Parallel and Distributed Stochastic Learning

-Towards Scalable Learning for Big Data Intelligence

李武军

南京大学计算机科学与技术系
软件新技术国家重点实验室

liwujun@nju.edu.cn

May 14, 2018

Outline

Machine Learning

- Supervised Learning:
  Given a set of training examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, supervised learning tries to solve the following regularized empirical risk minimization problem:

$$\min_{\mathbf{w}} f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{w}),$$

  where $f_i(\mathbf{w})$ is the loss function (plus some regularization term) defined on example i, and $\mathbf{w}$ is the parameter to learn.

  Examples:
  - Logistic regression: $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}[\log(1+e^{-y_i \mathbf{x}_i^T \mathbf{w}}) + \frac{\lambda}{2}\|\mathbf{w}\|^2]$
  - SVM: $f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}[\max\{0, 1 - y_i \mathbf{x}_i^T \mathbf{w}\} + \frac{\lambda}{2}\|\mathbf{w}\|^2]$
  - Deep learning models

- Unsupervised Learning:
  Many unsupervised learning models, such as PCA and matrix factorization, can also be reformulated as similar problems.

Machine Learning for Big Data

For big data applications, first-order methods have become much more popular than other higher-order methods for learning (optimization).

Gradient descent methods are the most representative first-order methods.

- (Deterministic) gradient descent (GD):

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t[\frac{1}{n}\sum_{i=1}^{n}\nabla f_i(\mathbf{w}_t)],$$

  where t is the iteration number.
  - *Linear* convergence rate: $O(\rho^t)$
  - Iteration cost is $O(n)$
- Stochastic gradient descent (SGD): In the t^{th} iteration, randomly choosing an example $i_t \in \{1, 2, ..., n\}$, then update

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \eta_t \nabla f_{i_t}(\mathbf{w}_t)$$

  - Iteration cost is $O(1)$
  - The convergence rate is *sublinear*: $O(1/t)$

Stochastic Learning for Big Data

Researchers recently proposed improved versions of SGD:
SAG [Roux et al., NIPS 2012], SDCA [Shalev-Shwartz and Zhang, JMLR 2013], SVRG [Johnson and Zhang, NIPS 2013]

Number of gradient (∇f_i) evaluation to reach ϵ for smooth and strongly convex problems:

- GD: $O(n\kappa \log(\frac{1}{\epsilon}))$
- SGD: $O(\kappa(\frac{1}{\epsilon}))$
- SAG: $O(n \log(\frac{1}{\epsilon}))$ when $n \geq 8\kappa$
- SDCA: $O((n+\kappa) \log(\frac{1}{\epsilon}))$
- SVRG: $O((n+\kappa) \log(\frac{1}{\epsilon}))$

where $\kappa = \frac{L}{\mu} > 1$ is the condition number of the objective function.

Stochastic Learning:

- Stochastic GD
- Stochastic coordinate descent
- Stochastic dual coordinate ascent

Parallel and Distributed Stochastic Learning

To further improve the learning scalability (speed):

- Parallel stochastic learning:
  One machine with multiple cores and a shared memory
- Distributed stochastic learning:
  A cluster with multiple machines

Key issues: cooperation

- Parallel stochastic learning:
  lock vs. lock-free: waiting cost and lock cost
- Distributed stochastic learning:
  synchronous vs. asynchronous: waiting cost and communication cost

Our Contributions

- Parallel stochastic learning: AsySVRG
  Fast Asynchronous Parallel Stochastic Gradient Descent: A Lock-Free Approach with Convergence Guarantee.
- Distributed stochastic learning: SCOPE
  Scalable Composite Optimization for Learning

Outline

Motivation and Contribution

Motivation:

- Existing asynchronous parallel SGD: Hogwild! [Recht et al. 2011], and PASSCoDe [Hsieh, Yu, and Dhillon 2015]
- No parallel methods for SVRG.
- Lock-free: empirically effective, but no theoretical proof.

Contribution:

- A fast asynchronous method to parallelize SVRG, called AsySVRG.
- A lock-free parallel strategy for both read and write
- Linear convergence rate with theoretical proof
- Outperforms Hogwild! in experiments

AsySVRG: a multi-thread version of SVRG

Initialization: p threads, initialize $\mathbf{w}_0, \eta$;
for $t = 0, 1, 2, ...$ **do**
  $\mathbf{u}_0 = \mathbf{w}_t$;
  All threads parallelly compute the full gradient
  $\nabla f(\mathbf{u}_0) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\mathbf{u}_0)$;
  $\mathbf{u} = \mathbf{w}_t$;
  For each thread, do:
  for $m = 1$ to M **do**
    Read current value of $\mathbf{u}$, denoted as $\hat{\mathbf{u}}$, from the shared memory.
    And randomly pick up an i from $\{1, \ldots, n\}$;
    Compute the update vector: $\hat{\mathbf{v}} = \nabla f_i(\hat{\mathbf{u}}) - \nabla f_i(\mathbf{u}_0) + \nabla f(\mathbf{u}_0)$;
    $\mathbf{u} \leftarrow \mathbf{u} - \eta \hat{\mathbf{v}}$;
  end for
  Take $\mathbf{w}_{t+1}$ to be the current value of $\mathbf{u}$ in the shared memory;
end for

Lock-free Analysis

In all the GD or SGD methods to solve the objective function, the key step can be written as

$$\mathbf{u} \leftarrow \mathbf{u} + \Delta$$

Notation

- $\Delta_{i,j}$: the j^{th} update vector computed by the i^{th} thread;
- $\mathbf{u} \in \mathbb{R}^d$ and $\mathbf{u} = (u^{(1)}, u^{(2)}, \ldots, u^{(d)})$;
- $t_{i,j}^{(k)}$: the time when the operation "$u^{(k)} \leftarrow u^{(k)} + \Delta_{i,j}^{(k)}$" has been completed (Not the time when the operation begins);
- Assuming $\forall i, j$, $t_{i,j}^{(1)} < t_{i,j}^{(2)} < \ldots < t_{i,j}^{(d)}$, which can be easily guaranteed by programming

Lock-free Analysis: update sequence

Since $u^{(1)}$ can only be changed by at most one thread at any absolute time, these $t_{i,j}^{(1)}$ are different from each other. So we can:

- Sort these $t_{i,j}^{(1)}$ as $t_0^{(1)} < t_1^{(1)} < \ldots < t_{\tilde{M}-1}^{(1)}$ $(\tilde{M} = p \times M)$;
- $\Delta_0, \Delta_1, \ldots, \Delta_{\tilde{M}-1}$ are the corresponding update vectors.

Since it is lock-free, for each update vector Δ_m, the real update vector is $\mathbf{B}_m \Delta_m$ because of over-written. The $\mathbf{B}_m$ is a diagonal matrix whose diagonal elements are 0 or 1.
After all the inner-loop stop, we can get:

$$\mathbf{w}_{t+1} = \mathbf{u}_0 + \sum_{m=0}^{\tilde{M}-1} \mathbf{B}_m \Delta_m \tag{1}$$

Lock-free Analysis: update sequence

According to (1), we define a sequence $\{\mathbf{u}_m\}$ as follows:

$$\mathbf{u}_m = \mathbf{u}_0 + \sum_{i=0}^{m-1} \mathbf{B}_i \Delta_i \tag{2}$$

which means $\mathbf{u}_{m+1} = \mathbf{u}_m + \mathbf{B}_m \Delta_m$.

Note

The sequence $\{\mathbf{u}_m\}$ $(m = 1, 2, \ldots, \tilde{M} - 1)$ is synthetic, and the whole $\mathbf{u}_m$ may never occur in the shared memory. What we can get is only the final value of $\mathbf{u}_{\tilde{M}}$.

Lock-free Analysis: read sequence

Assume the old update vectors $\Delta_0, \Delta_1, \dots, \Delta_{a(m)-1}$ have been completely applied to $\mathbf{u}$ when one thread is reading the shared variable. At the same time, some new update vectors might be updating $\mathbf{u}$. So we can write $\hat{\mathbf{u}}_m$ read by the thread to compute Δ_m as follows:

$$\hat{\mathbf{u}}_m = \mathbf{u}_{a(m)} + \sum_{i=a(m)}^{b(m)} \mathbf{P}_{m,i-a(m)} \Delta_i$$

where $\mathbf{P}_{m,i-a(m)}$ is a diagonal matrix whose diagonal elements are 0 or 1.

According to the principle of the order, $\Delta_i (i \geq m)$ should not be read by $\hat{\mathbf{u}}_m$. So $b(m) < m$, which means:

$$\hat{\mathbf{u}}_m = \mathbf{u}_{a(m)} + \sum_{i=a(m)}^{m-1} \mathbf{P}_{m,i-a(m)} \Delta_i$$

Convergence Result for Strongly Convex Problems

With some assumptions, our algorithm gets a linear convergence rate for strongly convex problems:

$$\mathbb{E}f(\mathbf{w}_{t+1}) - f(\mathbf{w}_*) \leq (c_1^{\tilde{M}} + \frac{c_2}{1-c_1})(\mathbb{E}f(\mathbf{w}_t) - f(\mathbf{w}_*)),$$

where $c_1 = 1 - \alpha\eta\mu + c_2$ and $c_2 = \eta^2(\frac{8\tau L^3\eta\rho^2(\rho^\tau - 1)}{\rho - 1} + 2L^2\rho)$, $\tilde{M} = p \times M$ is the total number of iterations of the inner-loop.

Note
Since it is lock-free, we do not know the exact $\mathbf{B}_m$ and we can not take the average sum of $\mathbf{B}_m\mathbf{u}_m$ to be $\mathbf{w}_{t+1}$.

Convergence Result for Non-Convex Problems

With some assumptions, our algorithm gets a sub-linear convergence rate for non-convex problems:

$$\frac{1}{T\tilde{M}}\sum_{t=0}^{T-1}\sum_{m=0}^{\tilde{M}-1}\mathbb{E}\|\nabla f(\mathbf{u}_{t,m})\|^2 \leq \frac{\mathbb{E}f(\mathbf{w}_0)-\mathbb{E}f(\mathbf{w}_T)}{T\tilde{M}\gamma}.$$

Similar to the analysis for strongly convex problems, we construct an equivalent write sequence $\{\mathbf{u}_{t,m}\}$ for the t^{th} outer-loop:

$$\mathbf{u}_{t,0} = \mathbf{w}_t,$$
$$\mathbf{u}_{t,m+1} = \mathbf{u}_{t,m} - \eta\mathbf{B}_{t,m}\hat{\mathbf{v}}_{t,m},$$

where $\hat{\mathbf{v}}_{t,m} = \nabla f_{i_{t,m}}(\hat{\mathbf{u}}_{t,m}) - \nabla f_{i_{t,m}}(\mathbf{u}_{t,0}) + \nabla f(\mathbf{u}_{t,0})$. $\mathbf{B}_{t,m}$ is a diagonal matrix whose diagonal entries are 0 or 1. And $\hat{\mathbf{u}}_{t,m}$ is read by the thread who computes $\hat{\mathbf{v}}_{t,m}$.

Experiments - Convex Case

Experimental platform: A server with 12 Intel cores and 64G memory.

Model: Logistic regression with $L2$-norm

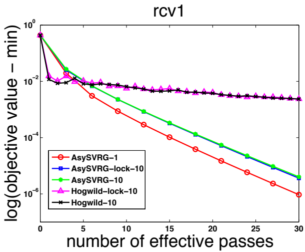
$$f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\log(1+e^{-y_i\mathbf{x}_i^T\mathbf{w}}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

Data set

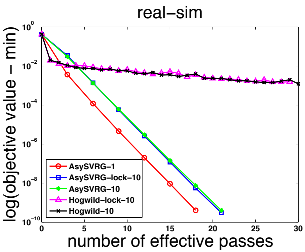
dataset	instances	features	memory	type
rcv1	20,242	47,236	36M	sparse
real-sim	72,309	20,958	90M	sparse
news20	19,996	1,355,191	140M	sparse
epsilon	400,000	2,000	11G	dense

We set $\lambda = 10^{-4}$.

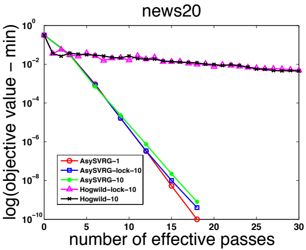
Experiments: Computation Cost



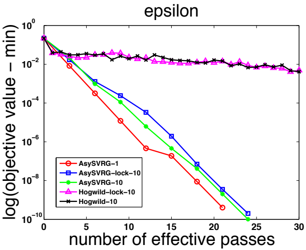
(a) rcv1



(b) realsim

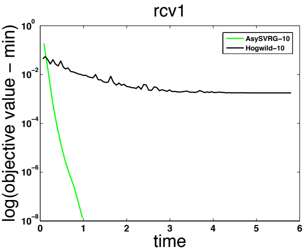


(c) news20

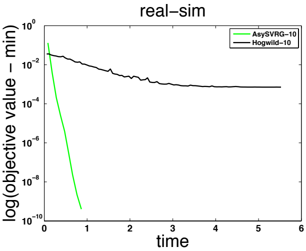


(d) epsilon

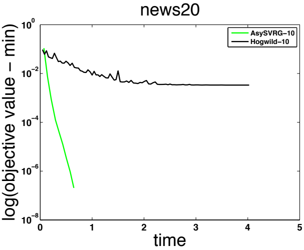
Experiments: Total Time Cost



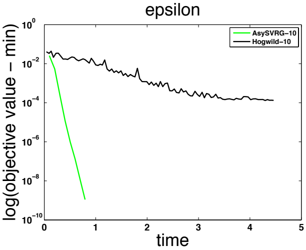
(a) rcv1



(b) realsim

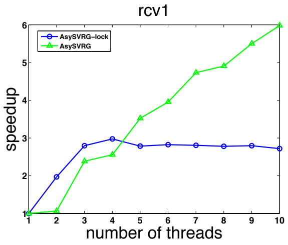


(c) news20

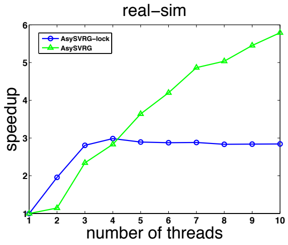


(d) epsilon

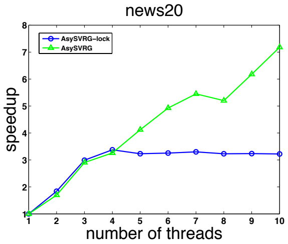
Experiments: Speed up



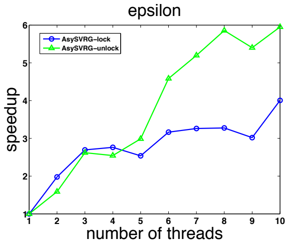
(a) rcv1



(b) realsim



(c) news20



(d) epsilon

Experiments - Non-Convex Case

Experimental platform: A server with 12 Intel cores and 64G memory.

Model: A fully-connected neural network

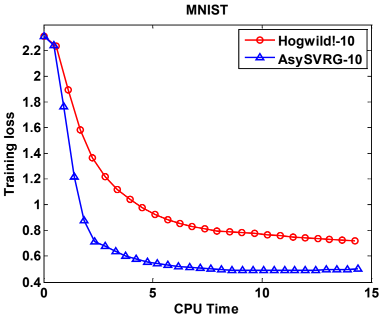
One hidden layer with 100 nodes, sigmoid function for activation.

$$f(\mathbf{w}, \mathbf{b}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\mathbf{1}\{y_i = k\}\log o_i^{(k)} + \frac{\lambda}{2}\|\mathbf{w}\|^2,$$

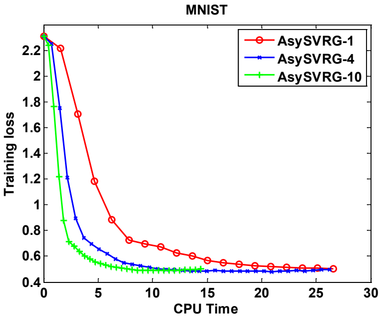
where $\mathbf{w}$ is the weights of the neural network, $\mathbf{b}$ is the bias, y_i is the label of instance $\mathbf{x}_i$, $o_i^{(k)}$ is the output corresponding to $\mathbf{x}_i$, K is the total number of class labels.

We set $\lambda = 10^{-3}$.

Experiments- Non-Convex Case



(a) MNIST



(b) AsySVRG on MNIST

Figure: Non-Convex Case.

Outline

Motivation and Contribution

Motivation:

- Bulk synchronous parallel (BSP) models, such as MapReduce, are commonly considered to be inefficient for distributed stochastic learning. Is there any technique to solve the issues of BSP models?

Contribution:

- A novel distributed stochastic learning method, called scalable composite optimization for learning (SCOPE), with BSP models
- Both computation-efficient and communication-efficient
- Linear convergence rate with theoretical proof
- SCOPE implemented on Spark outperforms other state-of-the-art distributed learning methods on Spark
- Parameter Server with SCOPE outperforms other Parameter Server platforms with stale synchronous parallel (SSP) or asynchronous parallel (ASP) models

Framework of SCOPE on Spark

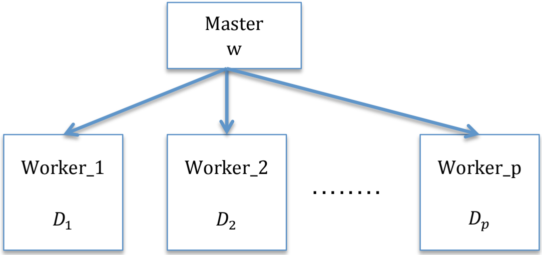


Figure: Distributed framework of SCOPE on Spark.

Optimization Algorithm: Master

Task of Master in SCOPE:

Initialization: p Workers, $\mathbf{w}_0$;
for $t = 0, 1, 2, \ldots, T$ **do**
  Send $\mathbf{w}_t$ to the Workers;
  Wait until it receives $\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_p$ from the p Workers;
  Compute the full gradient $\mathbf{z} = \frac{1}{n} \sum_{k=1}^{p} \mathbf{z}_k$, and then send $\mathbf{z}$ to each Worker;
  Wait until it receives $\tilde{\mathbf{u}}_1, \tilde{\mathbf{u}}_2, \ldots, \tilde{\mathbf{u}}_p$ from the p Workers;
  Compute $\mathbf{w}_{t+1} = \frac{1}{p} \sum_{k=1}^{p} \tilde{\mathbf{u}}_k$;
end for

Optimization Algorithm: Workers

Task of Workers in SCOPE:

Initialization: initialize η and $c > 0$;
For the Worker_k:
for $t = 0, 1, 2, \ldots, T$ **do**
  Wait until it gets the newest parameter $\mathbf{w}_t$ from the Master;
  Let $\mathbf{u}_{k,0} = \mathbf{w}_t$, compute the local gradient sum $\mathbf{z}_k = \sum_{i \in \mathcal{D}_k} \nabla f_i(\mathbf{w}_t)$, and then send $\mathbf{z}_k$ to the Master;
  Wait until it gets the full gradient $\mathbf{z}$ from the Master;
  for $m = 0$ to $M - 1$ **do**
    Randomly pick up an instance with index $i_{k,m}$ from $\mathcal{D}_k$;
    $\mathbf{u}_{k,m+1} = \mathbf{u}_{k,m} - \eta(\nabla f_{i_{k,m}}(\mathbf{u}_{k,m}) - \nabla f_{i_{k,m}}(\mathbf{w}_t) + \mathbf{z} + c(\mathbf{u}_{k,m} - \mathbf{w}_t))$;
  end for
  Send $\mathbf{u}_{k,M}$ or $\frac{1}{M}\sum_{m=1}^{M} \mathbf{u}_{k,m}$, which is called the locally updated parameter and denoted as $\tilde{\mathbf{u}}_k$, to the Master;
end for

Convergence

Let $\alpha = 1 - \eta(2\mu + c) < 1$, $\beta = c\eta + 3L^2\eta^2$ and $\alpha + \beta < 1$. We have the following theorems:

Theorem

If we take $\mathbf{w}_{t+1} = \frac{1}{p}\sum_{k=1}^{p}\mathbf{u}_{k,M}$, *then we can get the following convergence result:*

$$\mathbb{E}\|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2 \leq (\alpha^M + \frac{\beta}{1-\alpha})\mathbb{E}\|\mathbf{w}_t - \mathbf{w}^*\|^2.$$

Theorem

If we take $\mathbf{w}_{t+1} = \frac{1}{p}\sum_{k=1}^{p}\tilde{\mathbf{u}}_k$ *with* $\tilde{\mathbf{u}}_k = \frac{1}{M}\sum_{m=1}^{M}\mathbf{u}_{k,m}$, *then we can get the following convergence result:*

$$\mathbb{E}\|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2 \leq (\frac{1}{M(1-\alpha)} + \frac{\beta}{1-\alpha})\mathbb{E}\|\mathbf{w}_t - \mathbf{w}^*\|^2.$$

Communication Cost

- Traditional mini-batch based methods: $O(Tn)$
- SCOPE: $O(T)$

Experiment

Logistic regression (LR) with a L_2-norm regularization term:

$$f(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\left[\log(1+e^{-y_i\mathbf{x}_i^T\mathbf{w}}) + \frac{\lambda}{2}\|\mathbf{w}\|^2\right].$$

Table: Datasets for evaluation.

	♯instances	♯features	memory	λ
MNIST-8M	8,100,000	784	39G	1e-4
epsilon	400,000	2,000	11G	1e-4
KDD12	73,209,277	1,427,495	21G	1e-4
Data-A	106,691,093	320	260G	1e-6

Two Spark clusters with Intel CPUs:

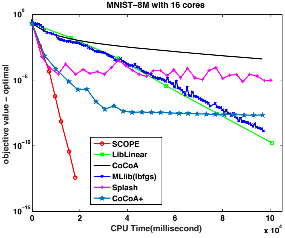
- small: 1 Master and 16 Workers
- large: 1 Master and 128 Workers

Experiment

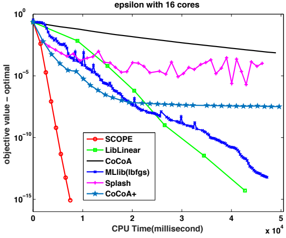
Baselines:

- MLlib [Meng et al., 2015]: MLlib is an open source library for distributed machine learning on Spark. We compare our method with distributed lbfgs for MLlib, which is a batch learning method and faster than the SGD version of MLlib.
- LibLinear [Lin et al., 2014a]: LibLinear is a distributed Newton method, which is also a batch learning method.
- Splash [Zhang and Jordan, 2015]: Splash is a distributed SGD method by using the local learning strategy to reduce communication cost.
- CoCoA [Jaggi et al., 2014]: CoCoA is a distributed dual coordinate ascent method.
- CoCoA+ [Ma et al., 2015]: CoCoA+ is an improved version of CoCoA. CoCoA+ adopts adding rather than average to combine local updates.

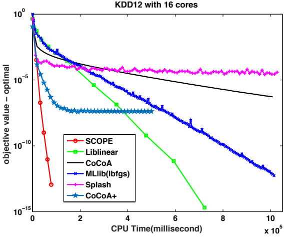
Experiment



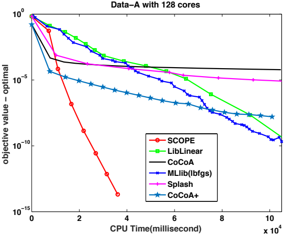
(a) MNIST-8M



(b) epsilon

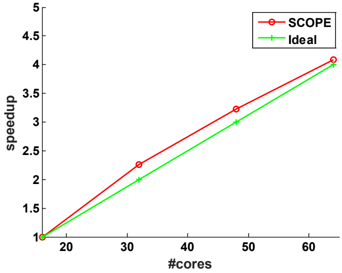


(c) KDD12

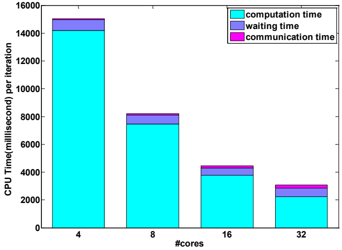


(d) Data-A

Experiment



(e) Speedup



(f) Synchronization cost

Parameter Server with SCOPE (PS-SCOPE)

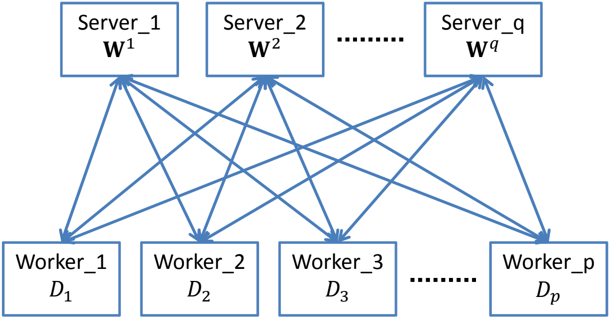
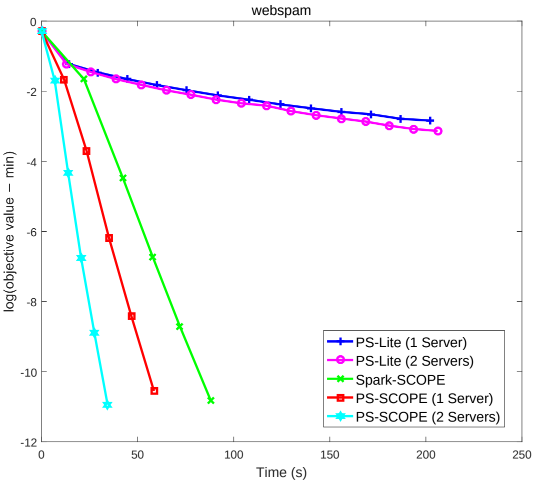


Figure: Distributed framework of PS-SCOPE.

Experiment for PS-SCOPE



PS-Lite is the parameter server proposed in [Mu Li, et al., OSDI 2014] with SSP/ASP models

Outline

Conclusion

- Stochastic learning is becoming popular for big data machine learning.
- Lock-free strategy is the key to get a good speedup in parallel stochastic learning.
- With properly designed techniques, BSP models are also efficient for distributed stochastic learning.

Future Work

Open source project:

LIBBLE: A library for big learning

- LIBBLE-Spark: https://github.com/LIBBLE/LIBBLE-Spark/
  - Classification: LR, SVM, LR with L1-norm Regularization
  - Regression: Linear Regression, Lasso
  - Generalized Linear Models: with L2-norm/L1-norm Regularization
  - Dimensionality Reduction: PCA, SVD
  - Matrix Factorization
  - Clustering: K-Means
- LIBBLE-PS: https://github.com/LIBBLE/LIBBLE-PS
- LIBBLE-MultiThread: Parallel Knowledge Graph Embedding https://github.com/LIBBLE/LIBBLE-MultiThread/tree/master/ParaGraphE
- LIBBLE-DeepLearning

References

- Shen-Yi Zhao, Ru Xiang, Ying-Hao Shi, Peng Gao, Wu-Jun Li. SCOPE: Scalable Composite Optimization for Learning on Spark. Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI), 2017.
- Shen-Yi Zhao, Gong-Duo Zhang, Wu-Jun Li. Lock-Free Optimization for Non-Convex Problems. Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI), 2017.
- Shen-Yi Zhao, Wu-Jun Li. Fast Asynchronous Parallel Stochastic Gradient Descent: A Lock-Free Approach with Convergence Guarantee. Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI), 2016.

Q & A

Thanks!