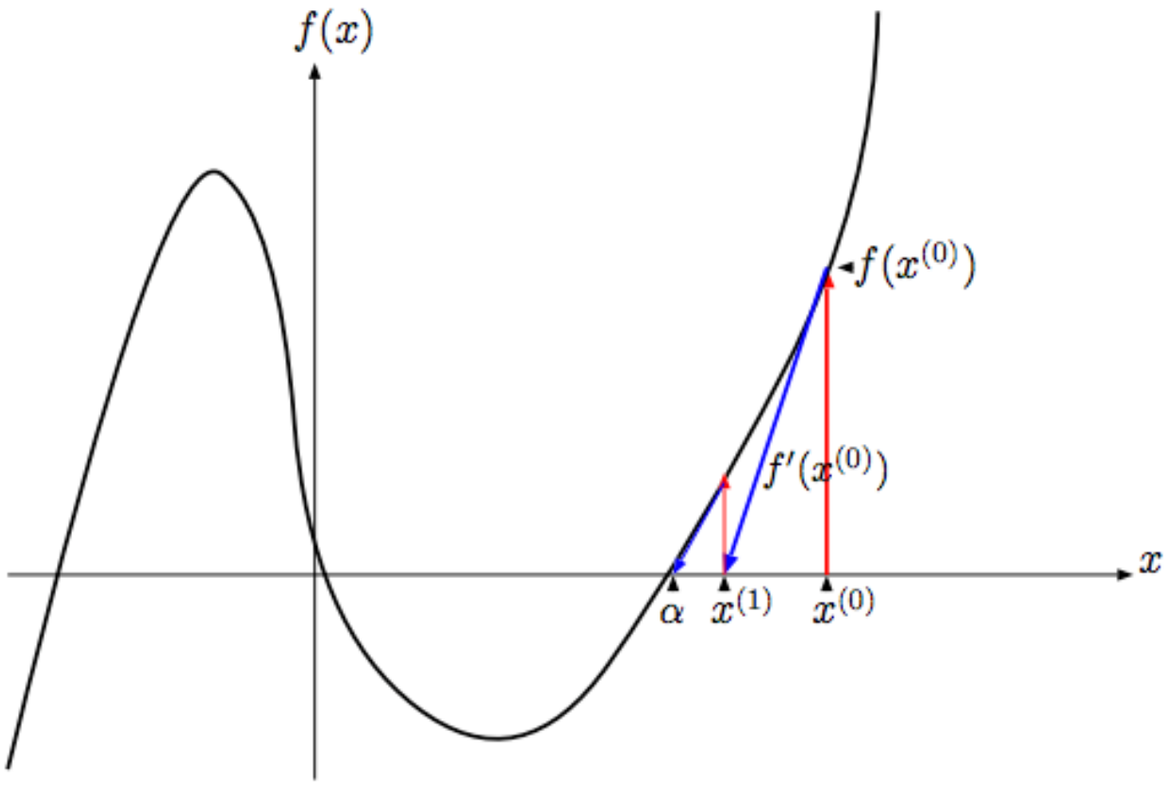
- Trust region Newton method is a type of truncated Newton approach.

- To save time, one may use only an“approximate” Newton direction in the early stages of the outer iterations.

- Such a technique is called truncated Newton method (or inexact Newton method).

Newton Method (skip)

To discuss Newton methods, we need the Hessian of f(w):



$$f(x^{(0)}) = f'(x^{(0)}) \times (x^{(0)} - x^{(1)})$$

$$x^{(1)} = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})}$$

$$x^{(\nu+1)} = x^{(\nu)} - \frac{f(x^{(\nu)})}{f'(x^{(\nu)})}$$

$$f_i(x_i^{(\nu)}) \sim \sum_{j=1}^{n} \frac{\partial f_i(x_i^{(\nu)})}{\partial x_j} \Delta x_j^{(\nu)}$$

- Newton's method assumes that the function can be locally approximated as a quadratic in the region around the optimum, and uses the first and second derivatives to find the stationary point.

- In higher dimensions, Newton's method uses the gradient and the Hessian matrix of second derivatives of the function to be minimized.

Trust Region Newton Method

At iteration t, given iterate w^t and trust region $\Delta_t > 0$, solve

$$\min_{\boldsymbol{d}} \quad q_t(\boldsymbol{d}), \quad \text{subject to} \quad \|\boldsymbol{d}\| \leq \Delta_t, \tag{5}$$

$$q_t(\boldsymbol{d}) \equiv \nabla f(\boldsymbol{w}^t)^T \boldsymbol{d} + \frac{1}{2} \boldsymbol{d}^T \nabla^2 f(\boldsymbol{w}^t) \boldsymbol{d} \tag{6}$$

(6) is the second-order Taylor approximation of $f(\boldsymbol{w}^t + \boldsymbol{d}) - f(\boldsymbol{w}^t)$

$$\rho_t = \frac{f(\mathbf{w}^t + \mathbf{d}) - f(\mathbf{w}^t)}{q_t(\mathbf{d})}.$$

$$\mathbf{w}^{t+1} = \begin{cases} \mathbf{w}^t + \mathbf{d} & \text{if } \rho_t > \eta, \\ \mathbf{w}^t & \text{if } \rho_t \leq \eta. \end{cases}$$

Adjust the trust region size by ρ_t.

If n is large: $\nabla^2 f(w) \in R^{n \times n}$ is too large to store.

Consider Hessian-free methods.

Trust Region Newton Method

At iteration t, given iterate w^t and trust region $\Delta_t > 0$, solve

$$\min_{\boldsymbol{d}} \quad q_t(\boldsymbol{d}), \quad \text{subject to} \quad \|\boldsymbol{d}\| \leq \Delta_t, \tag{5}$$

$$q_t(\boldsymbol{d}) \equiv \nabla f(\boldsymbol{w}^t)^T \boldsymbol{d} + \frac{1}{2} \boldsymbol{d}^T \nabla^2 f(\boldsymbol{w}^t) \boldsymbol{d} \tag{6}$$

(6) is the second-order Taylor approximation of $f(\boldsymbol{w}^t + \boldsymbol{d}) - f(\boldsymbol{w}^t)$

Because $\nabla^2 f(\boldsymbol{w})$ is too large to be formed and stored, a Hessian-free approach of applying CG (Conjugate Gradient) iterations is used to approximately solve (5) .

At each CG iteration we only need to obtain the Hessian-vector product $\nabla^2 f(w)v$ with some vector $\boldsymbol{v} \in \mathbf{R}^n$ generated by the CG procedure.

Trust Region Newton Method

Use a conjugate gradient (CG) method.

CG is an iterative method: only needs $\nabla^2 f(w)v$ for some $v \in \mathbf{R}^n$ at each iteration.(it's not necessary to calculate $\nabla^2 f(w^t)$)

Hessian-free methods.

For LR and SVM, at each CG iteration we compute

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + CX^T\left(D\left(X\boldsymbol{v}\right)\right). \qquad (7) \quad X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l]^T$$

is the data matrix and D is a diagonal matrix with values determined by w^t.

Conjugate gradient (CG) method (skip)

- an algorithm for the numerical solution of particular systems of linear equations, namely those whose matrix is symmetric and positive-definite.

Suppose we have some quadratic function

$$f(x) = \frac{1}{2}x^T Ax + b^T x + c$$

for $x \in \mathbb{R}^n$ with $A \in \mathbb{R}^{n \times n}$ and $b, c \in \mathbb{R}^n$.

We can write any quadratic function in this form, as this generates all the coefficients $x_i x_j$ as well as linear and constant terms. In addition, we can assume that $A = A^T$ (A is symmetric). (If it were not, we could just rewrite this with a symmetric A, since we could take the term for $x_i x_j$ and the term for $x_j x_i$, sum them, and then have $A_{ij} = A_{ji}$ both be half of this sum.)

Taking the gradient of f, we obtain

$$\nabla f(x) = Ax + b,$$

Distributed Hessian-vector Products

- (8) is bottleneck which is the product between the Hessian matrix $r^2 f(w^t)$ and the vector s^i
- This operation can possibly be parallelized in a distributed environment as parallel vector products.

Algorithm 1 CG procedure for approximately solving (5)

1: Given $\xi_t < 1, \Delta_t > 0$. Let $\bar{\boldsymbol{d}}^0 = 0, \boldsymbol{r}^0 = -\nabla f(\boldsymbol{w}^t)$, and $\boldsymbol{s}^0 = \boldsymbol{r}^0$.
2: For $i = 0, 1, \ldots$ (inner iterations)
3: If $\|\boldsymbol{r}^i\| \le \xi_t \|\nabla f(\boldsymbol{w}^t)\|$, output $\boldsymbol{d}^t = \bar{\boldsymbol{d}}^i$ and stop.
4: Compute

$$\boldsymbol{u}^i = \nabla^2 f(\boldsymbol{w}^t)\boldsymbol{s}^i. \tag{8}$$

5: $\alpha_i = \|\boldsymbol{r}^i\|^2 / ((\boldsymbol{s}^i)^T \boldsymbol{u}^i)$.
6: $\bar{\boldsymbol{d}}^{i+1} = \bar{\boldsymbol{d}}^i + \alpha_i \boldsymbol{s}^i$.
7: If $\|\bar{\boldsymbol{d}}^{i+1}\| \ge \Delta_t$, compute τ such that $\|\bar{\boldsymbol{d}}^i + \tau \boldsymbol{s}^i\| = \Delta_t$, then output the vector $\boldsymbol{d}^t = \bar{\boldsymbol{d}}^i + \tau \boldsymbol{s}^i$ and stop.
8: $\boldsymbol{r}^{i+1} = \boldsymbol{r}^i - \alpha_i \boldsymbol{u}^i$.
9: $\beta_i = \|\boldsymbol{r}^{i+1}\|^2 / \|\boldsymbol{r}^i\|^2$.
10: $\boldsymbol{s}^{i+1} = \boldsymbol{r}^{i+1} + \beta_i \boldsymbol{s}^i$.

Distributed Hessian-vector Products

- first partition the data matrix X and the labels Y into disjoint p parts.

$$X = [X_1, \ldots, X_p]^T,$$

$$Y = \operatorname{diag}(y_1, \ldots, y_l) = \begin{bmatrix} Y_1 & & \\ & \ddots & \\ & & Y_p \end{bmatrix}$$

- reformulate the function, the gradient and the Hessian- vector products of (1) as follows.

$$f(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{k=1}^{p} f_k(\boldsymbol{w}), \tag{9}$$

$$\nabla f(\boldsymbol{w}) = \boldsymbol{w} + C\sum_{k=1}^{p} \nabla f_k(\boldsymbol{w}), \tag{10}$$

where

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + C\sum_{k=1}^{p} \nabla^2 f_k(\boldsymbol{w})\boldsymbol{v}, \tag{11}$$

$$f_k(\boldsymbol{w}) \equiv \boldsymbol{e}_k^T \log\left(\sigma\left(Y_k X_k \boldsymbol{w}\right)\right), \tag{12}$$

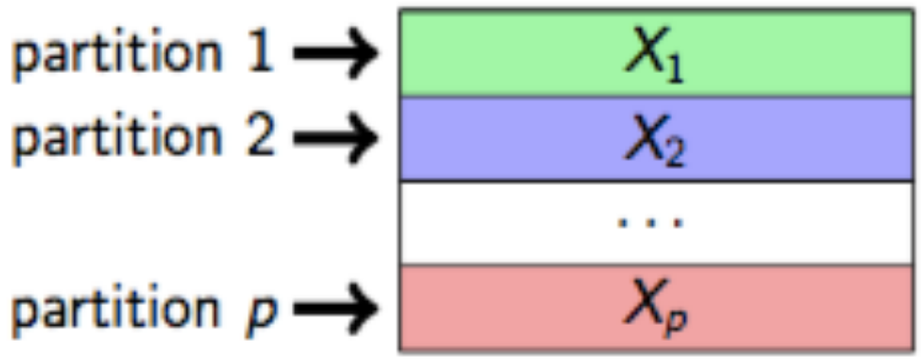
$$\nabla f_k(\boldsymbol{w}) \equiv \left(Y_k X_k\right)^T \left(\sigma\left(Y_k X_k \boldsymbol{w}\right)^{-1} - \boldsymbol{e}_k\right), \tag{13}$$

$$\nabla^2 f_k(\boldsymbol{w})\boldsymbol{v} \equiv X_k^T\left(D_k\left(X_k\boldsymbol{v}\right)\right), \tag{14}$$

$$D_k \equiv \operatorname{diag}\left(\left(\sigma\left(Y_k X_k \boldsymbol{w}\right) - \boldsymbol{e}_k\right)/\sigma\left(Y_k X_k \boldsymbol{w}\right)^2\right),$$

Distributed Hessian-vector Products

Data matrix X is distributedly stored



The functions $f_k, \nabla f_k$ and $\nabla^2 f_k$ are the map functions operating on the k-th partition. We can observe that for computing (12)-(14), only the data partition X_k is needed in computing.

$$f_k(\boldsymbol{w}) \equiv \boldsymbol{e}_k^T \log\left(\sigma\left(Y_k X_k \boldsymbol{w}\right)\right), \tag{12}$$

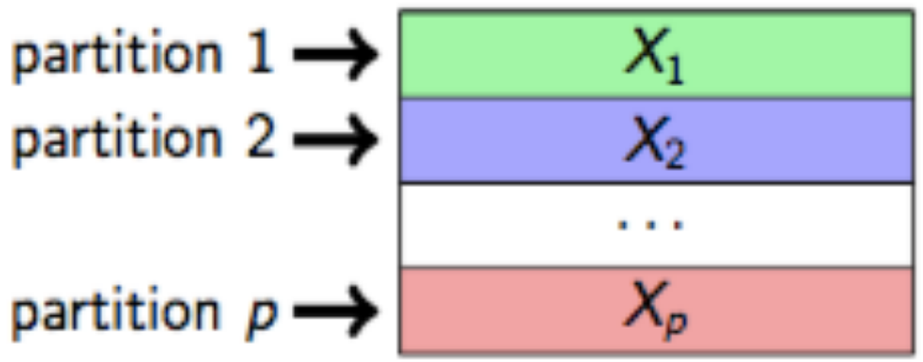
$$\nabla f_k(\boldsymbol{w}) \equiv \left(Y_k X_k\right)^T \left(\sigma\left(Y_k X_k \boldsymbol{w}\right)^{-1} - \boldsymbol{e}_k\right), \tag{13}$$

$$\nabla^2 f_k(\boldsymbol{w})\boldsymbol{v} \equiv X_k^T \left(D_k \left(X_k \boldsymbol{v}\right)\right), \tag{14}$$

$$D_k \equiv \operatorname{diag}\left(\left(\sigma\left(Y_k X_k \boldsymbol{w}\right) - \boldsymbol{e}_k\right) / \sigma\left(Y_k X_k \boldsymbol{w}\right)^2\right),$$

Distributed Hessian-vector Products

Data matrix X is distributedly stored



$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + CX^T\left(D\left(X\boldsymbol{v}\right)\right). \qquad (7)$$

$$X^T DX\mathbf{v} = X_1^T D_1 X_1\mathbf{v} + \cdots + X_p^T D_p X_p\mathbf{v}$$

- p ≥ (#slave nodes) for parallelization.
- Two communications per operation:
  1. Master sends w^t and the current v to the slaves.
  2. Slaves return $X^T D_i X_i v$ to master.
- The same scheme for computing function/gradient.

Outline

1. Introduction

2. Approach

3. Implementation design

4. Related Works

5. Discussions and Conclusions

Implementation design

- In this section, they study implementation issues for their software.

Implementation design

- In this section, they study implementation issues for their software.

- They name their distributed TRON implementation Spark LIBLINEAR because algorithmically it is an extension of the TRON implementation in the software LIBLINEAR[10]

Implementation design

- In this section, they study implementation issues for their software.

- They name their distributed TRON implementation Spark LIBLINEAR because algorithmically it is an extension of the TRON implementation in the software LIBLINEAR[10]

- Spark is implemented in Scala, we use the same language. So, The implementation of Spark LIBLINEAR involves complicated design issues resulting from Java, Scala and Spark.

Implementation design

- Spark is implemented in Scala, we use the same language. So, The implementation of Spark LIBLINEAR involves complicated design issues resulting from Java, Scala and Spark.

Implementation design

- Spark is implemented in Scala, we use the same language. So, The implementation of Spark LIBLINEAR involves complicated design issues resulting from Java, Scala and Spark.

For example, in contrast to traditional languages like C and C++, similar expressions in Scala may easily behave differently.

Implementation design

- Spark is implemented in Scala, we use the same language. So, The implementation of Spark LIBLINEAR involves complicated design issues resulting from Java, Scala and Spark.

For example, in contrast to traditional languages like C and C++, similar expressions in Scala may easily behave differently.

It is hence easy to introduce overheads in developing Scala programs

Implementation design

- They analyze the following different implementation issues for efficient computation, communication and memory usage.

  - Programming language:
    - Loop structure
    - Data encapsulation
  - Operations on RDD:
    - Using mapPartitions rather than map
    - Caching intermediate information or not
  - Communication:
    - Using broadcast variables
    - The cost of the reduce function

Implementation design

- They analyze the following different implementation issues for efficient computation, communication and memory usage.

  - Programming language:
    - Loop structure
    - Data encapsulation

  related to Java and Scala

  - Operations on RDD:
    - Using mapPartitions rather than map
    - Caching intermediate information or not
  - Communication:
    - Using broadcast variables
    - The cost of the reduce function

  related to Spark

Scala Issue: Loop structures

- From (12)-(14), clearly the computational bottleneck at each node is on the products between the data matrix X_k (or X_k^T) and a vector v.
  To compute this matrix-vector product, a loop to conduct inner products between all $x_i \in X_k$ and v is executed many times, it is the main computation in this algorithm.

$$f_k(\boldsymbol{w}) \equiv \boldsymbol{e}_k^T \log\left(\sigma\left(Y_k X_k \boldsymbol{w}\right)\right), \tag{12}$$

$$\nabla f_k(\boldsymbol{w}) \equiv \left(Y_k X_k\right)^T \left(\sigma\left(Y_k X_k \boldsymbol{w}\right)^{-1} - \boldsymbol{e}_k\right), \tag{13}$$

$$\nabla^2 f_k(\boldsymbol{w})\boldsymbol{v} \equiv X_k^T \left(D_k \left(X_k \boldsymbol{v}\right)\right), \tag{14}$$

$$D_k \equiv \operatorname{diag}\left(\left(\sigma\left(Y_k X_k \boldsymbol{w}\right) - \boldsymbol{e}_k\right) / \sigma\left(Y_k X_k \boldsymbol{w}\right)^2\right),$$

Scala Issue: Loop structures

- From (12)-(14), clearly the computational bottleneck at each node is on the products between the data matrix X_k (or X_k^T) and a vector v.
  To compute this matrix-vector product, a loop to conduct inner products between all $x_i \in X_k$ and v is executed many times, it is the main computation in this algorithm.

- Although a for loop is the most straightforward way to implement an inner product, unfortunately, it is known that in Scala, a for loop may be slower than a while loop.[2]

Scala Issue: Loop structures

- To study this issue, we discuss three methods to implement the inner product: for-generator, for-range and while.

for-generator

```
for(element ← collection) { ... }
```

for-range

```
for(i ← 0 to collection.length) { ... }
```

while

```
i = 0;
while(condition) {
        ...
        i = i + 1;
}
```

Experiment Data Information

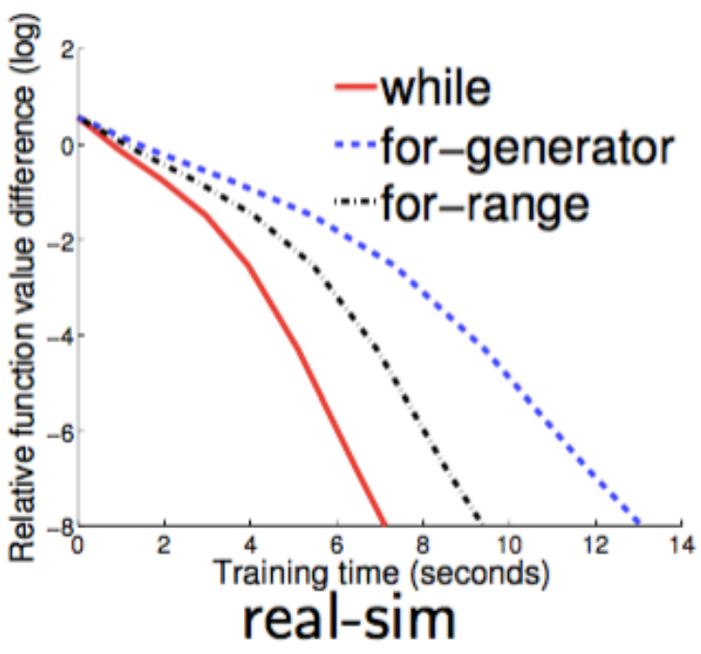
TABLE I. DATA INFORMATION: DENSITY IS THE AVERAGE RATIO OF NON-ZERO FEATURES PER INSTANCE.

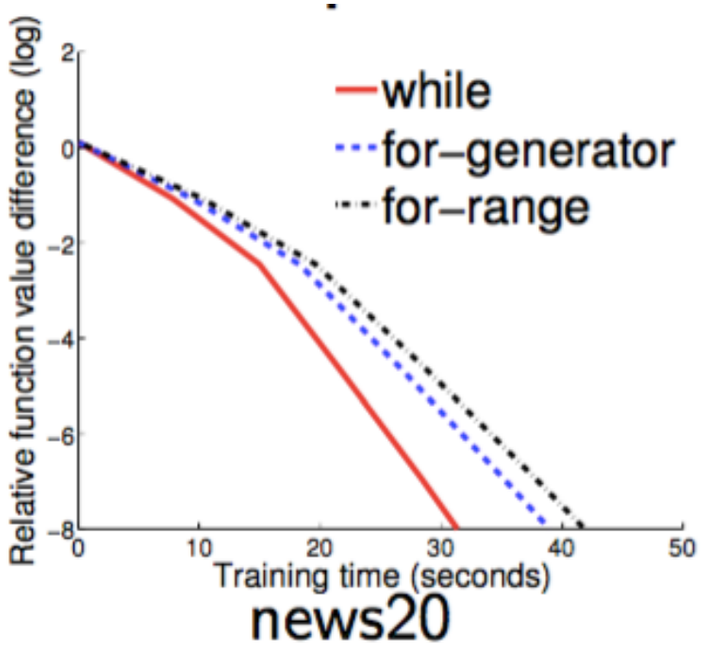
Data set	#instances	#features	density	#nonzeros	p_s	p_e
ijcnn	49,990	22	59.09%	649,870	1	
real-sim	72,309	20,958	0.25%	3,709,083	2	
rcv1	20,242	47,236	0.16%	1,498,952	1	
news20	19,996	1,355,191	0.03%	9,097,916	2	
covtype	581,012	54	22.00%	6,901,775		32
webspam	350,000	254	33.52%	29,796,333	6	32
epsilon	400,000	2,000	100.00%	800,000,000		183
rcv1t	677,399	47,236	0.16%	49,556,258		32
yahoo-japan	176,203	832,026	0.02%	23,506,415	5	32
yahoo-korea	460,554	3,052,939	0.01%	156,436,656		34

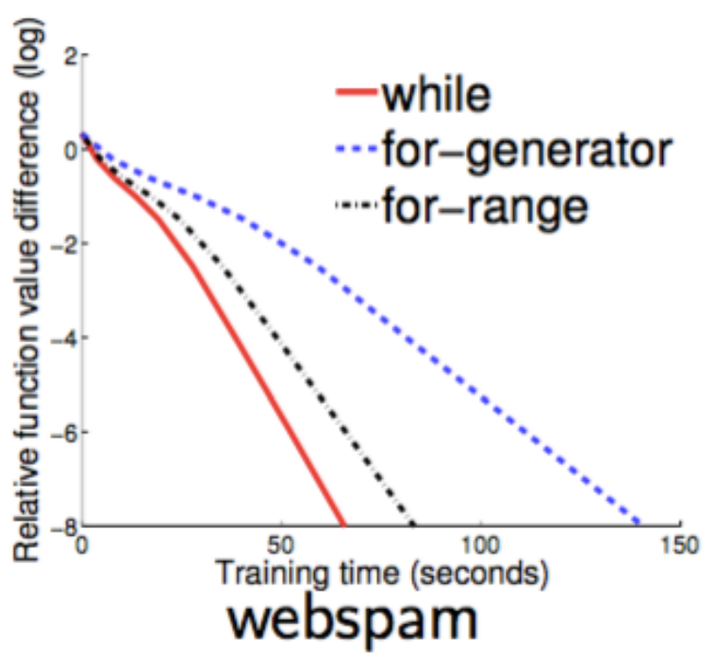
density: avg. ratio of non-zero features per instance.
p_s: used for the experiments of loops and encapsulation
p_e: applied for the rest. In the experiments of loops and encapsulation

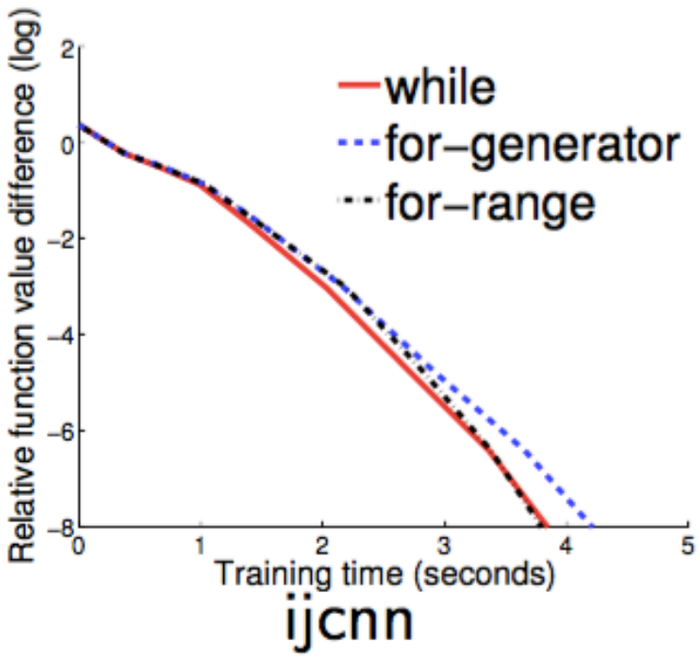
Scala Issue: Loop structures

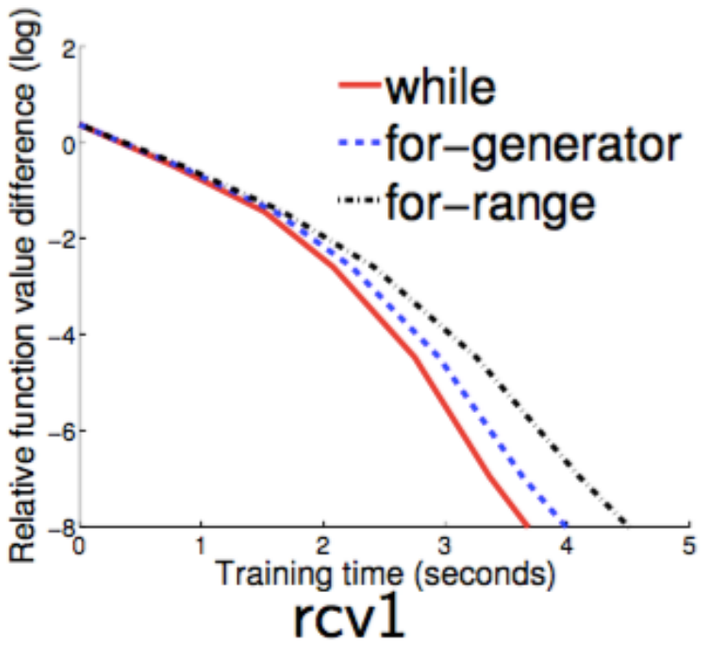
- Use one node in this experiment.

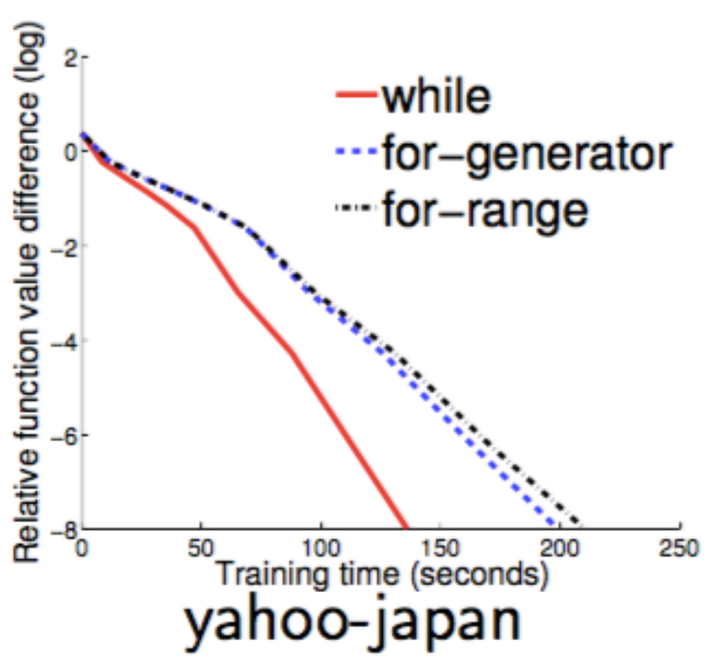






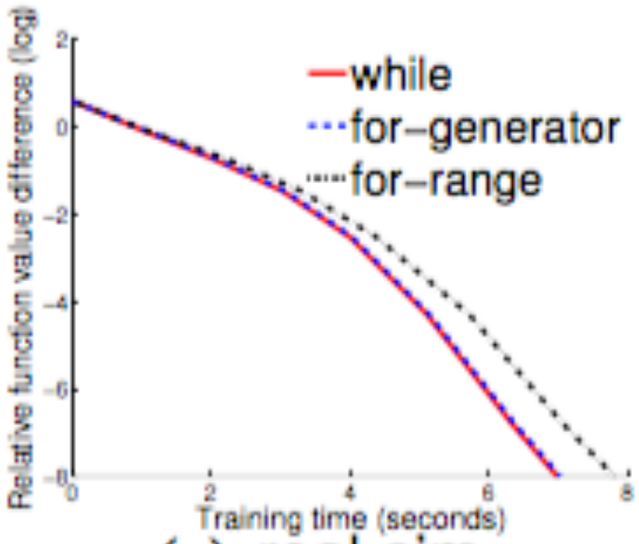




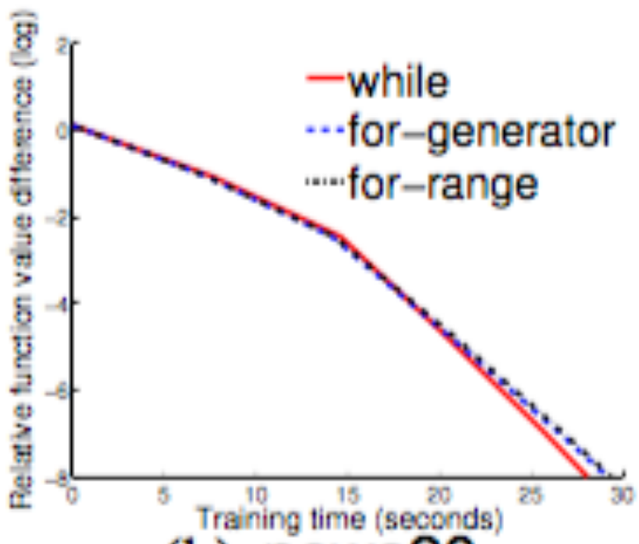


Scala Issue: Loop structures

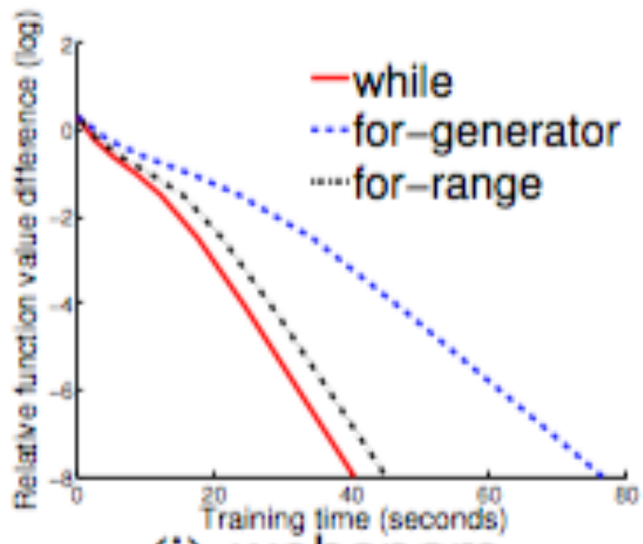
- Use two nodes in this experiment.



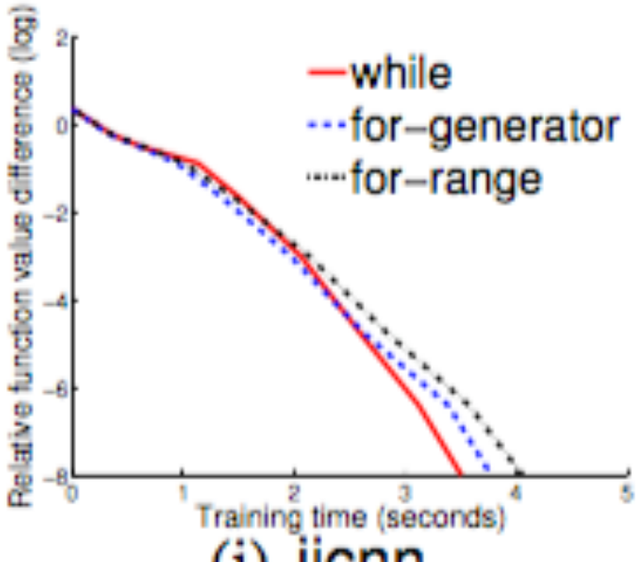
(g) real-sim



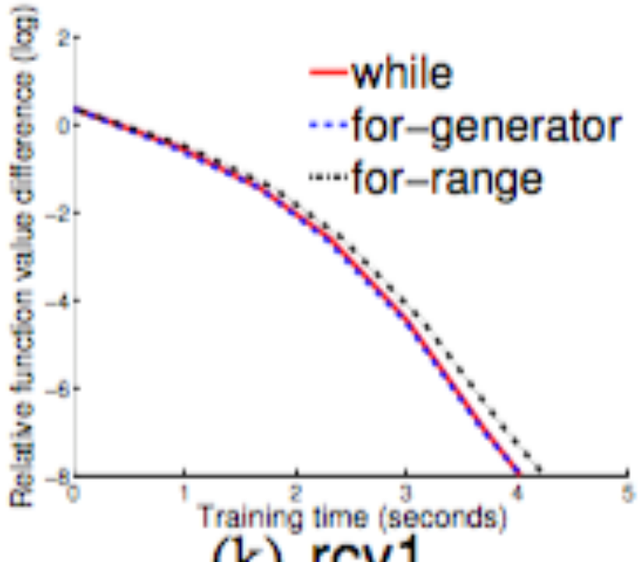
(h) news20



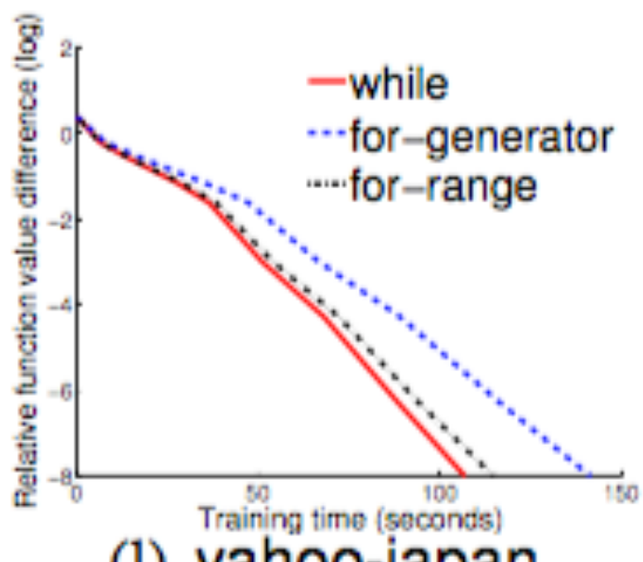
(i) webspam



(j) ijcnn



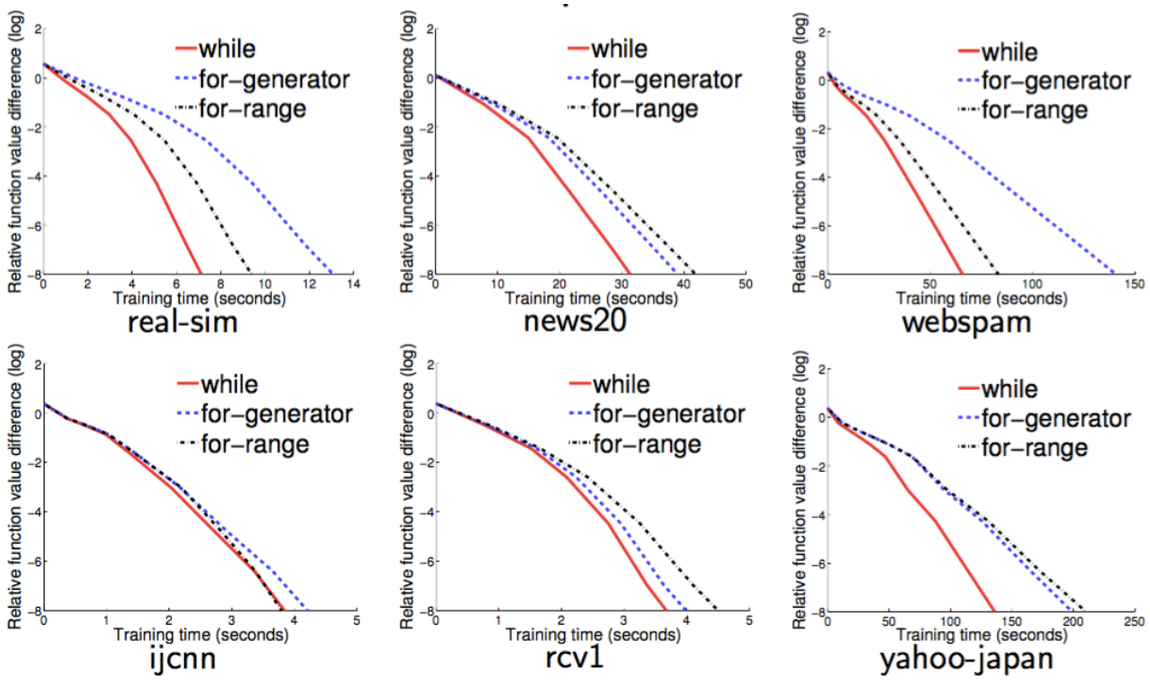
(k) rcv1



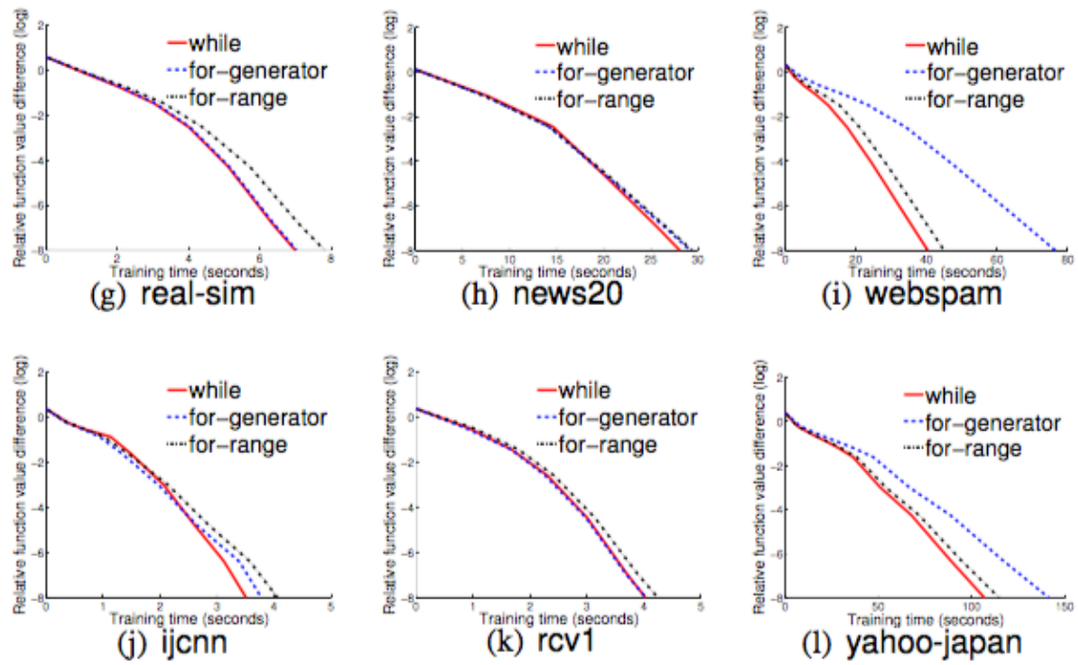
(l) yahoo-japan

Scala Issue: Loop structures

one node



two nodes



The reason is that when data is split between two nodes, each node requires conducting fewer loops.

Scala Issue: Loop structures

- The translation comes with overheads and the combination becomes complicated when more operations are applied.

- The optimization of a for expression has not been a focus in Scala development because this expression is too imperative to consist with the functional programming principle.

- In contrast, a while loop is a loop rather than an expression.

Scala Issue: Loop structures

- The translation comes with overheads and the combination becomes complicated when more operations are applied.

- The optimization of a for expression has not been a focus in Scala development because this expression is too imperative to consist with the functional programming principle.

- In contrast, a while loop is a loop rather than an expression.

- The while loop is chosen to implement their software.

Scala Issue: Encapsulation

- follow LIBLINEAR to represent data as a sparse matrix, where only non-zero entries are stored.
  This strategy is important to handle large-scale data.

Scala Issue: Encapsulation

- follow LIBLINEAR to represent data as a sparse matrix, where only non-zero entries are stored.
  This strategy is important to handle large-scale data.

For example, for a given 5-dimensional feature vector (2, 0, 0, 8, 0), only two index- value pairs of non-zero entries are stored.

Scala Issue: Encapsulation

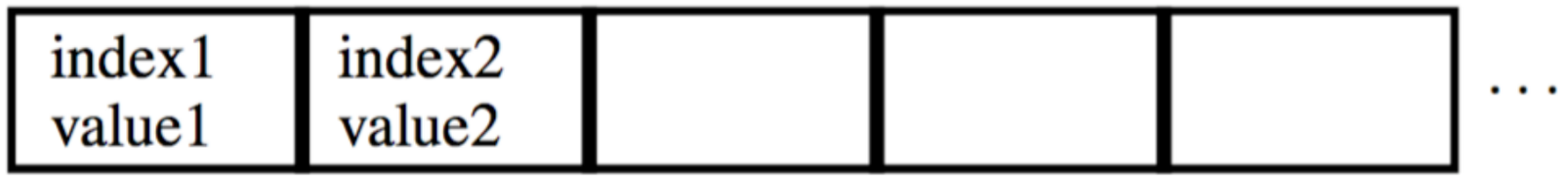
- follow LIBLINEAR to represent data as a sparse matrix, where only non-zero entries are stored.
  This strategy is important to handle large-scale data.

  For example, for a given 5-dimensional feature vector (2, 0, 0, 8, 0), only two index- value pairs of non-zero entries are stored.

- Investigate how to store the index-value information such as "1:2" and "4:8" in memory. The discussion is based on two encapsulation implementations: the Class approach (CA) and the Array approach (AA).

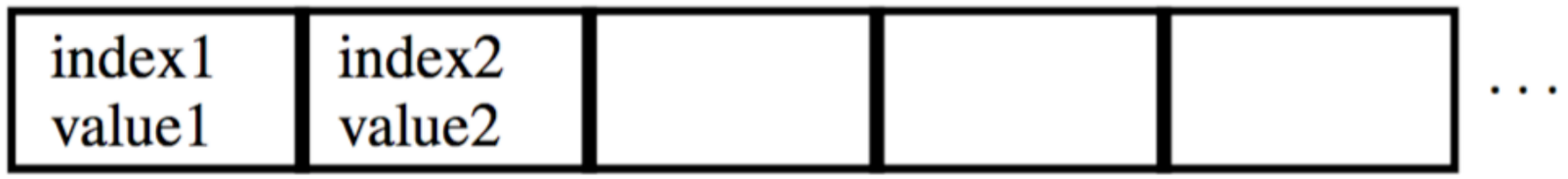
Scala Issue: Encapsulation

- CA encapsulates a pair of index and feature value into a class, and maintains an array of class objects for each instance.

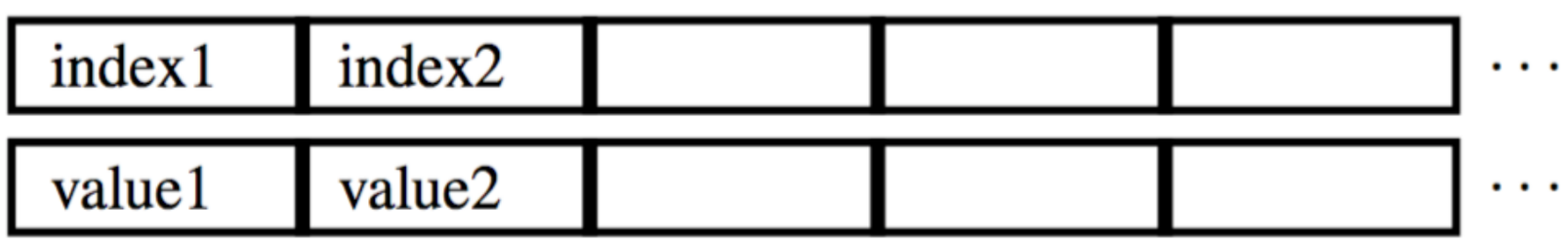


Scala Issue: Encapsulation

- CA encapsulates a pair of index and feature value into a class, and maintains an array of class objects for each instance.

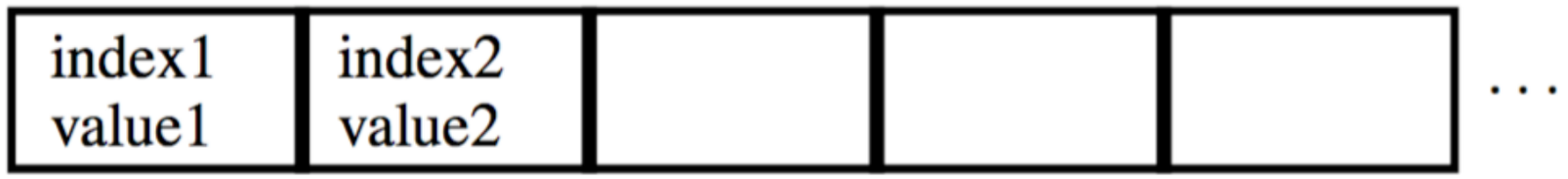


- In contrast, AA directly uses two arrays to store indices and feature values of an instance.

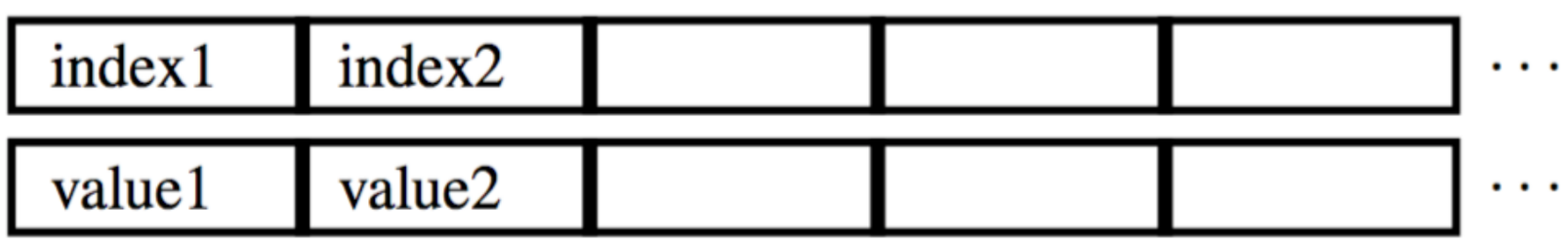


Scala Issue: Encapsulation

- CA encapsulates a pair of index and feature value into a class, and maintains an array of class objects for each instance.



- In contrast, AA directly uses two arrays to store indices and feature values of an instance.



- AA is faster because it directly accesses indices and values, while the CPU cache must access the pointers of class objects first if CA is applied

RDD: Using mapPartitions rather than map

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + CX^T\left(D\left(X\boldsymbol{v}\right)\right). \qquad (7)$$

- The second term of the Hessian-vector product (7) can be represented as the following form.

$$\sum_{i=1}^{l} \boldsymbol{x}_i D_{i,i} \boldsymbol{x}_i^T \boldsymbol{v} = \sum_{i=1}^{l} a(\boldsymbol{x}_i, y_i, \boldsymbol{w}, \boldsymbol{v})\boldsymbol{x}_i,$$

where $a(y_i, x_i, w, v) = D_{i,i} x^T_i v$, can be computed by either map or mapPartitions.

RDD: Using mapPartitions rather than map

- The second term of the Hessian-vector product (7) can be represented as the following form.

$$\sum_{i=1}^{l} \boldsymbol{x}_i D_{i,i} \boldsymbol{x}_i^T \boldsymbol{v} = \sum_{i=1}^{l} a(\boldsymbol{x}_i, y_i, \boldsymbol{w}, \boldsymbol{v}) \boldsymbol{x}_i,$$

where $a(y_i, x_i, w, v) = D_{i,i} x^T_i v$, can be computed by either map or mapPartitions.

Algorithm 3 map implementation

```
1: data.map(new Function() {
2:     call(x, y) { return a(x, y, w, v)x }
3: }).reduce(new Function() {
4:     call(a, b) { return a + b }
5: })
```

RDD: Using mapPartitions rather than map

- Then map and reduce operations can be directly applied.
- Considerable overheads occur in the map operations because for each instance x_i, an intermediate vector $a(x_i, y_i, w, v)x_i$ is created.

Algorithm 3 map implementation

```
1: data.map(new Function() {
2:     call(x, y) { return a(x, y, w, v)x }
3: }).reduce(new Function() {
4:     call(a, b) { return a + b }
5: })
```

RDD: Using mapPartitions rather than map

- Then map and reduce operations can be directly applied.
- Considerable overheads occur in the map operations because for each instance x_i, an intermediate vector $a(x_i, y_i, w, v)x_i$ is created.
- In addition to the above-mentioned overheads, the reduce function in Algorithm 3 involves complicated computation.

Algorithm 3 map implementation

```
1: data.map(new Function() {
2:     call(x, y) { return a(x, y, w, v)x }
3: }).reduce(new Function() {
4:     call(a, b) { return a + b }
5: })
```

RDD: Using mapPartitions rather than map

$$\sum_{i=1}^{l} \boldsymbol{x}_i D_{i,i} \boldsymbol{x}_i^T \boldsymbol{v} = \sum_{i=1}^{l} a(\boldsymbol{x}_i, y_i, \boldsymbol{w}, \boldsymbol{v}) \boldsymbol{x}_i,$$

where $a(y_i, x_i, w, v) = D_{i,i} x^T_i v$, can be computed by either map or mapPartitions.

Algorithm 4 mapPartitions implementation

```
1: data.mapPartitions(new Function() {
2:     call(partition) {
3:         partitionHv = new DenseVector(n)
4:         for each (x, y) in partition
5:             partitionHv += a(x, y, w, v)x
6:     }
7: }).reduce(new Function() {
8:     call(a, b) { return a + b }
9: })
```

RDD: Using mapPartitions rather than map

- To avoid the overheads and the complicated additions of sparse vectors, they consider the mapPartitions operation in Spark.

Algorithm 4 mapPartitions implementation

```
1: data.mapPartitions(new Function() {
2:     call(partition) {
3:         partitionHv = new DenseVector(n)
4:         for each (x, y) in partition
5:             partitionHv += a(x, y, w, v)x
6:     }
7: }).reduce(new Function() {
8:     call(a, b) { return a + b }
9: })
```

RDD: Using mapPartitions rather than map

- This setting ensures that computing a Hessian-vector product involves only p intermediate vectors.
- The overheads of using map- Partitions is less than that of using map with l intermediate vectors.

Algorithm 4 mapPartitions implementation

```
1: data.mapPartitions(new Function() {
2:     call(partition) {
3:         partitionHv = new DenseVector(n)
4:         for each (x, y) in partition
5:             partitionHv += a(x, y, w, v)x
6:     }
7: }).reduce(new Function() {
8:     call(a, b) { return a + b }
9: })
```

RDD: Using mapPartitions rather than map

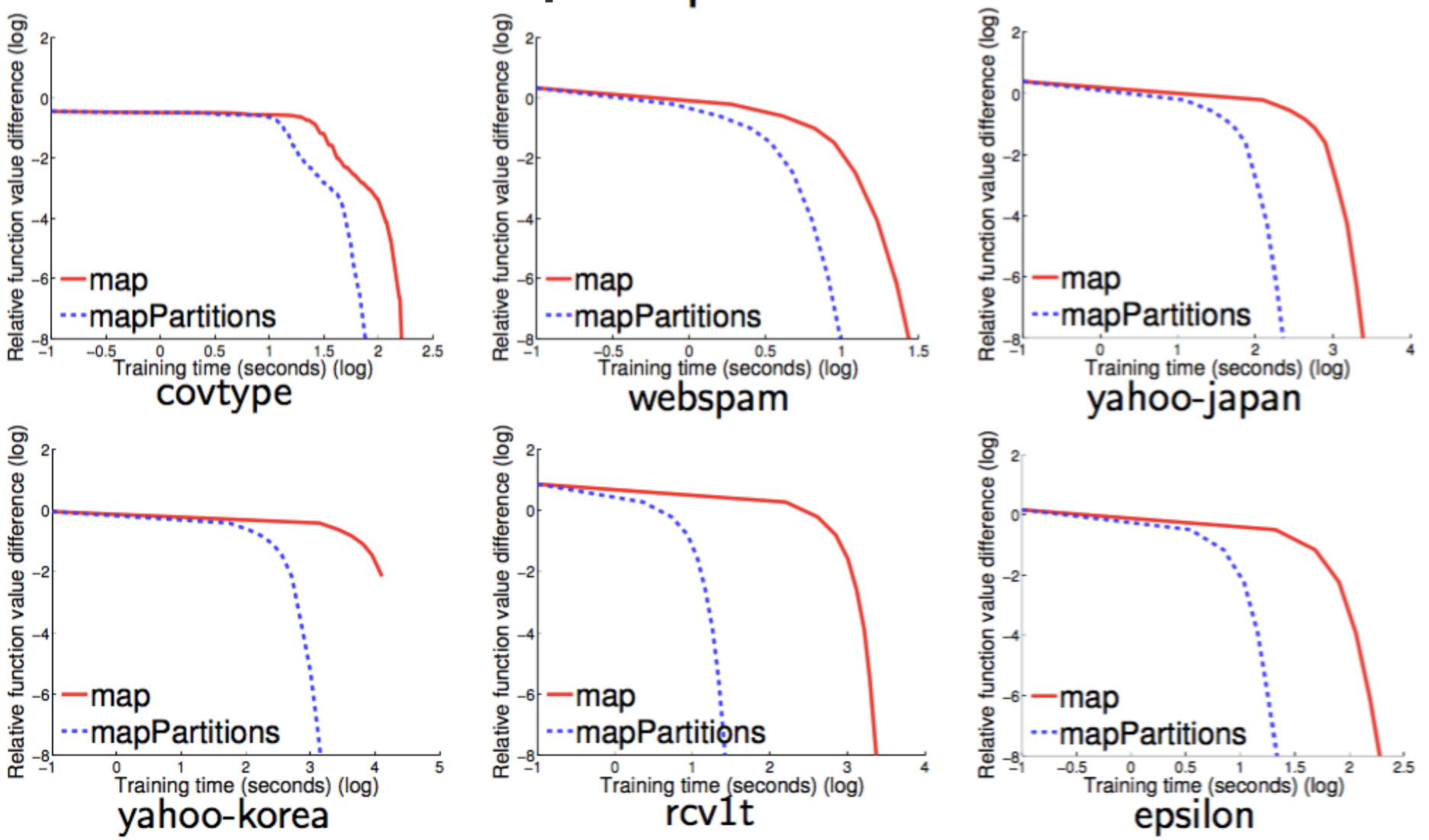
- Note that the technique of using mapPartitions can also be applied to compute the gradient.

Algorithm 4 mapPartitions implementation

```
1: data.mapPartitions(new Function() {
2:     call(partition) {
3:         partitionHv = new DenseVector(n)
4:         for each (x, y) in partition
5:             partitionHv += a(x, y, w, v)x
6:     }
7: }).reduce(new Function() {
8:     call(a, b) { return a + b }
9: })
```

RDD: Using mapPartitions rather than map

- Use 16 nodes in this experiment.



- the longer running time of map implies its higher computation cost

RDD: Caching intermediate information or not 83

- The calculations of (12)-(14) all involve the vector $(Y_k X_k w)$.

- Instinctively, if this vector is cached and shared between different operations, the training procedure can be more efficient.

- In fact, the single-machine package LIBLINEAR uses this strategy in their implementation of TRON.

$$f_k(\boldsymbol{w}) \equiv \boldsymbol{e}_k^T \log\left(\sigma\left(Y_k X_k \boldsymbol{w}\right)\right), \tag{12}$$

$$\nabla f_k(\boldsymbol{w}) \equiv \left(Y_k X_k\right)^T \left(\sigma\left(Y_k X_k \boldsymbol{w}\right)^{-1} - \boldsymbol{e}_k\right), \tag{13}$$

$$\nabla^2 f_k(\boldsymbol{w})\boldsymbol{v} \equiv X_k^T \left(D_k \left(X_k \boldsymbol{v}\right)\right), \tag{14}$$

$$D_k \equiv \operatorname{diag}\left(\left(\sigma\left(Y_k X_k \boldsymbol{w}\right) - \boldsymbol{e}_k\right) / \sigma\left(Y_k X_k \boldsymbol{w}\right)^2\right),$$

RDD: Caching intermediate information or not 84

- Spark does not allow any single operation to gather information from two different RDDs and run a user-specified function such as (12), (13) or (14).

- It is necessary to create one new RDD per iteration to store both the training data and the information to be cached.

- Unfortunately, this approach incurs severe overheads in copying the training data from the original RDD to the new one.

RDD: Caching intermediate information or not 85

- Based on the above discussion, they decide not to cache (Y_k X_kw) because recomputing them is more cost-effective.

- This example demonstrates that specific properties of a parallel programming framework may strongly affect the implementation.

RDD: Communication

- In Algorithm 2, communication occurs at two places. The first one is sending w and v from the master machine to the slave machines, and the second one is reducing the results of (12)-(14) to the master machine.

Algorithm 2 A distributed TRON algorithm for LR and SVM

1: Given $\boldsymbol{w}^0, \Delta_0, \eta, \epsilon$.
2: For $t = 0, 1, \ldots$
3: The master ships $\boldsymbol{w}^t$ to every slave.
4: Slaves compute $f_k(\boldsymbol{w}^t)$ and $\nabla f_k(\boldsymbol{w}^t)$ and reduce them to the master.
5: If $\|\nabla f(\boldsymbol{w}^t)\| < \epsilon$, stop.
6: Find $\boldsymbol{d}^t$ by solving (5) using Algorithm 1.
7: Compute $\rho_t = \frac{f(\boldsymbol{w}^t + \boldsymbol{d}^t) - f(\boldsymbol{w}^t)}{q_t(\boldsymbol{d}^t)}$.
8: Update $\boldsymbol{w}^t$ to $\boldsymbol{w}^{t+1}$ according to

$$\boldsymbol{w}^{t+1} = \begin{cases} \boldsymbol{w}^t + \boldsymbol{d}^t & \text{if } \rho_t > \eta, \\ \boldsymbol{w}^t & \text{if } \rho_t \leq \eta. \end{cases}$$

9: Obtain Δ_{t+1} by rules in [13].

$$f_k(\boldsymbol{w}) \equiv \boldsymbol{e}_k^T \log\left(\sigma\left(Y_k X_k \boldsymbol{w}\right)\right), \tag{12}$$

$$\nabla f_k(\boldsymbol{w}) \equiv \left(Y_k X_k\right)^T \left(\sigma\left(Y_k X_k \boldsymbol{w}\right)^{-1} - \boldsymbol{e}_k\right), \tag{13}$$

$$\nabla^2 f_k(\boldsymbol{w})\boldsymbol{v} \equiv X_k^T \left(D_k \left(X_k \boldsymbol{v}\right)\right), \tag{14}$$

$$D_k \equiv \operatorname{diag}\left(\left(\sigma\left(Y_k X_k \boldsymbol{w}\right) - \boldsymbol{e}_k\right) / \sigma\left(Y_k X_k \boldsymbol{w}\right)^2\right),$$

RDD: Using Broadcast Variables

- In Spark, when an RDD is split into partitions, one single operation on this RDD is divided into tasks working on different partitions.

- Under this setting, many redundant communications occur because just need to send a copy to each slave machine but not each partition.

- In such a case where each partition shares the same information from the master, it is recommended to use broadcast variables

RDD: Using Broadcast Variables

- Use broadcast variables to improve.

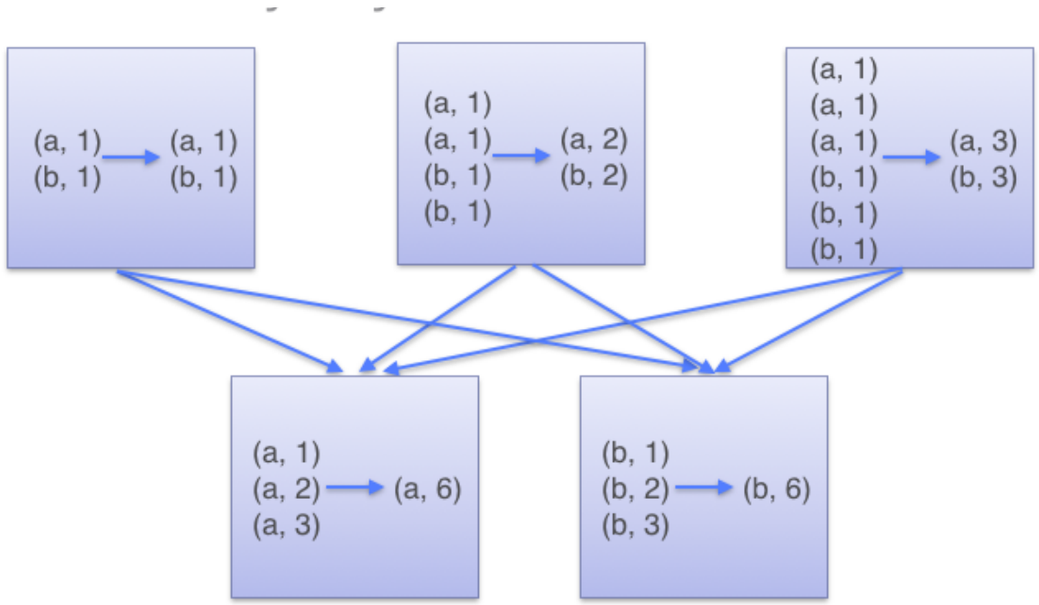
Read-only variables shared among partitions in the same node.

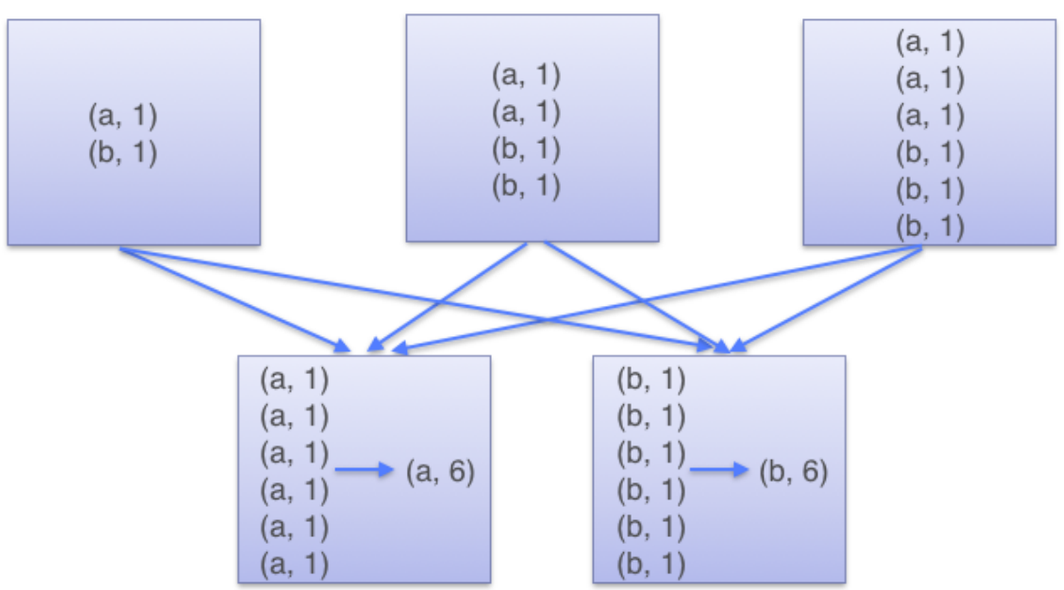
Cached in the slave machines.

RDD: The Cost of the reduce Function

- Slaves to master: Spark by default collect results from each partition separately.

- Use the coalesce function: Merge partitions on the same node before communication.





RDD: The Cost of the reduce Function

Use 16 nodes in this experiment.

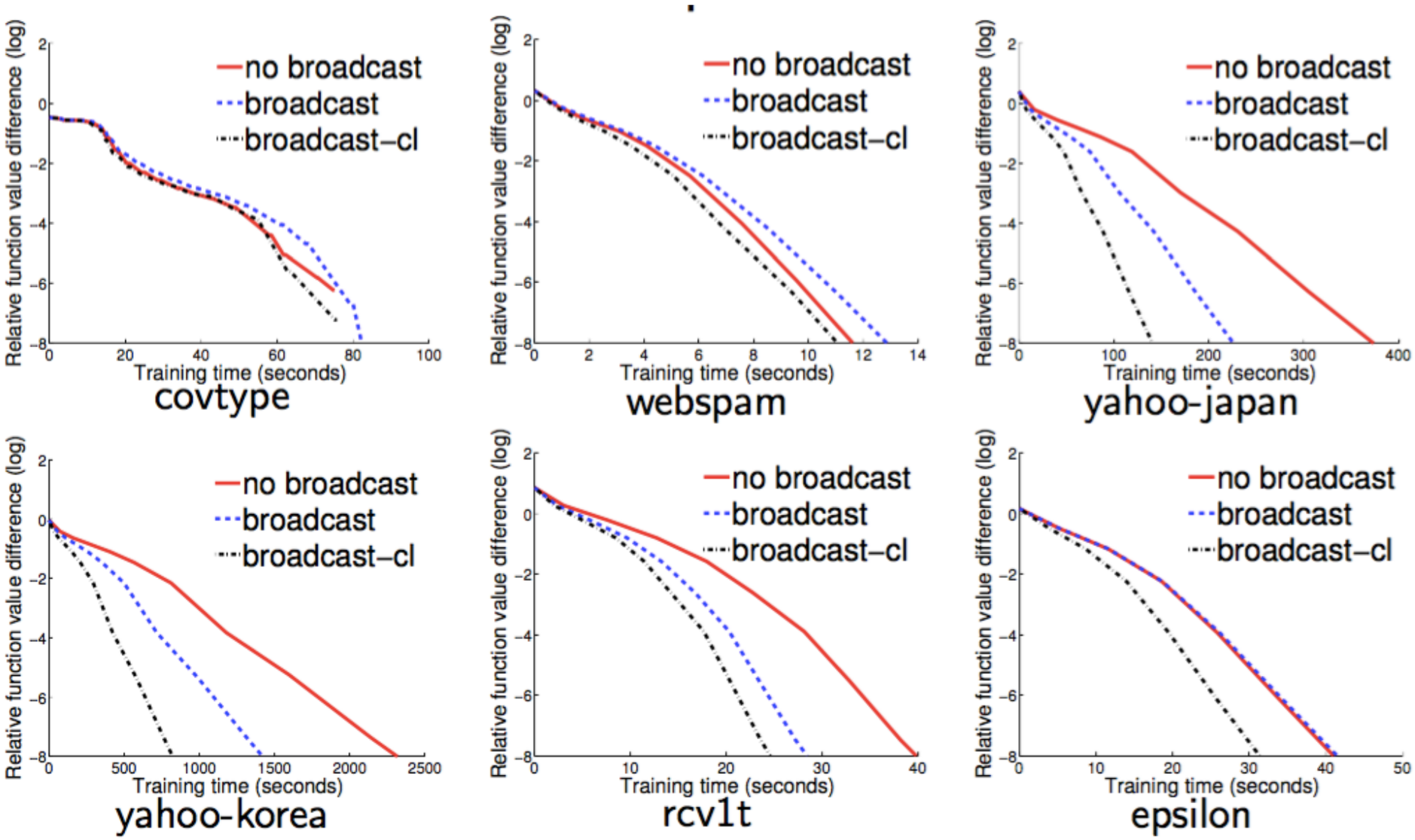


Fig. 4. Broadcast variables and coalesce: We present running time (in seconds) versus the relative objective value difference. We run LR with C = 1 on 16 nodes. Note that broadcast-cl represents the implementation with both broadcast variables and the coalesce function.

RDD: The Cost of the reduce Function

- for data with few features like covtype and webspam, adopting broadcast variables slightly degrades the efficiency because the communication cost is low and broadcast variables introduce some overheads.

- Regarding the coalesce function, it is bene- ficial for all data sets.

Outline

1. Introduction

2. Approach

3. Implementation design

4. Related Works

5. Discussions and Conclusions

Related Works

- MLlib is a machine learning library implemented in Apache Spark.
- A stochastic gradient method for LR and SVM (but default batch size is the whole data).

Related Works

Use 16 nodes in this experiment.

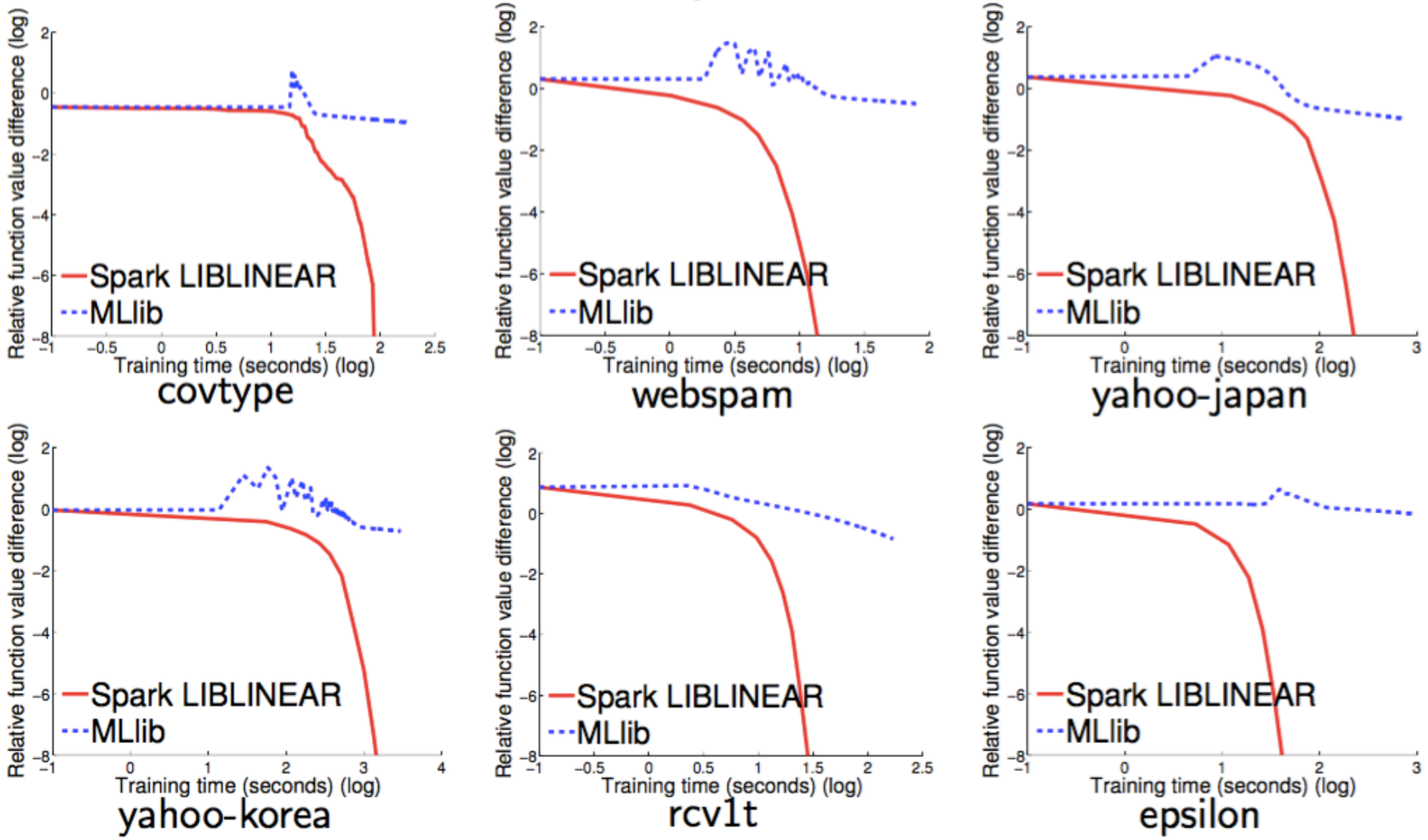


Fig. 6. Comparison with MLlib: We present running time (in seconds, log scale) versus the relative objective value difference. We run LR with C = 1 on 16 nodes.

Related Works

- The convergence of MLlib is rather slow in comparison with Spark LIBLINEAR.

- The reason is that the GD method is known to have slow convergence, while TRON enjoys fast quadratic local convergence for LR. Note that as MLlib requires more iterations to converge, the communication cost is also higher.

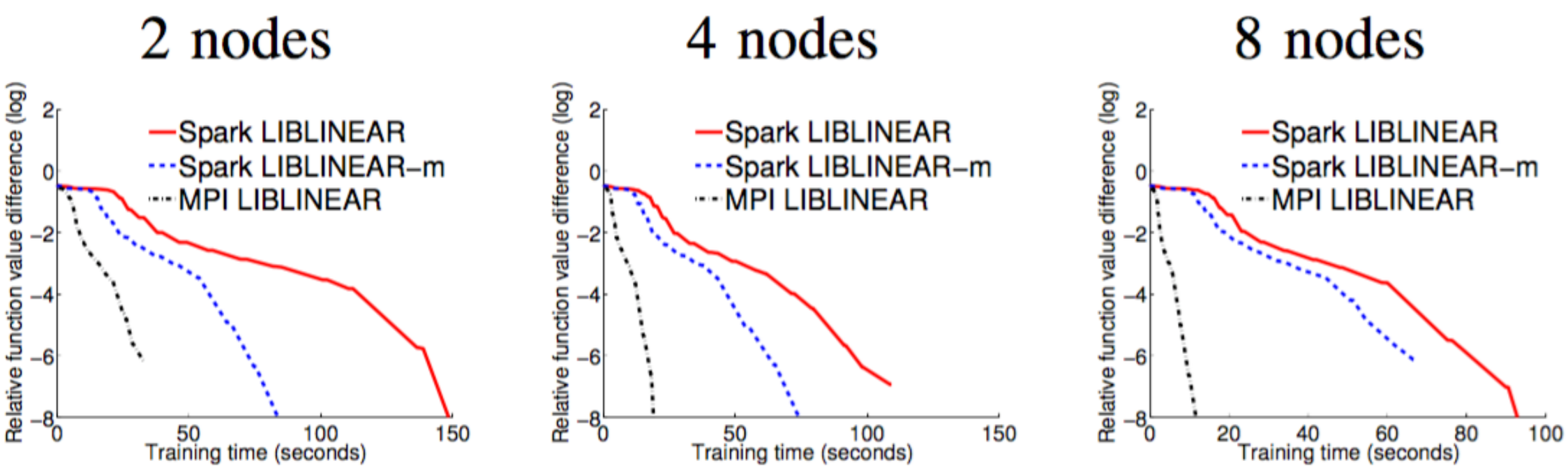
Related Works

- A C++/MPI implementation by Zhuang et al. (2014) of the distributed trust region Newton algorithm in this paper.
- No fault tolerance.
- Should be faster than our implementation:

More computational e cient: implemented in C++.
More communicational e cient: the slave-slave structure with all-reduce only communicates once per operation.

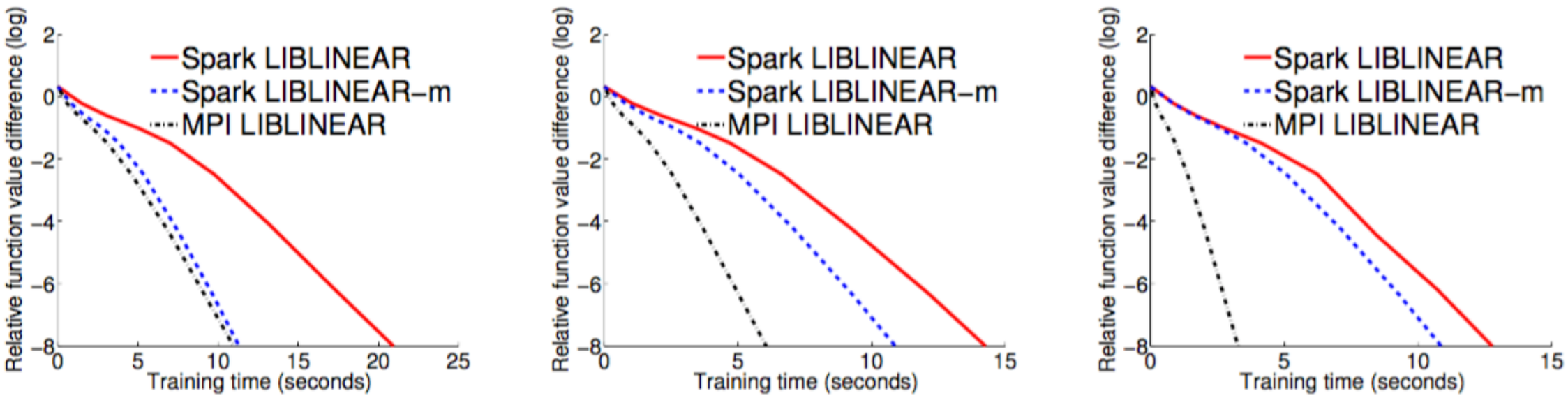
- Should be faster, but need to know how large is the difference.

Related Works



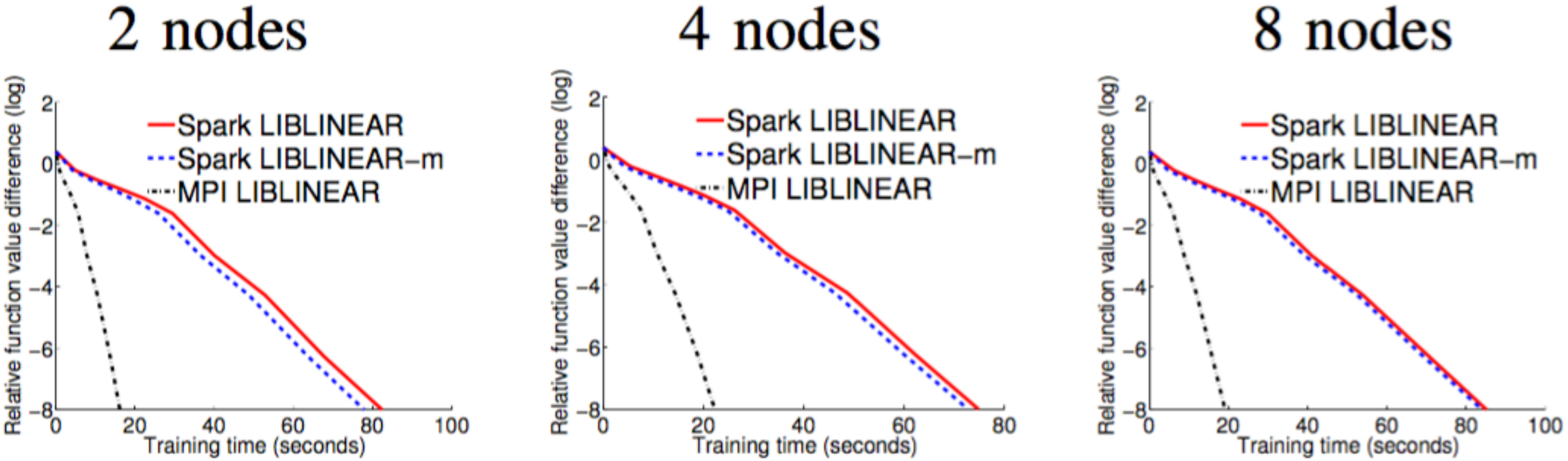
(a) covtype

m means multi-core



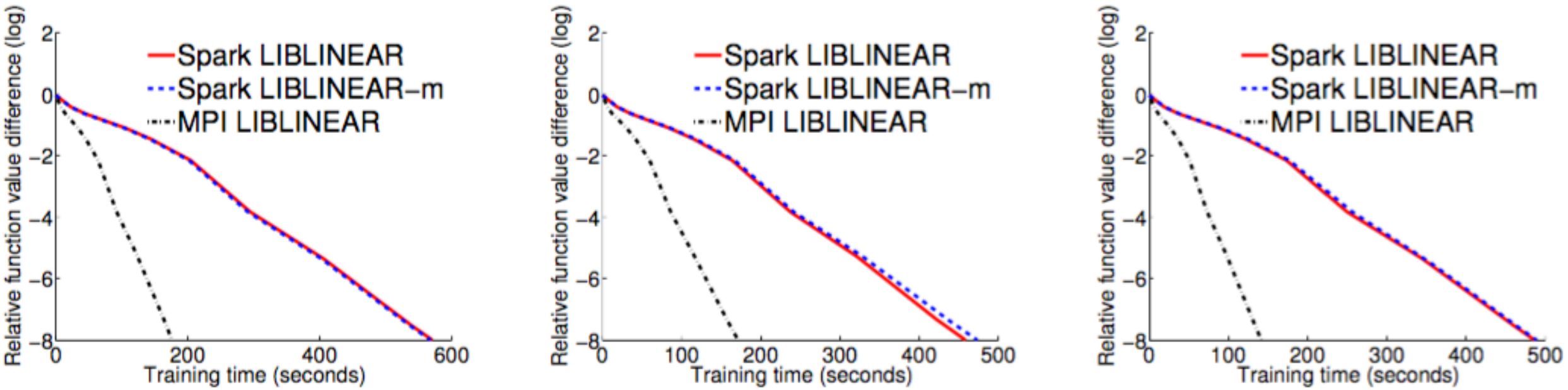
(b) webspam

Related Works



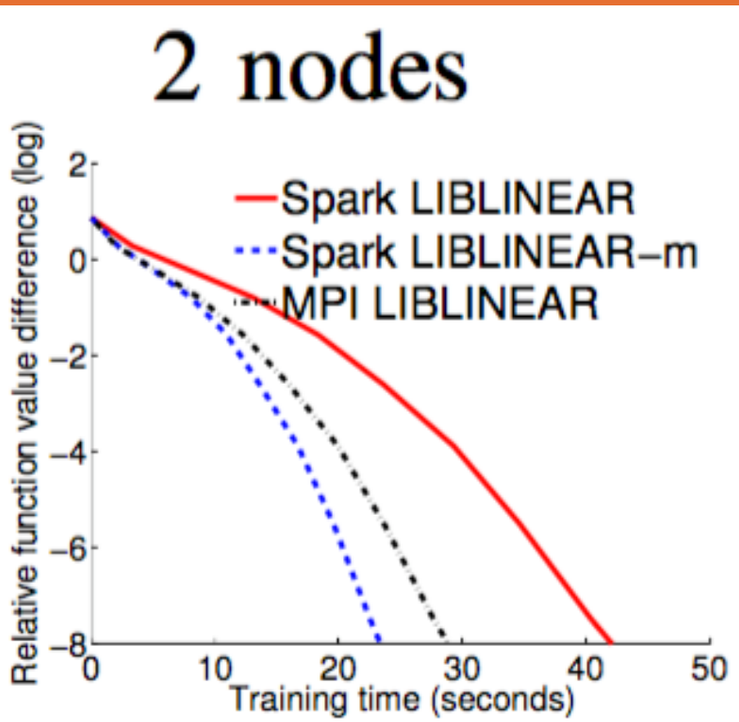
m means multi-core

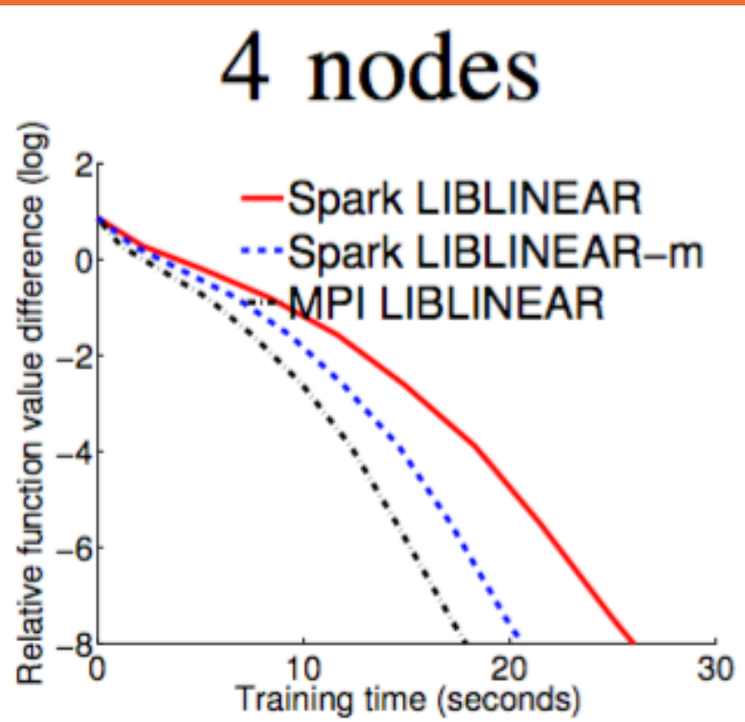
(c) yahoo-japan

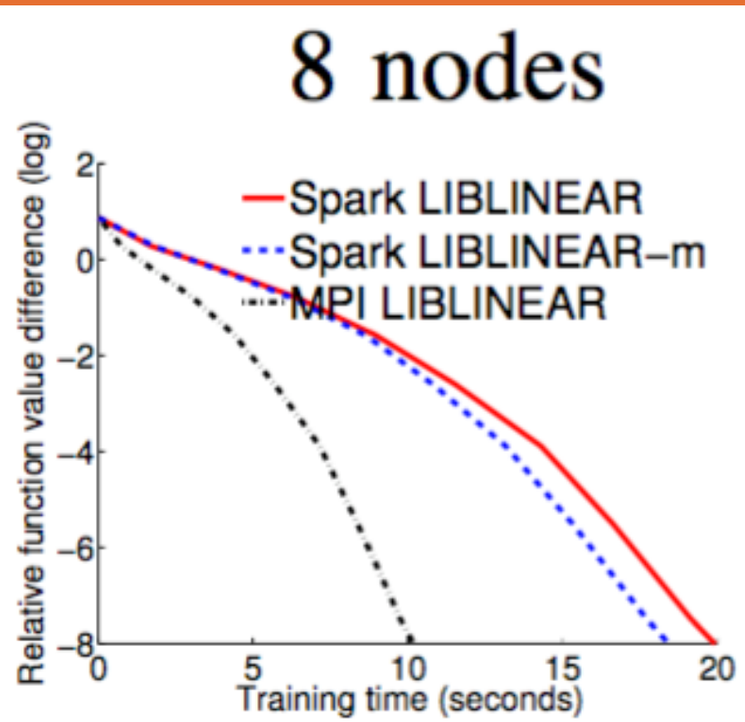


(d) yahoo-korea

Related Works

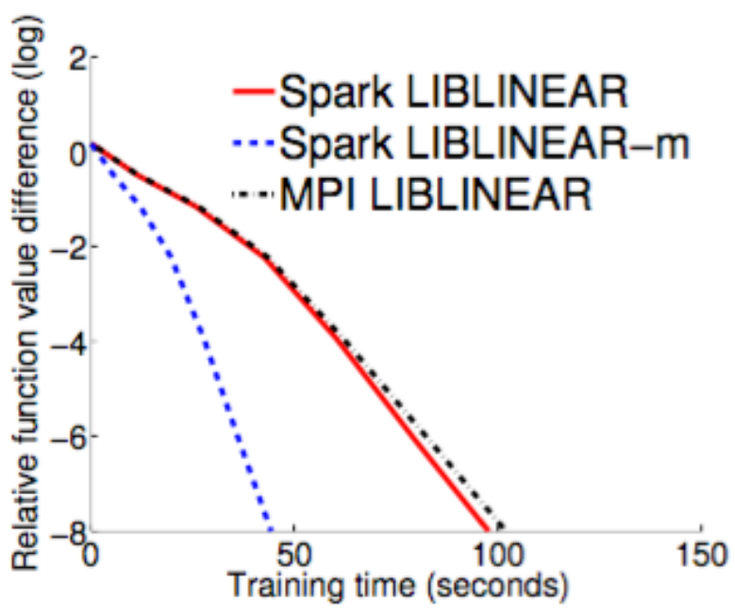


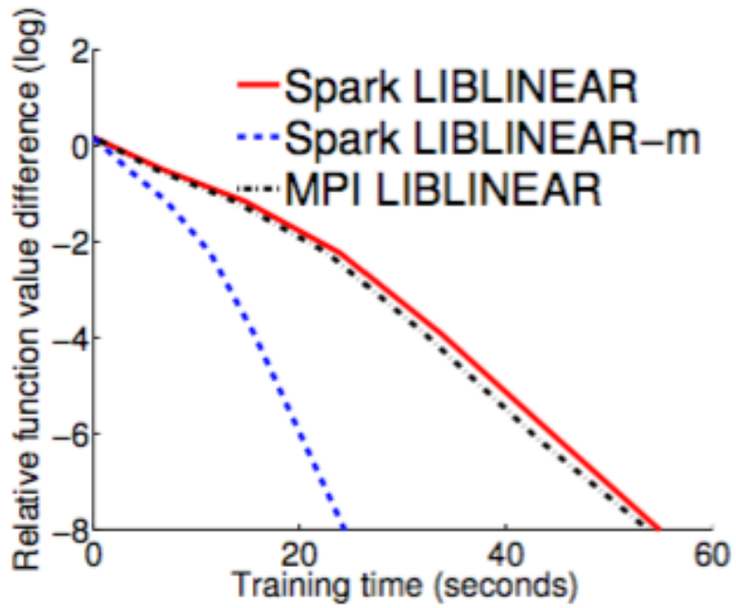




(e) rcv1t

m means multi-core





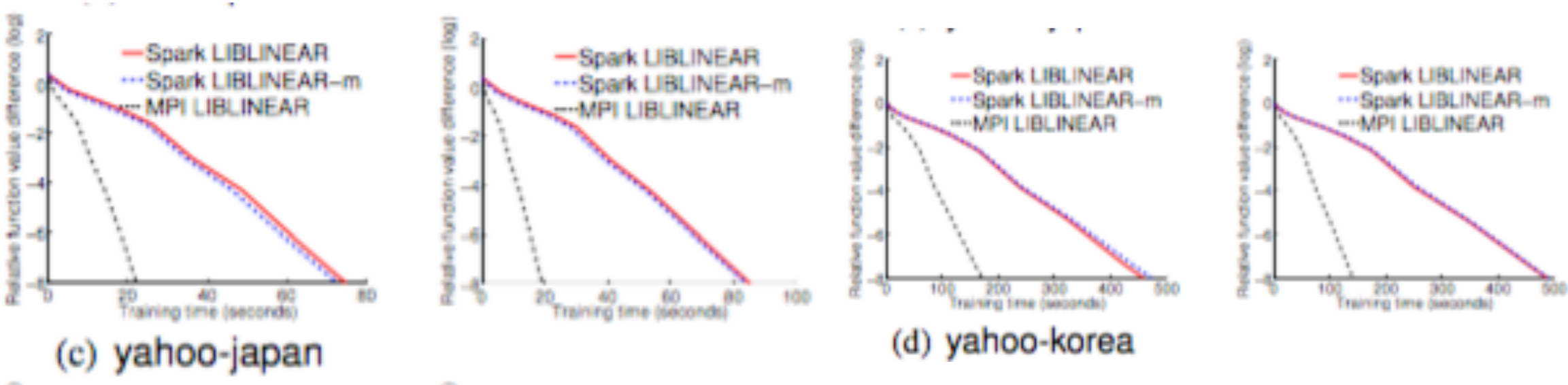
(f) epsilon

Related Works

- using multiple cores is not beneficial on yahoo-japan and yahoo-korea.

- A careful profiling shows that the bottleneck of the training time on these data sets is communication and using more cores does not reduce this cost.

m means multi-core



Outline

1. Introduction

2. Approach

3. Implementation design

4. Related Works

5. Discussions and Conclusions

Discussions and Conclusions

- Consider a distributed trust region Newton algorithm on Spark for training LR and linear SVM.

Discussions and Conclusions

- Consider a distributed trust region Newton algorithm on Spark for training LR and linear SVM.

- Many implementation issues are thoroughly studied with careful empirical examinations.

Discussions and Conclusions

- Consider a distributed trust region Newton algorithm on Spark for training LR and linear SVM.

- Many implementation issues are thoroughly studied with careful empirical examinations.

- Implementation in this paper on Spark is competitive with state-of-the-art packages. (2014)

Discussions and Conclusions

- Consider a distributed trust region Newton algorithm on Spark for training LR and linear SVM.

- Many implementation issues are thoroughly studied with careful empirical examinations.

- Implementation in this paper on Spark is competitive with state-of-the-art packages. (2014)

- Spark LIBLINEAR is an distributed extension of LIBLINEAR and it is available.