Learning with Staleness

Wei Dai

March 2018
CMU-ML-18-100

Machine Learning Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Eric P. Xing, Chair
Gregory R. Ganger
Andrew Pavlo
Joseph E. Gonzalez (UC Berkeley)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Copyright © 2018 Wei Dai

This research was sponsored by the National Science Foundation award CCF1629559, the Air Force Research Laboratory award FA87501220324, the Office of Naval Research award N000141410684, and the Defense Advanced Research Project Agency awards FA872105C0003 and FA870215D0002.

Keywords: Large Scale Machine Learning, Distributed Optimization Method, Distributed System, Parameter Server

Dedicated to my parents

for their endless love, support, and encouragement

Acknowledgments

The key person who made all this possible is my advisor, Eric Xing, who has provided guidance and support throughout my PhD study. I want to especially thank him for trusting in me and letting me pursue my interests in diverse research topics. This intellectual freedom has allowed me to find many interesting things to work on, and I've definitely learned a lot more than I would without the broader exploration.

I also want to thank other thesis committee members: Greg Ganger, Andy Pavlo, and Joseph Gonzalez. They asked me tough questions and gave me lots of advice, so much that my thesis title has changed entirely after my thesis proposal—for the better. I want to give a special acknowledgement to Garth Gibson and Phil Gibbons. They have been two of the pillars in the BigLearning meetings that were really formative for my time at CMU. Many of the ideas in this thesis come from those weekly meetings that stretched over a remarkable span of 4 years.

I appreciate the helpful discussions and the camaraderie—especially during the paper deadline seasons—with my collaborators: Jinliang Wei, Abhimanu Kumar, Qirong Ho, Henggang Cui, Yi Zhou, Hao Zhang, Yu-Xiang Wang, Veeranjaneyulu Sadhanala, Willie Neiswanger, Xun Zheng, Jin Kyu Kim, Seunghak Lee, Yaoliang Yu, and Jim Cipar. I have learned so much from all of them, and the heated debate with some of them are a fond memory of my PhD years.

I want to thank my friends who accompany me through the highs and lows during my study. Friends in my PhD class year and in MLD have been a fun bunch, especially during the late nights leading up to course project and paper deadlines. Friends at my church, Oakland International Fellowship, and my small group, Psalm 22:27, are my precious connections to the rest of the "normal" world where people actually do not all do research. I also want to give at shout-out to my housemates who have colored my life with little dramas and stories. I love you all!

My brother Chia Dai deserves his own paragraph. His world class free food gathering skill has amazed my office mates and tided me over for more than a few meals during my study.

Thank you Jesus for opening the door for me to come to CMU, and for bringing all the wonderful people into my life.

Lastly, I want to thank my parents for their love and support throughout my life, including the past 11 years away from home in a different country. You have always challenged me to keep on going when I don't think I can. This thesis would not be possible without your continual encouragement. Thank you!

Abstract

A fundamental assumption behind most machine learning (ML) algorithms and analyses is the sequential execution. That is, any update to the ML model can be immediately applied and the new model is always available for the next algorithmic step. This basic assumption, however, can be costly to realize, when the computation is carried out across multiple machines, linked by commodity networks that are usually 10^4 times slower than the memory speed due to fundamental hardware limitations. As a result, concurrent ML computation in the distributed settings often needs to handle delayed updates and perform learning in the presence of staleness.

This thesis characterizes learning with staleness from three directions: (1) We extend the theoretical analyses of a number of classical ML algorithms, including stochastic gradient descent, proximal gradient descent on non-convex problems, and Frank-Wolfe algorithms, to explicitly incorporate staleness into their convergence characterizations. (2) We conduct simulation and large-scale distributed experiments to study the empirical effects of staleness on ML algorithms under indeterministic executions. Our results reveal that staleness is a key parameter governing the convergence speed for all considered ML algorithms, with varied ramifications. (3) We design staleness-minimizing parameter server systems by optimizing synchronization methods to effectively reduce the runtime staleness. The proposed optimization of a bounded consistency model utilizes the additional network bandwidths to communicate updates eagerly, relieving users of the burden to tune the staleness level. By minimizing staleness at the framework level, our system stabilizes diverging optimization paths and substantially accelerates convergence across ML algorithms without any modification to the ML programs.

Contents

1 Introduction **1**
1.1 Thesis Statement . . . 2
1.2 Backgrounds . . . 3
1.2.1 Iterative-Convergent ML . . . 3
1.2.2 Data Parallelism and Parameter Server . . . 4
1.2.3 Staleness Trade-offs . . . 5
1.3 Contributions and Outline . . . 6

2 Effects of Staleness on Machine Learning **9**
2.1 Asynchrony or Not? . . . 9
2.2 Scope and Methods . . . 11
2.2.1 Models and Algorithms . . . 11
2.2.2 Datasets . . . 16
2.3 Experiments . . . 17
2.3.1 System Configurations . . . 17
2.3.2 Simulation Model . . . 18
2.3.3 Deep Neural Networks with Staleness . . . 18
2.3.4 Staleness and Model Complexity . . . 20
2.3.5 Gradient Coherence . . . 25
2.3.6 Matrix Factorization with Staleness . . . 28
2.3.7 Variational Autoencoder with Staleness . . . 32
2.3.8 Latent Dirichlet Allocation with Staleness . . . 35
2.4 Staleness and ML Algorithms . . . 38

3 Analysis of Consistency Models **39**
3.1 Preliminaries . . . 39
3.2 Consistency Models for Parameter Servers . . . 40
3.2.1 Bulk Synchronous Parallel (BSP) . . . 40
3.2.2 Total Asynchronous Parallel (TAP) . . . 41
3.2.3 Stale Synchronous Parallel (SSP) . . . 41
3.2.4 Value-bounded Asynchronous Parallel (VAP) . . . 42
3.3 Theoretical Analysis . . . 44
3.3.1 SGD for Low Rank Matrix Factorization . . . 44

3.3.2 Preliminaries . . . 44
3.3.3 Theorems for VAP Consistency . . . 45
3.3.4 Theorems for SSP Consistency . . . 46
3.3.5 Comparison of VAP and ESSP . . . 49
3.4 Bösen System Overview . . . 50
3.4.1 API and Bounded Staleness Consistency . . . 50
3.4.2 System Architecture . . . 52
3.5 Evaluation . . . 54
3.5.1 Experiment Details . . . 55
3.5.2 System Evaluations . . . 55
3.5.3 ML Evaluation and Discussions . . . 57
3.6 Additional Related Work . . . 58

4 Model Parallel Learning with Staleness 61
4.1 Introduction . . . 61
4.2 Preliminaries . . . 63
4.3 Problem Formulation . . . 64
4.4 Convergence Analysis . . . 67
4.5 Economical Implementation . . . 70
4.6 Experiments . . . 72
4.6.1 Group Lasso . . . 72
4.6.2 Large-scale Lasso . . . 74
4.7 Additional Related Work . . . 75

5 Staleness in Parallel Frank-Wolfe Algorithms 77
5.1 Introduction . . . 77
5.2 Algorithm . . . 79
5.3 Analysis . . . 80
5.3.1 Main convergence results . . . 81
5.3.2 Effect of parallelism / mini-batching . . . 83
5.3.3 Convergence with delayed updates . . . 84
5.4 Experiments . . . 85
5.4.1 Minibatches of Data . . . 86
5.4.2 Shared Memory Parallel Workers . . . 87
5.4.3 Performance gain with asynchronous updates . . . 87
5.4.4 Convergence under unbounded heavy-tailed delay . . . 89
5.5 Additional Related Work . . . 89
5.6 Conclusion . . . 90

6 Conclusion and Future Work 93
6.1 Future Work . . . 95

7 The Debate: Synchronous vs Non-Synchronous Training for Machine Learning 97
7.1 Async Isn't Aways More Stale than Sync . . . 99

- 7.2 Computation-to-Communication Ratio . . . 101
- 7.3 Staleness and Momentum . . . 103
- 7.4 Staleness and the Convergence Dynamics . . . 104
- 7.5 Non-synchronous Training Gets to an “Okay” Model Faster than Synchronous Training . . . 107
- 7.6 Staleness’ Effects on the Final Model Quality . . . 109
- 7.7 Looking to The Future . . . 113

Appendices **117**

A Appendix for Chapter 2 **119**

B Appendix for Chapter 3 **127**
- B.1 Proof of Theorem 3.1 . . . 127
- B.2 Proof of Theorem 3.4 . . . 130
- B.3 Proof of Theorem 3.5 . . . 132
- B.4 Proof of Theorem 3.2 . . . 135

C Appendix for Chapter 4 **137**
- C.1 Proof of Theorem 4.1 . . . 137
- C.2 Proof of Theorem 4.2 . . . 143
- C.3 Proof of Lemma 4.1 . . . 145
- C.4 Proof of Example 1 . . . 148
- C.5 Proof of Theorem 4.3 . . . 151

D Appendix for Chapter 5 **153**
- D.1 Convergence analysis . . . 153
  - D.1.1 Primal Convergence . . . 153
  - D.1.2 Convergence of the surrogate duality gap . . . 155
- D.2 Proofs of other technical results . . . 160
  - D.2.1 Pseudocode for the Multicore Shared Memory Architecture . . . 162
- D.3 Application to Structural SVM . . . 164
- D.4 Other technical results and discussions . . . 165
  - D.4.1 Oracle assumption and heterogeneous blocks . . . 165
  - D.4.2 Controlling collisions in distributed setting . . . 166
  - D.4.3 Curvature and Lipschitz Constant . . . 168
  - D.4.4 Examples and illustrations . . . 169
  - D.4.5 Comparison to parallel block coordinate descent . . . 170

Bibliography **173**

List of Figures

1.1 Staleness is both a system and an ML algorithm parameter, and a primary way that distributed systems and ML algorithms interact. 3

1.2 (a) Illustration of data parallelism. (b) Parameter server topology. Servers and clients interact via a bipartite topology. Note that this is the logical topology; physically the servers can collocate with the clients to utilize CPU on all machines. 5

1.3 Illustration of the trade-off due to staleness (x-axis). The system throughput (iterations per second) generally improves with higher staleness (green curve), as less synchronization is needed. On the other hand, using more stale version of the ML model will result in lower quality updates computed in each iteration of the ML algorithms (blue curve). The goal is to maximize the convergence per second (red curve) within these complex trade-offs. 6

2.1 Variational Autoencoder (VAE) at training time, assuming continuous input $\boldsymbol{x}$ and isotropic Gaussian prior $p(\boldsymbol{z}) \sim \mathcal{N}(0, \boldsymbol{I})$. The encoder $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ encodes input $\boldsymbol{x}$ to mean $\mu(\boldsymbol{x})$ and variance $\sigma^2(\boldsymbol{x})$ such that the sample $\boldsymbol{z} = \mu(\boldsymbol{x}) + \sigma(\boldsymbol{x}) \odot \epsilon$ is sampled according to distributed $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ ($\epsilon \sim \mathcal{N}(0, \boldsymbol{I})$). Thanks to the reparametric trick [76], the loss functions are differentiable with respect to θ, ϕ and can be back-propagated throughout to compute the gradient. Minimization objectives are denoted in dashed blue boxes. In our experiments we use DNNs as the encoder and the decoder. 16

2.2 The number of batches to reach 95% test accuracy using 1 hidden layer and 1 worker. Each color represents a batch size, while each cluster corresponds to the maximum staleness in the simulation model. 19

2.3 The number of batches to reach 95% test accuracy using 1 hidden layer and 1 worker, using AMSGrad [122] and varying batch sizes. $s = 32$ did not converge and thus not shown. 20

2.4 The number of batches to reach 95% test accuracy using 1 hidden layer and 1 worker, respectively normalized by $s = 0$. The panel is a normalized counterpart of Fig. 2.2 21

2.5 The number of batches to reach 92% test accuracy using Deep Neural Networks with varying numbers of hidden layers under 1 worker. We consider several variants of SGD algorithms (a)-(e). Note that with depth 0 the model reduces to multi-class logistic regression (MLR), which is convex. The numbers are averaged over 5 randomized runs. We omit the bars whenever convergence is not achieved within the experiment horizon (77824 batches), such as SGD with momentum at depth 6 and $s = 32$. We do not include FTRL result due to the unstable convergence. The unnormalized version is provided in the appendix (Fig. A.1). 22

2.6 The number of batches to reach 92% test accuracy with Adam and SGD on 1, 8, 16 workers with varying staleness. Each depth is normalized by the staleness 0's values, respectively. The numbers are average over 5 randomized runs. Depth 0 under SGD with 8 and 16 workers did not converge within the experiment horizon (77824 batches) for all staleness values, and is thus not shown. The unnormalized version is in Appendix (Fig. A.2) 24

2.7 Gradient coherence for DNNs with varying depths (depth 0–6) optimized by the Adam optimization using 1 worker and no staleness ($s = 0$). The x-axis is $k = 1, ..., 32$. Here we show cosine distance up to 32 batches back. (a) is a snapshot taken from the first 512 batches, while (b) is taken from 512 batches starting from batch 25088 after the algorithm has converged. The error bars around the means represent 1 standard deviation computed from 5 randomized runs. 26

2.8 Gradient coherence for DNNs with varying depths (depth 0–6) optimized by SGD using 1 worker and no staleness ($s = 0$). The x-axis is $k = 1, ..., 32$. Here we show cosine distance up to 32 batches back. (a) is a snapshot taken from the first 512 batches, while (b) is taken from 512 batches starting from batch 25088 after the algorithms have converged (Fig. 2.7(b), Fig. 2.8(b)). The error bars around the means represent 1 standard deviation computed from 5 randomized runs. 27

2.9 Convergence of Matrix Factorization (MF) using 4 and 8 workers, with staleness ranging from 0 to 50. We use the number of batches processed across all workers as the logical time. Shaded area represents 1 standard deviation around the means (represented by the curves) computed on 5 randomized runs. 30

2.10 The number of batches to reach training loss of 0.5 for Matrix Factorization (MF) optimized by SGD. Mean and error bar (representing 1 standard deviation) are based on 5 randomized runs. 31

2.11 The number of batches to reach training loss of 0.5 for Matrix Factorization (MF), normalized by the values of staleness 0 of each worker setting, respectively. This is the normalized version of Fig. 2.10. 31

2.12 The number of batches to reach test loss 130 by Variational Autoencoders (VAEs) on 1 worker, under staleness 0 to 16. We consider VAEs with depth 1, 2, and 3 (the number of layers in the encoder and the decoder networks). The numbers of batches are normalized by $s = 0$ for each VAE depth, respectively. Configurations that do not converge to the desired test loss are omitted, such as Adam optimization for VAE with depth 3 and $s = 16$. The unnormalized version is provided in the appendix (Fig. A.5). 33

2.13 Gradient coherence for VAEs with varying depths (depth 1∼3) optimized by SGD optimization using 1 worker with no staleness ($s = 0$). The x-axis is $k = 1, ..., 32$. Here we show cosine distance up to 32 batches back. (a) is a snapshot taken from the first 512 batches, while (b) is taken from 512 batches starting from batch 25088 after algorithms have converged. The error bars around the means represent 1 standard deviation computed from 5 randomized runs. 34

2.14 Convergence of LDA log likelihood using 10 topics with respect to the number of documents processed by Gibbs sampling, with varying staleness and number of workers. The shaded regions are 1 standard deviation around the means (curves) based on 5 randomized runs. 36

2.15 Convergence of LDA log likelihood using 100 topics with respect to the number of documents processed by Gibbs sampling, with varying staleness and the number of workers. The shaded regions are 1 standard deviation around the means (curves) based on 5 randomized runs. 37

3.1 Logical clock is distinct from global time. 40

3.2 An illustration of Bulk Synchronous Parallel (BSP) execution. Worker 2 is blocked at the end of clock 2 as other workers have not completed and communicated updates for clock 2. 41

3.3 An illustration of Stale Synchronous Parallel (SSP) execution with a staleness bound $s = 3$. The black and green blocks denote the updates that are visible to worker 2; the green updates are visible due to read-my-write consistency. The blue updates are not necessarily visible to worker 2 under SSP. In order to satisfy SSP constraint, worker 2 is blocked at the end of clock 4 because worker 1 has not finished clock 1. 42

3.4 An illustration of Eager Stale Synchronous Parallel (ESSP) execution. The execution is similar to that of SSP (Fig. 3.3), except that updates are communicated eagerly as shown in the red blocks. 43

3.5 The ordering of the updates for analyzing SSP. 47

3.6 **Bösen Parameter Server Architecture.** The PS consists of client processes (bottom) and the server partitions (top). The Bösen client library maintains cached image of the server parameters $\tilde{A}_p$ (p indexes the worker processes). The user application instantiates compute threads, which have access to data partition D_p, and generates updates $\Delta(\tilde{A}_p, D_p)$ that are buffered by the Bösen client library. The user program's access to the parameter cache is modulated by the consistency manager to ensure bounded staleness conditions. The communication with the server is performed by background communication threads, separate from compute threads. The server processes maintain partitioned master copy of parameters A_i, where i indexes the server threads. 52
3.7 Empirical staleness distribution from matrix factorization. X-axis is (parameter age - local clock), *i.e.*, the clock differential. Y-axis is the distribution of the clock differentials observed in parameter reads. Note that in Bulk Synchronous Parallel (BSP) system such as Map-Reduce, the staleness is always -1. We use rank 100 for matrix factorization, and each clock is 1% minibatch (i.e., a minibatch corresponds to 1% of the dataset). The experiment is run on a 64 node cluster. 56
3.8 Communication and Computation breakdown for LDA for SSP and ESSP with staleness $s = 2, 4, 8$. The lower part of the bars are computation, and the upper part is communication. 56
3.9 **ML Convergence.** The convergence speed per iteration and per second for LDA and MF. The y-axes are the training objectives. In the case of LDA the training objective is log-likelihood, for which higher is better. In the case of MF the training objective is square loss (the regularization loss is negligible compared with square loss), for which lower is better. In certain cases for MF the training objective diverges (i.e., fails to converge), such as SSP $s = 5$ with 10% minibatch. In those cases the convergence curves are truncated at the clock that divergence occurs. 60

4.1 The algorithm msPG under model parallelism and stale synchronism. Machine i keeps a local model $\mathbf{x}^i(t)$ that contains stale parameters of other machines (due to communication delay and network latency). These local models are used to compute the partial gradient $\nabla_i f(\mathbf{x}^i)$ which is then used to update the parameters $x_i(t)$ in each machine. See Section 4.5 for an economical implementation of msPG. 66
4.2 Convergence over clock for group lasso under msPG using learning rate $\eta(10)$ for staleness $s = 0, 10, 20, 30$. 73
4.3 Convergence of parameter over clock for group lasso under msPG using learning rate $\eta^*(\alpha s)$ for staleness $s = 0, 10, 20, 30$. 73
4.4 Convergence over clock for group lasso under msPG using learning rate $\eta^*(\alpha s)$ for staleness $s = 0, 10, 20, 30$. The linear convergence on the logrithmic scale is consistent with the finite length property in Theorem 4.2. 74
4.5 Convergence over clock for large-scale lasso under msPG for staleness $s = 0, 1, 3, 5, 7$. 75

4.6 Convergence over wall clock time for large-scale lasso under msPG for staleness $s = 0, 1, 3, 5, 7$. 76
4.7 Empirical staleness distribution under staleness $s = 0, 1, 3, 5, 7$. X-axis is parameter age - local clock, *i.e.*, the absolute value of clock differential, similar to Fig. 3.7. Y-axis is the distribution of the clock differentials observed in parameter reads. $s = 0$ reduces to Bulk Synchronous Parallel where the empirical staleness is concentrated at clock differential -1. 76

5.1 Illustration of the AP-BCFW in the distributed (in red) and share-memory settings (in blue). The "cloud" of all worker nodes (or CPU threads) is abstracted into an oracle that keeps feeding the server (or writing to the memory bus) with updates from solving possibly approximate (and/or delayed) solutions to (5.2) on iid uniform random blocks. 79
5.2 Performance improvement with τ for (a) Structual SVM on the OCR dataset [82, 136] and (b) Group Fused Lasso on a synthetic dataset. f^* denotes primal optimum (the "primal" problem is actually referring to the dual problem in both cases). The performance metric here is the number of iterations to achieve ϵ-suboptimality. 86
5.3 **(a)** Primal suboptimality vs wall-clock time using 8 workers ($T = 8$) and various mini-batch sizes τ. **(b)** Primal suboptimality vs wall-clock time for varying T with best τ chosen for each T separately. **(c)** Speedup via parallelization with the best τ chosen among multiples of T ($T, 2T, ...$) for each T. **(d)** The same with longer subproblems. 88
5.4 Speedup with parallelization on a synthetic OCR dataset. Left plot shows the decay of primal suboptimality and the right one shows the speedup. 89
5.5 Average time per data pass in asynchronous and synchronous modes for two cases: one worker is slow with return probability p (left); workers have return probabilities (p_is) uniformly in $[\theta, 1]$ (right). Times normalized separately for AP-BCFW, SP-BCFW w.r.t. to where workers run at full speed. 90
5.6 Illustrations of the convergence BCFW with delayed updates. On the left, we have the delay sampled from a Poisson distribution. The figure on the right is for delay sampled from a Pareto distribution. We run each problem until the duality gap reaches 0.1. 91

7.1 (Fig. 3 in [36]) Convergence over logical time (work done) under synchronous training of topic models. 100
7.2 (Fig. 3 in [36]) Convergence over logical time (work done) under synchronous training (BSP, A-BSP) and non-synchronous (SSP) of topic models. 100
7.3 Illustration of gradient descent at two iterates. 103
7.4 (Fig. 3 in [109]) Explicit momentum μ to achieve the best convergence result. . . 105
7.5 106
7.6 (Fig. 3 in [31]) Test accuracies with respect to training time for sync, async training on 50, 100, and 200 nodes. 107

7.7 (Fig. 3 in [65]) The computation time vs network waiting time breakdown for topic model (Latent Dirichlet Allocation) optimized by collapsed Gibbs sampling under various staleness levels. The experiment runs on 32 VMs (each with 8 cores). 108

7.8 (Fig. 13 in [94]) The computation and network waiting time breakdown for sparse logistic regression optimized by block proximal gradient method under different maximal delays. The experiment runs on 1000 machines, each with 16 cores. 108

7.9 (Fig. 1 in [31]) The number of epochs (left) and test accuracy (right) of Inception model trained on ImageNet by varying numbers of workers (x-axis). 109

7.10 Fig. 5.2. Parallel block coordinate Frank Wolfe method applied to structural SVM on OCR dataset shows different speedup with different convergence threshold. When primal threshold is stringent (e.g., the blue curve), the algorithm does not scale beyond 50 parallel updates (x-axis). 110

7.11 Twitter feed captured on May 1, 2018. 111

7.12 (Fig. 1 of [73]) 112

7.13 (Table 1 of [1]) Validation error on ImageNet using ResNet-50. kn denotes mini-batch size, ranging from 256 (small batch) to large batch (8k). Notice that with gradual warm-up the error rate is very close to that of the small batch result. . . . 112

7.14 (Fig. 5 of [73]) Test accuracy of small batch (SB) method and large batch (LB) method that is warm started (“piggybacked”) from the SB estimates at each of the 100 SB epochs (x-axis). 113

A.1 The number of batches to reach 92% test accuracy using Deep Neural Networks with varying numbers of hidden layers using 1 worker. We consider several variants of SGD algorithms (a)-(e). Note that with depth 0 the model reduces to multi-class logistic regression (MLR), which is convex. MLR generally takes many more batches to converge because 92% test accuracy is close to the limit of the model performance, whereas deeper models can easily achieve 95-98% test accuracy. The error bars represent 1 standard deviation, computed from 5 randomized runs. 120

A.2 The number of batches to reach 92% test accuracy for Adam and SGD on 1, 8, 16 workers with varying staleness. The error bars represent 1 standard deviation based on 5 randomized runs. Depth 0 under SGD with 8 and 16 workers do not converge within the experiment horizon (77824 batches) and is thus not shown. . 121

A.3 The number of batches to reach 92% test accuracy for SGD with momentum and RMSProp on 1, 8, 16 workers with varying staleness. The error bars represent 1 standard deviation based on 5 randomized runs. Both optimizers did not reach 92% test accuracy for all runs with staleness 16 and 32 on worker 8 and 16 within the experiment horizon (77824 batches), and thus are not shown. 122

A.4 Gradient coherence for VAEs with varying depths (depth 1∼3) optimized by Adam optimization using 1 worker with no staleness ($s = 0$). The x-axis is $k = 1, ..., 32$. Here we show cosine distance up to 32 batches back. (a) is a snapshot taken from the first 512 batches, while (b) is taken from 512 batches starting from batch 25088 after algorithms have converged. The error bars around the means represent 1 standard deviation computed from 5 randomized runs. 123
A.5 The number of batches to reach test loss 130 by Variational Autoencoders (VAEs) on 1 worker, under staleness 0 to 16. We consider VAE with depth 1, 2, and 3 (the number of layers in the encoder and the decoder networks). Configurations that do not converge to the desired test loss are omitted, such as Adam optimization for VAEs with depth 3 and $s = 16$. 124
A.6 Convergence of LDA log likelihood using 50 topics with respect to the number of documents processed by Gibbs sampling, with varying staleness and the number of workers. The shaded regions are 1 standard deviation around the means (curves) based on 5 randomized runs. 125

D.1 Illustration of the signal data used in the Fused Lasso experiments. We show the original signal (left), the noisy signal given to the algorithm (middle), and the signal recovered after performing the fused lasso optimization (right). 171

List of Tables

2.1 Overview of the models, algorithms, and dataset in our study. η denotes learning rate, β_1, β_2 are parameters associated with the optimizers. α, β in LDA are the Dirichlet priors for document topic and word topic random variables, respectively. 17

3.1 **Bösen Client API.** The API is similar to key-value interfaces. The user program can read parameters via `Get` and `GetRow` (batched reads) and make (additive) updates via `Inc` and `IncRow` (batched updates). Since bounded staleness is defined with respect to logical clock, the user program needs to signal the completion of a logical unit of work via `Clock`. 50

Chapter 1

Introduction

Machine Learning (ML) is becoming a primary mechanism for extracting information from data, and the driving force behind many modern applications, such as predictive analysis, personalized content recommendation, conversational assistant, facial recognition, and autonomous vehicles. With the surging volume of data from Internet activities and sensor advancements, the available data for ML consumption in the industry have quickly grown to terabytes and more [29, 153]. Simultaneously, the computation complexity for each data sample increases drastically with the rise of high dimensional models and deep learning. For example, it is not uncommon to have models with billions of parameters [87, 107], and some models take billions of floating point operation to process each sample [130]. Despite the increasing capacity of the computing hardware, no single machine can support training ML at the scale of "Big Data" and "Big Model" in a reasonable amount of time.

A common approach to large-scale ML resorts to distributed computing, which coordinates many computers to collectively solve a resource intensive problem. Adapting ML methods to such distributed settings, however, is highly challenging. In most commodity hardware, the interconnect between the computing nodes can be up to 10^4 times slower than the CPU-memory speed within a computer, which poses significant bottlenecks. This is indeed the main difficulty in scaling ML from one machine to many. As a result of this fundamental hardware limitation, it becomes highly inefficient to require concurrent computation across different machines to observe the outcomes of each computation immediately. In other words, in order for computation to perform efficiently in the distributed environments, *staleness* during the execution of ML algorithms is often inevitable.

Under the sequential execution, there is exactly one version of model parameters at any given point of the algorithm. This model are updated by the information from the selected data and the *current* model (e.g., through gradient information). In distributed settings, however, this is no longer the case. Since communication over the network is much slower than local memory access, it is preferable to store a version of the model locally at each machine, saving the expensive round trips over the network in computing each updates. For system performance reasons, the communication model may not always require each local model to update in lock-steps. Consequently, the local models can become out of sync. This creates *stale* models. When these stale models are used to compute subsequent updates, the updates deviate from what would have been generated from the "current" model under the sequential setting, and the effects of staleness compound.

Staleness in ML algorithms is in general not well understood. Most existing theoretical results characterizing ML algorithms assume the sequential execution model, which may be the reason that for a long time distributed ML is limited to bulk synchronous parallel computation model that faithfully reproduce the sequential execution at the expense of system performance [9]. Besides gaps in theoretical understanding, the impacts of staleness on ML algorithms' behaviors are not well understood in practice either. Several reasons may have contributed to this. For one, staleness generally co-occurs with asynchrony in distributed systems, which lead to indeterministic execution that are challenging to profile. Furthermore, the performance of distributed systems is inherently coupled with the hardware environments, so good performance on one cluster does not always generalize to other environments. Oftentimes innocuous change in the system configurations such as adding more machines or sharing resources with other jobs can have profound implication on ML algorithms by altering the underlying staleness levels. Unfortunately, these complex factors can interact in unpredictable ways that are difficult to reproduce, hiding the true effects of staleness on ML algorithms.

1.1 Thesis Statement

We are interested in iterative-convergent ML algorithms, which repeatedly execute a set of updates to refine the model until certain convergence criteria are met, such as the plateauing of loss function values. This characterizes most important ML algorithms, such as gradient-based optimization, coordinate descent, other first order methods like Frank-Wolfe algorithms [50], sampling methods such as Gibbs sampling [20]. Because they often have simple subproblems structures, these iterative methods are often the most effective in solving large-scale ML problems. The iterative nature of these algorithms is indeed the crux of our study of staleness:

> **Thesis Statement:** *We can characterize staleness in machine learning algorithms and design distributed systems to perform learning efficiently at scale.*

In particular, we want to balance the following aspects in our approach to learning with staleness:

- *Empirical and Theoretical.* Staleness often appears in indeterministic environments, and its manifestation is highly coupled with the specific choices of the problems, algorithms, and data. For a complex problem as such, theories can provide the general guidance on how convergence depends on staleness, while empirical results directly verify the impact of staleness. We therefore take the complementary approach and seek to understand staleness both from the empirical angle and with theoretical characterization.
- *Algorithms and Systems.* An ultimate goal to learn with staleness is to perform learning in

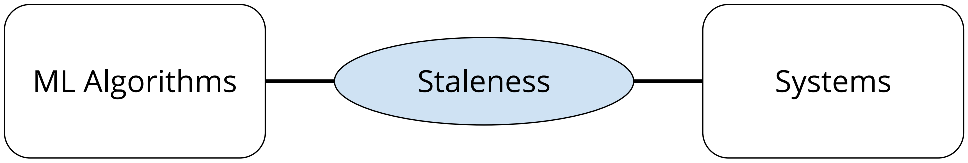


Figure 1.1: Staleness is both a system and an ML algorithm parameter, and a primary way that distributed systems and ML algorithms interact.

settings where delay is unavoidable, including in the distributed settings. Toward this goal, we want to both study the algorithmic behaviors as well as designing systems to battle test our results in realistic environments. The insights gleaned from the algorithmic study also enable us to improve existing consistency models to be more effective for stale learning.

- *Generic and Special Algorithms.* ML algorithms exhibit an astounding diversity, and there is a trade-off: more generic results can cover a broader categories of ML algorithms, while algorithms and theories developed for a special class of problems can offer sharper characterizations. We aim to balance these approaches in our study and provide a mix of generic and specific results.

Perhaps not so obvious is the fact that staleness is both a system and an ML algorithm parameter Fig. 1.1. Within the distributed systems, staleness offers opportunities to perform system optimizations such as hiding latency by overlapping communication and computation. From the ML algorithms stand point, staleness governs the ML convergence and formally enters into the convergence rates in our theoretical characterization. Furthermore, staleness is a primary way that ML algorithms interact with the distributed systems: To implement efficient distributed ML systems, one generally needs to introduce staleness to ML algorithms. Conversely, for ML algorithms to converge correctly and efficiently, the level of staleness normally has to be highly controlled in the underlying distributed systems. It is a key design knob in achieving a sweet spot between these trade-offs, and as we will see in the sequel, this can vary quite a bit across ML models and algorithms.

1.2 Backgrounds

1.2.1 Iterative-Convergent ML

Although ML programs come in many forms, almost all seek a set of parameters (typically a vector or matrix) to a global model A that best summarizes or explains the input data D as measured by an explicit objective function such as likelihood or reconstruction loss [20]. Such problems are usually solved by *iterative-convergent* algorithms, many of which can be abstracted as the following form:

$$A^{(t)} = F(A^{(t-1)}, \Delta(A^{(t-1)}, D)), \tag{1.1}$$

where, $A^{(t)}$ is the state of the model parameters at iteration t, and D is all input data. The update function $\Delta(\cdot)$ computes the model updates from data, which are applied through $F(\cdot)$ to form the model state of the next iteration. This operation repeats itself until A stops changing, i.e., converges — known as the fixed-point (or attraction) property in optimization.

As examples, for stochastic gradient (SGD) algorithms, $\Delta(\cdot)$ computes the gradient, and $F(\cdot)$ applies the additive update $A^{(t)} = A^{(t-1)} + \eta\Delta(A^{(t-1)}, D_b)$ where η is the step size and D_b is a subset (mini-batch) of D. Eq. (1.1) can also apply to sampling methods, such as collapsed Gibbs sampling for topic model (LDA) as well. In this case $\Delta(\cdot)$ increments and decrements the sufficient statistics (the "word-topic" counts) according to the sampled topic assignments (see Section 2.2.1). The update equation can also express coordinate descent algorithms where Δ returns updates on a single coordinate or a block of coordinates. $F(\cdot)$ simply overwrite the existing parameters with the new update: $A^{(t)} = \Delta(A^{(t-1)}, D)$.

1.2.2 Data Parallelism and Parameter Server

ML programs often assume that the data D are *independent and identically distributed (i.i.d)*; that is to say, the contribution of each datum D_i to the estimate of model parameter A is independent of other data D_j[1]. This in turn implies the validity of a *data-parallel* scheme *within each iteration* of the iterative convergent program that computes $A^{(t)}$, where data D is split over different threads and worker machines, in order to compute the update $\Delta(A^{(t-1)}, D)$ based on the globally-shared model state from the previous iteration, $A^{(t-1)}$.

In data-parallel ML the data set D is pre-partitioned or naturally stored on worker machines, indexed by $p = 1, ..., P$ (Fig. 1.2a). Let D_p be the p-th data partition, $A^{(t)}$ be the model parameters at clock t, the data-parallel computation executes the following update equation until some convergence criteria is met:

$$A^{(t)} = F(A^{(t-1)}, \sum_{p=1}^{P} \Delta(A^{(t-1)}, D_p))$$

where $\Delta(\cdot)$ now performs computation using full model state $A^{(t-1)}$ on data partition D_p. The sum of intermediate results from $\Delta(\cdot)$ and current model state $A^{(t-1)}$ are aggregated by $F(\cdot)$ to generate the next model state.

To achieve data-parallel ML computation, we need to (1) make model state A available to all workers, and (2) accumulate the Δ updates from workers. A Parameter Server (PS) serves these needs by providing a distributed shared memory interface (a shared key-value store using a centralized storage model). The model parameters A are stored on the server, which can be distributed and thus not limited by a single machine's memory (Fig. 1.2b). The worker machines access the entire model state on servers via a key-value interface. This distributed shared memory provided by the PS can be easily retrofitted to existing single-machine multi-threaded ML programs, by simply replacing reads/updates to the model parameters with read/update calls

---

[1] More precisely, each D is *conditionally independent* of other D_j given knowledge of the true A.

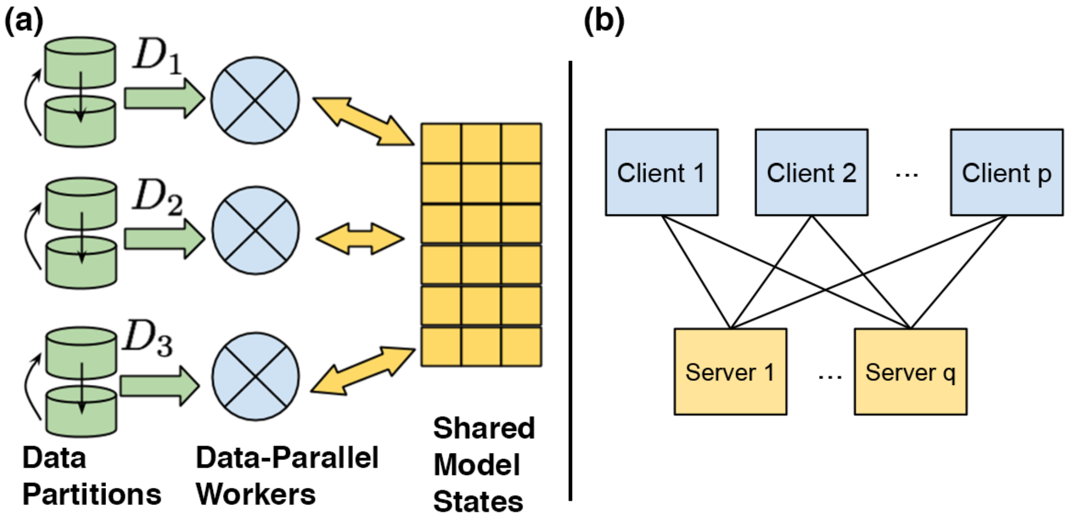


Figure 1.2: (a) Illustration of data parallelism. (b) Parameter server topology. Servers and clients interact via a bipartite topology. Note that this is the logical topology; physically the servers can collocate with the clients to utilize CPU on all machines.

to the PS. Because inter-machine communication over networks is several orders of magnitude slower than CPU-memory communication in both bandwidth and latency, one challenge in implementing an efficient PS is to design consistency models for efficient reads/updates of model parameters, a central theme in this thesis. In the sequel we will harness the error tolerance of ML algorithms in our design and implementation of consistency models.

1.2.3 Staleness Trade-offs

Staleness is parameter in both system and ML algorithms, and a point of "negotiation" between the two (Fig. 1.1). Fig. 1.3 illustrates the general trade-off along the degree of staleness between ML algorithms and distributed systems. From the system stand point, higher staleness lowers synchronization requirement, as the ML parameters do not need to be communicated as quickly. Furthermore, higher staleness also offers more opportunity to perform communication optimization, since there is a longer horizon offered by staleness for the distributed systems to reconcile different model versions on each worker.

On the other hand, from the ML algorithms' stand point, higher staleness can slow down the progress made in each iteration, as the updates are computed using more inaccurate models that do not carry the information from the latest updates. In fact, it is not a priori known whether algorithms can converge correctly under staleness, which is a subject of the thesis.

It is also important to keep in mind that Fig. 1.3 is for illustration only. The actual ramification of staleness in general is highly problem-specific. For example, we will see in Chapter 2 that ML algorithms' response to staleness is often quite non-linear and possibly not continuous. Similarly, while the system throughput is generally easy to measure, systematically mapping out the system throughput along staleness is also impractical, as the communication loads and patterns are highly dependent on the ML algorithm in question. As a result, it is generally not practical to obtain the actual convergence per second curve, and the sweet spot in the illustration remains an ideal to be empirically discovered in a somewhat ad hoc fashion.

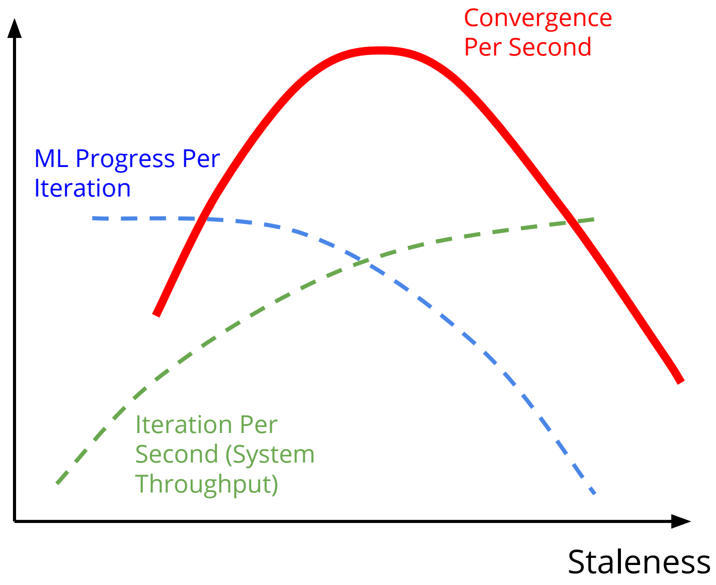


Figure 1.3: Illustration of the trade-off due to staleness (x-axis). The system throughput (iterations per second) generally improves with higher staleness (green curve), as less synchronization is needed. On the other hand, using more stale version of the ML model will result in lower quality updates computed in each iteration of the ML algorithms (blue curve). The goal is to maximize the convergence per second (red curve) within these complex trade-offs.

1.3 Contributions and Outline

We outline the contributions in this thesis below:

- **Effects of Staleness on Machine Learning (Chapter 2).** Staleness affects different models differently. Some existing works have been able to take great advantage of staleness, while others suggest that asynchrony severely impact the convergence and is outperformed by synchronous updates. Even with the same model there can be different conclusions about the effect of staleness. In this chapter, we perform simulation studies of the impact of staleness across 5 different models and 10 algorithms, including optimization, sampling, and variational inference that is the hybrid of the two. We find that while some algorithms are more robust to staleness, no ML method is immune to the negative impact of staleness. In other words, staleness is a key governing parameter of ML convergence. For variants of stochastic gradient descent algorithms—now a staple in large-scale ML problems, our results show that even some are much more sensitive to staleness than others. Perhaps surprising, certain level of staleness sometimes can accelerate the convergence speed. The response to staleness can also be highly non-linear, in which staleness below a certain threshold makes virtually no impact but precipitates rapid convergence degradation above certain threshold. For gradient-based methods we further investigate the gradient coherence during the convergence path as a possible explanation for an algorithm's sensitivity to staleness.

- **Analysis of Consistency Models (Chapter 3)**. As demonstrated in Chapter 2, staleness generally has rather significant negative impact on ML algorithms' convergence. We therefore consider a number of consistency models aimed at limiting the empirical staleness. We begin with existing consistency models such as Bulk Synchronous Parallel (BSP), Total Asynchronous Parallel (TAP), Staleness Synchronous Parallel (SSP), and propose Eager Staleness Synchronous Parallel (ESSP) implementation within the SSP. We provide extensive analyses of ML convergence under these consistency models. Our theoretical characterization goes beyond the existing correctness guarantees, and make use of empirical staleness level, instead of the worst case staleness. Our theories are consistent with the empirical study that lower empirical staleness improves convergence. We also offer bounds on the variance. We implement ESSP on a parameter server system called Bösen, and show that ESSP substantially reduces the empirical staleness in our profiling. Using Bösen we show that the lowered staleness indeed leads to better convergence on two considered applications.

- **Model Parallel Learning with Staleness (Chapter 4).** Going beyond the effects of staleness in the data parallelism setting, in this chapter we focus on dividing problem by model coordinates under staleness. In particular, we extend the proximal gradient descent to the bounded staleness environments. This is advantageous for problems with high dimensions in which transmitting and storing the entire model can lead to significant network and memory overheads. Under the model parallel learning framework each worker only needs to handle a subset of model parameters, and transmitting updates of size linear in the number of data. In situations such as biological data where the sample size is small relative to the high dimensional (e.g., hundreds of millions gene interaction features), model parallelism offers a compelling alternative to data parallelism. Our theoretical result shows that under bounded staleness condition model parallel proximal gradient descent converges correctly to critical points. Importantly, our results do not assume convexity either on the smooth loss or the non-smooth regularizer, and thus generalizes to virtually all practically useful objective functions. We support our theoretical results with empirical evaluation of non-convex Group Lasso and a large-scale Lasso problem solved on 100-node cluster.

- **Staleness in Parallel Frank-Wolfe Algorithms (Chapter 5).** The classical Frank-Wolfe (FW) algorithm, first proposed in 1956, is a simple method to solve constrained convex optimization with differentiable objective function. At each step of the algorithm, FW considers the linearization of the objective function at the current position and moves towards a minimizer o this linear function over the (compact and convex subset of a vector space) domain. This simple method has recently witnessed a revival due to its simple subproblem structures that are effective for large-scale ML problem. We consider parallelizing the block-coordinate FW algorithm, and our analyses reveals data and problem dependent quantities that governs the convergence behavior. A notable feature of the algorithms is that they do not depend on worst-case delay, but only mildly on the expected delay. Our algorithm is effective for structural SVM and Group Fused Lasso, achieving significant speedup over competing state-of-the-art synchronous methods.

Chapter 2

Effects of Staleness on Machine Learning

Most machine learning (ML) methods are iterative-convergent. Under the sequential execution paradigm, the updates from the previous iteration are immediately available, before the start of the next iteration. However, to efficiently perform learning in the distributed settings where the inter-computer communication is slow, the immediate availability of updates across the network is no longer practically attainable. In other words, we have to perform learning under *staleness*. In this chapter, we attempt to answer some of the fundamental questions about learning with staleness: *How to quantify staleness? To what extent does staleness affect the convergence of ML algorithms? What is the interplay between model complexity and the effects of staleness?* The revealed insights will guide the system design and implementation, as well as the theoretical works in the ensuing chapters.

2.1 Asynchrony or Not?

There is a growing body of works studying the behaviors of ML models and algorithms under non-synchronous execution. These works, however, seem to point to inconsistent conclusions on whether asynchronous execution helps convergence, measured in the wall clock time to reach certain model quality. To be sure, there is a broad consensus that lower synchronization results in lower system overhead, which in turn improves system throughput [31, 32, 36, 65]. It is also clear that asynchrony can potentially lead to slower convergence per algorithmic iteration, which may require additional iterations to overcome [31, 38]. Therefore the key trade-off is between the system throughput and ML progress made per iteration, as illustrated in Fig. 1.3.

Some of these works suggest that the trade-off is a favorable one. That is, the asynchronous execution does not significantly affect the convergence and benefits from improved system throughput due to the lower synchronization overheads [36, 43, 120, 131]. At the same time, bounded asynchrony offers theoretical guarantees not available under fully asynchronous environments [65, 86, 95, 158]. These work also empirically show that the increase in system throughput indeed outweighs the impact from limited delay. In contrast, there are works suggesting the opposite, that trading synchronous execution for increased throughput is disadvantageous, as asynchrony severely affects the convergence, leading to degraded model convergence or lower test performance [31, 38]. Their empirical results show that it is more beneficial to avoid asynchrony altogether and instead apply synchronous updates only.

In practice, while it is easy to measure the system throughput, such as the number iterations per second, it is often difficult to know exactly how much staleness is introduced. Furtermore, the trade-off between system speed and impact on convergence is a complex one, with a strong dependency on the specific problem. It is possible that certain models are more robust to staleness. For example, [65, 131] shows that Latent Dirichlet Allocation (LDA), an approach to topic modeling, via collapsed Gibbs sampling is insensitive to staleness. This is consistent with the findings in [40, 131, 145, 153]. Similarly, Matrix Factorization (MF) solved by stochastic gradient descent is also observed to be insensitive to staleness, as shown in [40, 65, 145]. When solved with alternating direction method of multipliers (ADMM), MF exhibit similar robustness against staleness [158]. Neither LDA or MF are convex, so convexity is not a prerequisite for introducing staleness.

While non-convexity does not preclude the use of staleness, for deep neural networks, the evidence is more mixed. On the positive side, [43] provides one of the first large-scale distributed asynchronous implementation of stochastic gradient descent that successfully speeds up the training of Deep Neural Networks (DNNs), achieving state of the art results at the time. From the theoretical stand point, [95] has shown that, under fairly mild assumptions, asynchronous stochastic gradient descent can achieves speedup on non-convex functions. These early empirical evidences and theoretical results, however, are not fully supported by the later empirical observations. [31] demonstrates that asynchronous training introduces errors in convergence, which require additional iterations to achieve similar level of accuracy, if achievable at all. The speedup from higher throughput often does not justify the additional iterations. This reduced training quality is also observed in [38].

On the whole, these studies reveal isolated instances of convergence behaviors under various staleness levels. They also raise several questions. For one, is it the case that certain class of models are amenable to staleness, while others, are not? For example, are LDA and MF robust to staleness, while others like Deep Neural Networks, are not? These works also reveal that there is a potential gap between theory and practice. Existing theories often require convexity [40, 65, 86], or certain conditions in the early iterations [95], which are generally not followed in practice, and yet it is still possible to achieve convergence with delayed updates. Theoretical guarantees [95] also may not translate to empirical success [31]. Moreover, there has not been any systematic study of the effects of staleness on convergence behaviors across a range of models and algorithms. It is to these questions that we begin our investigation.

2.2 Scope and Methods

Through simulation, we aim to provide a broader understanding of the empirical convergence behaviors of several models and algorithms under staleness.

2.2.1 Models and Algorithms

We want to study a diverse set of models that spans the spectrum from "simple" to "complex". Furthermore, we focus on algorithms that lend itself to data parallelism, which are the primary approaches employed in distributed implementation of ML models.[1] In particular we consider, optimization, sampling, and variational inference that is a hybrid of optimization and sampling method. Table 2.1 presents the summary of studied models and algorithms. In particular, within the SGD family, recent works have introduced several variants of SGD that uses gradient history to scale learning rate or adjust the gradient direction [44, 64, 75, 106]. We explore them in the context of Multi-class Logistic Regression (MLR), Deep Neural Networks (DNNs), and Variational Autoencoders (VAEs).

We consider the following models and algorithms. The pseudo-codes assume the Parameter Server (PS) application programming interface (API) detailed in Table 3.1.

Multi-class Logistic Regression (MLR)

Logistic Regression is a classical method used in large-scale classification [150], natural language processing [55, 99], and ad click-through-rate prediction [107] among others. Multiclass Logistic Regression generalizes LR to multi-way classification, such as the one used as the final layer in ImageNet 1000-way classification models [78, 130]. For each observation, MLR produces a categorical distribution over the label classes. The model stored on the PS has the size $J \times d$, where d is the input dimension and J is the number of output labels. We can solve MLR using stochastic gradient descent (SGD). The pseudo-code for MLR implemented against a distributed shared memory framework (such as parameter server or our simulation) is presented in Algorithm 1.

Multi-class Logistic Regression (MLR) represents a convex problem with excellent convergence properties under stochastic gradient descent (SGD) [26]. With (strong) convexity, MLR satisfies most existing theories of SGD convergence under asynchronous execution [65, 86, 95].

Deep Neural Networks (DNNs)

Deep Neural Networks (DNNs) are neural networks composed of fully connected layers. DNNs offer the opportunity to increase the model complexity by adding additional layers, and we can

---

[1] For a treatment of model parallelism, see Chapter 4 and Chapter 5.

Algorithm 1 SGD and its variants on parameter server. Parameter server API is introduced in detailed in Section 3.4.1. Advanced step size scaling in variants of SGD is handled in PS.

```
Require: Objective Function $f(\cdot)$, learning rate $\eta$, model parameters $x^0$ stored in (simulated) parameter server PS.
for each minibatch $D_m$ on worker $p$ do
    $\boldsymbol{x}_p^{(t)} \leftarrow$ PS.Get('$x$')    // Read model $\boldsymbol{x}_p^{(t)}$
    $\boldsymbol{g} \leftarrow \nabla_{D_m} f(\boldsymbol{x}_p^{(t)})$    // Compute gradient on the minibatch
    // For SGD, increment by $-\eta\boldsymbol{g}$; otherwise let PS handle step size
    PS.Inc(`$x$', $-\boldsymbol{g}$)
    PS.Clock()
end for
```

use this tuning knob to study the interactions between model complexity and staleness. DNNs are also a type of latent space model, where neurons in the hidden layers are in principle unidentifiable, which can pose challenge to asynchronous execution. Our DNNs have 1 to 6 hidden layers, with 256 neurons in each layer. We use rectified linear units (ReLU) for nonlinearity after each hidden layer [111]. The pseudo-code, similar to MLR, is presented in Algorithm 1.

Latent Dirichlet Allocation (LDA)

Topic Model, or more specifically, Latent Dirichlet Allocation (LDA), is an unsupervised method to uncover hidden semantics ("topics") from a group of documents, each represented as a multiset of tokens (bag-of-words). In LDA each token w_{ij} (j-th token in the i-th document) is assigned with a latent topic z_{ij} from totally K topics. We use Gibbs sampling to infer the topic assignments z_{ij}[2]. With Dirichlet priors α, β, LDA with K topics assumes the following generative model [58]:

$$\begin{aligned}
\boldsymbol{\phi}_k &\sim \text{Dirichlet}(\boldsymbol{\beta}) && \text{Sample topic word distribution for each topic } k = \{1, ..., K\} \\
\boldsymbol{\theta}_i &\sim \text{Dirichlet}(\boldsymbol{\alpha}) && \text{Sample document topic distribution for each document } i \\
z_{ij}|\boldsymbol{\theta}_i &\sim \text{Discrete}(\boldsymbol{\theta}_i) && \text{Sample topic } z_i \in \{1, ..., K\} \text{ for each token } j \text{ in document } i \\
w_{ij}|z_{ij}, \boldsymbol{\phi}_{z_{ij}} &\sim \text{Discrete}(\boldsymbol{\phi}_{z_{ij}}) && \text{Sample word from topic word distribution } \boldsymbol{\phi}_{z_{ij}}
\end{aligned}$$

We measure the model quality using log likelihood, defined as

$$\log p(\boldsymbol{w}, \boldsymbol{z}) = \log p(\boldsymbol{w}|\boldsymbol{z}) + \log p(\boldsymbol{z}) \tag{2.1}$$

Let W be the number of vocabularies, n_k^w be the number of word w assigned topic k, $n_k^{(\cdot)}$ be the number of tokens assigned topic k, n_k^i be the number of tokens assigned to topic k in document

---

[2] For the distributed implementation in Chapter 3, we use the SparseLDA variant in [147] which is also used in YahooLDA [6] that we compare with in the sequel.

i, and $n^i_{(\cdot)}$ be the number of tokens in document i, we can write

$$\log p(\boldsymbol{w}|\boldsymbol{z}) = K\big[\log\Gamma(W\beta) - W\log\Gamma(\beta)\big] + \sum_{k=1}^{K}\big[\sum_{w=1}^{W}\log\Gamma(n^w_k+\beta) - \log\Gamma(n^{(\cdot)}_k + W\beta)\big]$$
$$\log p(\boldsymbol{z}) = D\big[\log\Gamma(K\alpha) - K\log\Gamma(\alpha)\big] + \sum_{i=1}^{D}\big[\sum_{k=1}^{K}\log\Gamma(n^i_k+\alpha) - \log\Gamma(n^i_{(\cdot)} + K\alpha)\big]$$

, where Γ is the usual gamma function.

The Gibbs sampling step involves three sets of parameters, known as *sufficient statistics*: (1) document-topic vector $\theta_i \in \mathbb{R}^K$ where θ_{ik} the number of topic assignments within document i to topic $k = 1...K$; (2) word-topic vector $\phi_w \in \mathbb{R}^K$ where ϕ_{wk} is the number of topic assignments to topic $k = 1, ..., K$ for word (vocabulary) w across all documents; (3) $\tilde{\phi} \in \mathbb{R}^K$ where $\tilde{\phi}_k = \sum_{w=1}^{W}\phi_{wk}$ is the number of tokens in the corpus assigned to topic k. The corpus (w_{ij}, z_{ij}) is partitioned to worker nodes (i.e each node has a set of documents), and θ_i is computed on-the-fly before sampling tokens in document i. ϕ_w and $\tilde{\phi}$ are stored as rows in PS. The pseudo-code is presented in Algorithm 2.

Algorithm 2 Distributed LDA via Gibbs Sampling on parameter server

```
Require: $w_{ij}, z_{ij}$ partitioned to workers; $\phi_w$ stored in PS; number of topics $K$ and dirichlet priors: $\alpha, \beta$.
for iteration $t = 1 \to T$ do
    for each document $i$ and token $j$ in data partition $p$ do
        $\phi_{w_{ij}} \leftarrow$ PS.Get('$\phi_{w_{ij}}$')
        $k_1 \leftarrow z_{ij}$
        $k_2 \leftarrow$ Gibbs$(\theta_i, \phi_{w_{ij}}, \tilde{\phi}, \alpha, \beta)$
        if $k_1 \neq k_2$ then
            PS.Inc('$\phi_{w_{ij},k_1}$',-1)
            PS.Inc('$\phi_{w_{ij},k_2}$',+1)
            PS.Inc('$\tilde{\phi}_{k_1}$',-1)
            PS.Inc('$\tilde{\phi}_{k_2}$',+1)
            update $\theta_i$ based on $k_1, k_2$
        end if
    end for
    PS.Clock()
end for
```

LDA is a well studied model in the (bounded) asynchronous literature [65, 131]. It is an important model with many applications [22]. Using Gibbs sampling on LDA we will study an instance of sampling method under staleness.

Matrix Factorization (MF)

Matrix factorization (MF) is commonly used in recommender systems, such as recommending movies to users on Netflix. Given a matrix $D \in \mathbb{R}^{M \times N}$ which is partially filled with observed ratings from M users on N movies, MF factorizes D into two factor matrices L and R such that their product approximates the ratings: $D \approx LR^T$, where matrix $L \in \mathbb{R}^{M \times r}$ and $R \in \mathbb{R}^{N \times r}$, and $r \ll \min(M, N)$ is the user-specified rank which determines the model size (along with M and N). The ℓ_2-penalized optimization problem is:

$$\min_{L,R} \sum_{(i,j) \in D_{obs}} ||D_{ij} - \sum_{k=1}^{K} L_{ik} R_{kj}||^2 + \lambda(||L||_F^2 + ||R||_F^2)$$

where $|| \cdot ||_F$ is the Frobenius norm and λ is the regularization parameter. The stochastic gradient updates for each observed entry $D_{ij} \in D_{obs}$ are

$$\begin{aligned} L_{i*} &\leftarrow L_{i*} + 2\eta(e_{ij} R_{*j}^{\top} - \frac{\lambda}{n_i} L_{i*}) \\ R_{*j} &\leftarrow R_{*j} + 2\eta(e_{ij} L_{i*}^{\top} - \frac{\lambda}{m_j} R_{*j}) \end{aligned} \tag{2.2}$$

where L_{i*}, R_{*j} are row and column of L, R respectively. $e_{ij} := D_{ij} - L_{i*} R_{*j}$, and $L_{i*} R_{*j}$ is the vector product. $n_i := \sum_{j=1}^{M} \mathbb{I}(D_{ij} \neq 0)$ is the number of non-zero elements in row i of the data matrix D, and m_j to denote the number of non-zeros in column j of D. Note that here we absorb constants such as $|D_{obs}|$ into the learning rate η. The parameter settings are presented in Table 2.1.

We partition observations D to P workers and solve MF via stochastic gradient descent (SGD). The pseudo-code for MF implemented on our PS is presented in Algorithm 3.

MF is another commonly studied model in the distributed ML literature [65, 74, 79]. While it is non-convex, it is bi-convex which can have a simpler problem structure than general non-convex problems. MF can also be considered as an embedding model in which the parameters of the model are the embeddings for each users and items, and thus each update will only be sparsely applied to the relevant embedding vectors.

Variational Auto-Encoder (VAE)

Variational Autoencoder (VAE) is an unsupervised model that leverages deep neural networks to construct (non-linear) encoding and decoding functions, in which the encoder function embed objects $\boldsymbol{x}$ into a latent code $\boldsymbol{z}$ [76]. Fig. 2.1 gives the “mathematical architecture” or VAE. In particular, the inputs to the VAE training include both the data feature $\boldsymbol{x}$ and a random sample ϵ, which has implication on the gradient behaviors. It represents an interesting hybrid of optimization and sampling method, and the optimization is performed over data input and random samples of variables. We will explore this further in the sequel.

Algorithm 3 Matrix Factorization via SGD

```
Require: Learning rate schedule $\eta_t$, factor matricies $L$ and $R$ stored in PS. Let $n_j, m_j$ denotes the number of non-zero entries in row $j$ of $D$, respectively.
for iteration $t = 1 \to T$ do
    for each observed entry $D_{ij}$ in data partition $p$ do
        $\boldsymbol{L}_i \leftarrow$ PS.Get('$L_i$')    // $\boldsymbol{L}_i$ is $i$-th row of $\boldsymbol{L}$
        $\boldsymbol{R}_j \leftarrow$ PS.Get('$R_j$')    // $\boldsymbol{R}_j$ is $j$-th row of $\boldsymbol{R}$
        $e_{ij} = -2(D_{ij} - \boldsymbol{L}_i \cdot \boldsymbol{R}_j^T)$
        $\boldsymbol{L}_{grad} = (e_{ij}\boldsymbol{R}_j + \frac{2\lambda}{n_i}\boldsymbol{L}_i)$
        $\boldsymbol{R}_{grad} = (e_{ij}\boldsymbol{L}_i^T + \frac{2\lambda}{m_j}\boldsymbol{R}_j)$
        PS.Inc('$L_i$', $-\eta_t\boldsymbol{L}_{grad}$)
        PS.Inc('$R_j$', $-\eta_t\boldsymbol{R}_{grad}$)
    end for
    PS.Clock()
end for
```

Algorithm 4 SGD and its variants on parameter server. Advanced step size scaling in variants of SGD is handled in PS.

```
Require: $\phi, \theta$ stored in parameter server (PS), learning rate $\eta$ and other optimization parameters.

for each minibatch $D_m$ on worker $p$ do
    $\boldsymbol{\theta}_p^{(t)}, \boldsymbol{\phi}_p^{(t)} \leftarrow$ PS.Get('$x$')    // Read model parameters
    $\boldsymbol{g} \leftarrow \nabla_{\theta,\phi}\mathcal{L}(\boldsymbol{\phi}_p^{(t)}, \boldsymbol{\theta}_p^{(t)} | D_m, \epsilon)$    // Compute gradient on the minibatch
    // For SGD, increment by $-\eta\boldsymbol{g}$; otherwise let PS handle step size
    PS.Inc('$x$', $-\boldsymbol{g}$)
    PS.Clock()
end for
```

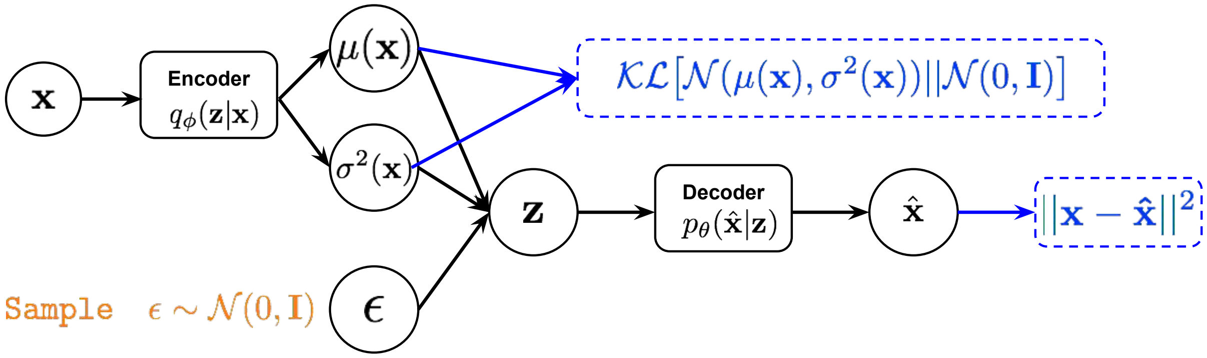

$\mathbf{x}$
Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
$\mu(\mathbf{x})$
$\sigma^2(\mathbf{x})$
$\mathcal{KL}[\mathcal{N}(\mu(\mathbf{x}),\sigma^2(\mathbf{x}))||\mathcal{N}(0,\mathbf{I})]$
$\mathbf{z}$
Decoder
$p_\theta(\hat{\mathbf{x}}|\mathbf{z})$
$\hat{\mathbf{x}}$
$||\mathbf{x}-\hat{\mathbf{x}}||^2$
Sample $\epsilon \sim \mathcal{N}(0,\mathbf{I})$
ϵ

Figure 2.1: Variational Autoencoder (VAE) at training time, assuming continuous input $\boldsymbol{x}$ and isotropic Gaussian prior $p(\boldsymbol{z}) \sim \mathcal{N}(0,\boldsymbol{I})$. The encoder $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ encodes input $\boldsymbol{x}$ to mean $\mu(\boldsymbol{x})$ and variance $\sigma^2(\boldsymbol{x})$ such that the sample $\boldsymbol{z} = \mu(\boldsymbol{x}) + \sigma(\boldsymbol{x}) \odot \epsilon$ is sampled according to distributed $q_\phi(\boldsymbol{z}|\boldsymbol{x})$ ($\epsilon \sim \mathcal{N}(0,\boldsymbol{I})$). Thanks to the reparametric trick [76], the loss functions are differentiable with respect to θ, ϕ and can be back-propagated throughout to compute the gradient. Minimization objectives are denoted in dashed blue boxes. In our experiments we use DNNs as the encoder and the decoder.

Both the VAE encoder and decoders are DNNs with 1∼3 layers, each with 256 units furnished with rectified linear function for non-linearity. The model quality is measured by the training objective value, defined on a mini-batch of data $D_m = \{\boldsymbol{x}_i\}_{i=1}^{|D_m|}$ as:

$$\begin{aligned}\mathcal{L}(\boldsymbol{\phi},\boldsymbol{\theta};D_m) &:= \sum_{i=1}^{|D_m|} -\mathcal{KL}\big[\mathcal{N}(\mu(\mathbf{x}^i),\sigma^2(\mathbf{x}^i))||\mathcal{N}(0,\mathbf{I})\big] + \log p_\theta(\boldsymbol{x}^i|\boldsymbol{z}^i) \\ &\approx \sum_{i=1}^{|D_m|}\sum_{d=1}^{D}\frac{1}{2}\big[1+\log((\sigma_d^i)^2) - (\mu_d^i)^2 - (\sigma_d^i)^2\big] + ||\boldsymbol{x}^i - \hat{\boldsymbol{x}}^i||^2 + \text{const}\end{aligned}$$

where we sum over D dimensions to obtain $\mathcal{KL}$ divergence, and $\hat{\boldsymbol{x}}$ is the reconstructed sample. Note that $\approx$ is based on the unbiased estimates using a single sample z^i, which is what we use in the experiments.

2.2.2 Datasets

We use the following datasets in our study:

- **MNIST** [90] is a popular benchmark for deep learning models. It is an image dataset of hand-written digits. The version we use has 50,000 training samples and 10,000 test samples. The image resolution is 28 by 28. In our experiments we convert each image into a 784-dimensional feature.

- **20 NewsGroup** [123] is a text corpus consisting of news articles from 20 news categories. There are in total 11269 documents, 1.3 million tokens, and 61188 unique vocabularies in the corpus. Since LDA is an unsupervised method, we ignore the text category labels in the dataset and consider only the words.
- **MovieLens 1M** [61] is a movie rating dataset collected from the MovieLens website.[3] We consider the subset of 1 million ratings made by 6040 users on 3952 movies. The ratings are made on a 5-star scale (integer 1-5), and each user has at least 20 ratings. In our MF experiments we scale the ratings to {-1,-0.5,0,0.5,1} instead.

Model	Algorithms	Key Parameters	Dataset
DNN, MLR	SGD	$\eta = 0.01$	MNIST[90]
	SGD with Momentum[117]	$\eta = 0.01$, momentum=0.9	
	Adam[75]	$\eta = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$	
	AdaDelta[155]	$\eta = 0.01$	
	RMSProp[64]	$\eta = 0.01$, decay=0.9, momentum=0	
	FTRL[106]	$\eta = 0.01$	
	AMSGrad[122]	$\eta = 0.01, \beta_1 = 0.9, \beta_2 = 0.999$	
LDA	Gibbs Sampling	$\alpha = 0.1, \beta = 0.1$	20 NewsGroup[123]
MF	SGD	rank=5, $\lambda = 0.0001$	MovieLens1M [61]
VAE	VI[4]	Optimization parameters same as MLR/DNN	MNIST[90]

Table 2.1: Overview of the models, algorithms, and dataset in our study. η denotes learning rate, β_1, β_2 are parameters associated with the optimizers. α, β in LDA are the Dirichlet priors for document topic and word topic random variables, respectively.

2.3 Experiments

2.3.1 System Configurations

All experiments are conducted on machines configured with 16-core Intel Xeon and 64GB of memory running Ubuntu 16.04. MLR, DNNs, and VAEs are implemented in TensorFlow [2] version 1.5. MF and LDA are implemented in Python.

---

[3] http://movielens.org

[4] Variational Inference

2.3.2 Simulation Model

We simulate the multi-worker settings on single machine. For each simulated worker we maintain a model parameter version similar to the local cache in distributed implementations [65, 145]. For each worker p its local model always receives the update from itself at the end of each iteration. For updates sent to other workers, we apply a uniformly random delay. Specifically, let u_p^t be the update generated at iteration t by worker p. For each worker $p' \neq p$, our delay model applies a delay $r_{p,p'}^t \sim Categorical(1, 2, .., S)$, where S is the maximum delay and $Categorical()$ is the categorical distribution placing equal weights on each integer. Under this delay model, update u_p^t shall arrive at worker p' at the start of iteration $t + r_{p,p'}^t$. The theoretical average delay under this model is $\frac{P-1}{2P}S$ where P is the number of workers.[5] Since model versions on each worker are symmetrical, we use the first worker's model to evaluate the model quality. Finally, in the simulation study we are most interested in measuring convergence against logical time, and wall clock time is in general not material as the simulation on single machine is not optimized for performance.

2.3.3 Deep Neural Networks with Staleness

We begin with DNNs optimized by SGD and its variants using a minimal architecture with one hidden layer. Moreover, we use only 1 worker so that for staleness 0 we recover the sequential setting. Fig. 2.2 shows the number of batches needed to reach 95% test accuracy for 6 SGD variants: SGD, SGD with momentum, Adam, RMSProp, Adagrad, and FTRL. We vary batch sizes (colors) and maximum staleness (clusters). The error bars represent 1 standard deviation, obtained from 5 runs with randomized initialization. We also consider two additional SGD variants: AdaDelta and AMSGrad. Fig. 2.3 shows the AMSGrad result, while AdaDelta was not able to reach 95% test error under all settings and is not shown here.[6] We now make several observations:

- Convergence speeds under sequential execution differ for different optimization algorithms. At staleness 0 ($s = 0$), we recover the sequential setting. It is clear that different SGD converges at very different speed, with Adam and RMSProp being the fastest (i.e., lowest number of of minibatches), and SGD and FTRL the slowest. Both SGD and FTRL can be sensitive to the initial learning rate, which we only tuned to the highest order of magnitude that still converges under staleness 0 (Table 2.1).
- Staleness has uneven impacts on different SGD variants. For example, Adam, SGD with Momentum, RMSProp, and AMSGrad are highly sensitive to the staleness. Staleness generally increases the number of batches needed to reach target test accuracy, and the increase can be substantial. On the other hand, SGD, Adagrad, and FTRL appear to be robust to the staleness. In the case of FTRL staleness appears to lower the number of

---

[5] We defer different delay models to future work.

[6] Due to its poor performance, we will not consider AdaDelta in subsequent experiments.

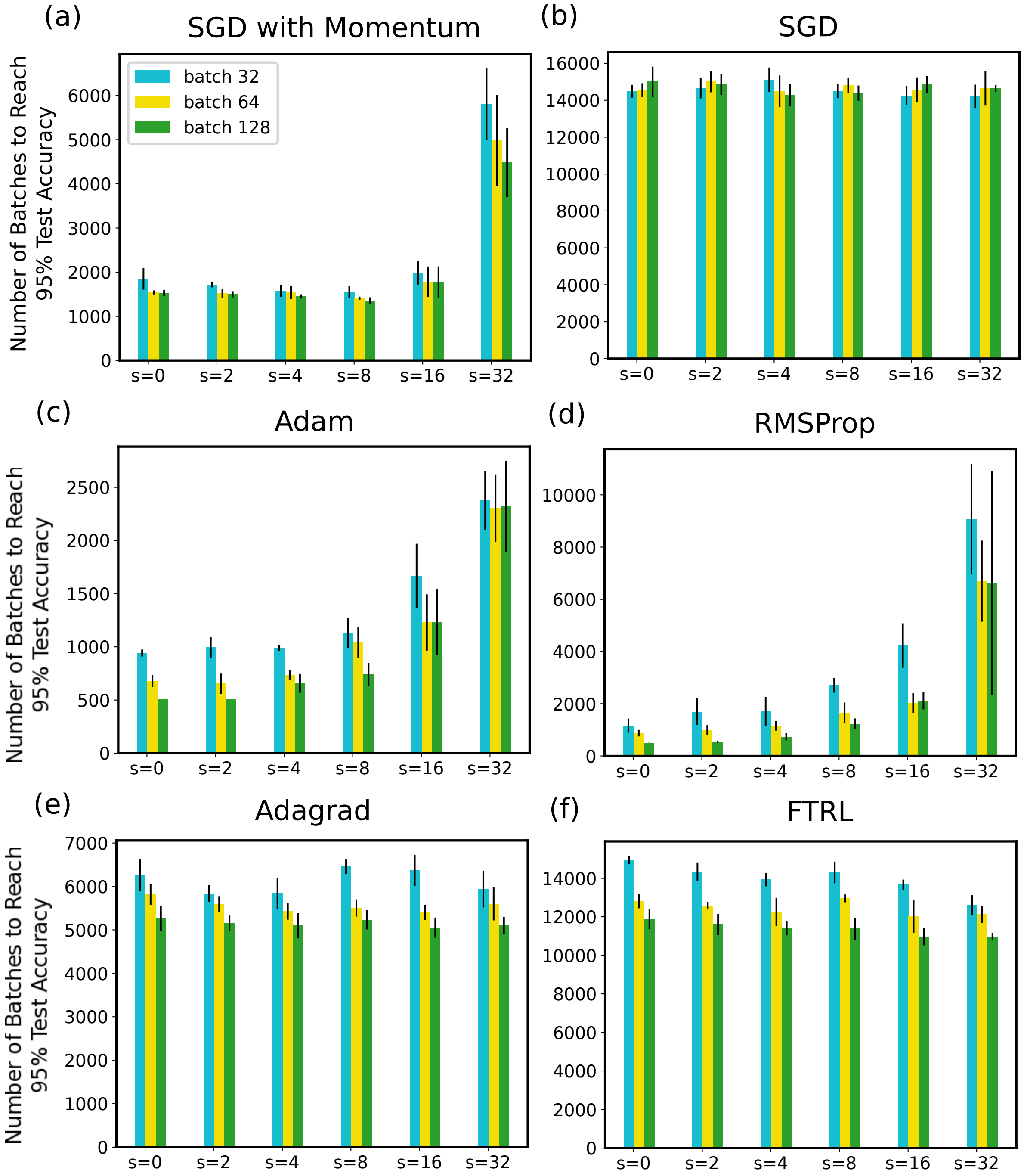


Figure 2.2: The number of batches to reach 95% test accuracy using 1 hidden layer and 1 worker. Each color represents a batch size, while each cluster corresponds to the maximum staleness in the simulation model.

batches necessary, potentially due to the implicit momentum created by staleness [109], and momentum is helpful for convergence as evident in Fig. 2.2(a).

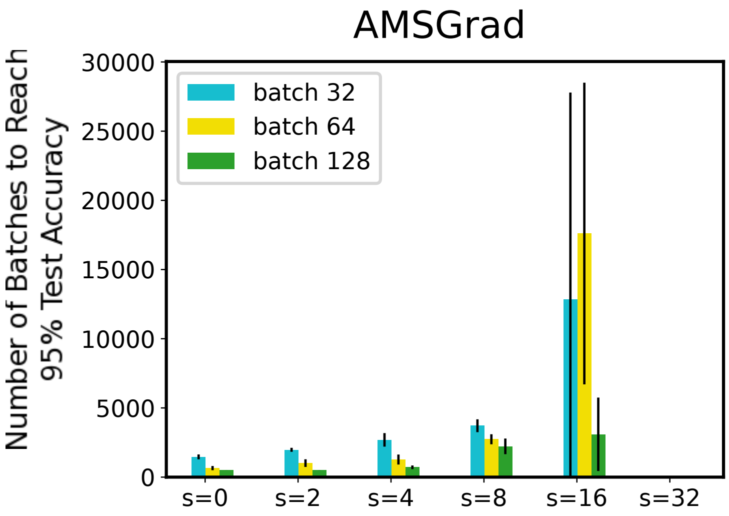


Figure 2.3: The number of batches to reach 95% test accuracy using 1 hidden layer and 1 worker, using AMSGrad [122] and varying batch sizes. $s = 32$ did not converge and thus not shown.

- Batch size has limited effect on the number of batches needed to converge. It is true that larger batch sizes reduces the number of batches to convergence, but only very mildly. For example, when increasing batch size 4 folds (from 32 to 128), the number of batches to convergence is reduced by at most $\sim$50% in the case of Adam and RMSProp. In all other cases the reduction is only a small fraction. In other words, larger batch size will require more samples to be processed to reach convergence.

These experiments also help us to answer the question: how do we measure staleness for SGD? We may consider measuring the staleness in terms of the number of data being processed, or the percentage of the dataset that is processed. However, as our experiments suggest, they are not the most direct way to quantify staleness for SGD. As discussed earlier, larger batch sizes generally require many more data samples to be processed to reach the same model quality. If we use the number of data samples being processed to quantify staleness, then $s = 32$ for batch size of 32 should have similar effect as $s = 8$ for batch size of 128. That, however, is not what we observe. Fig. 2.4 shows the number of batches to reach 95% test accuracy, normalized by $s = 0$ for each batch size. It shows how staleness degrades the performance. We can see that, for most SGD variants, the response to staleness, once normalized by $s = 0$, is similar across batch sizes. This indicates that the number of batches, rather than the number of data samples processed, is a good proxy measure for staleness and progress for SGD algorithms. We therefore will consider only batch size 32, and use the number of batches as the measure for a unit of work and staleness in subsequent experiments.

2.3.4 Staleness and Model Complexity

Under the sequential setting, model complexity can often lead to optimization difficulty. For example, before residual learning, the depth of convolutional neural networks were generally

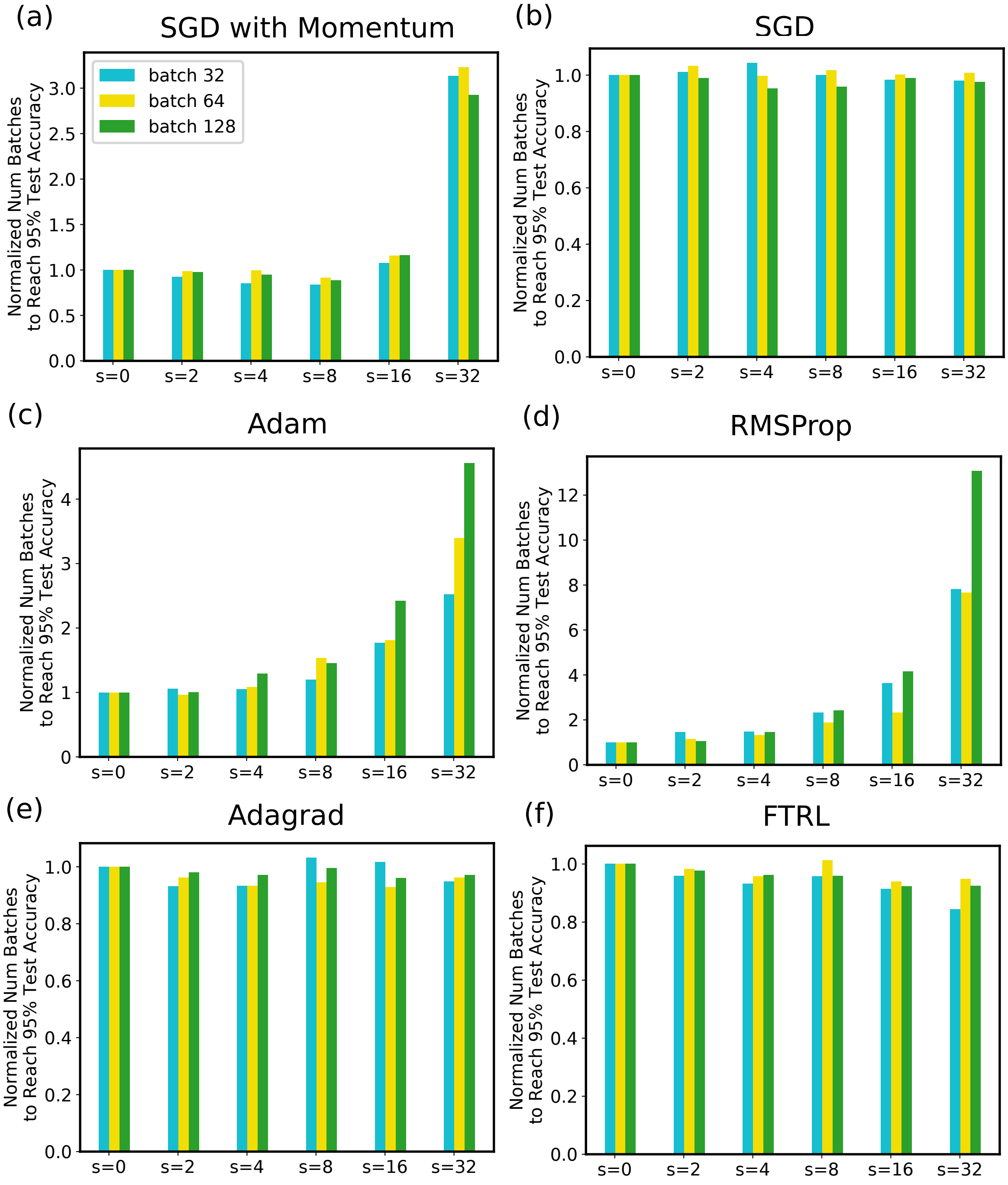


Figure 2.4: The number of batches to reach 95% test accuracy using 1 hidden layer and 1 worker, respectively normalized by $s = 0$. The panel is a normalized counterpart of Fig. 2.2

limited by optimization challenges [63]. Does this optimization difficulty compound with the effect of staleness, which we have shown to have negative impact on convergence?

DNNs are suitable test beds to study this question. By changing the number of hidden layers, we

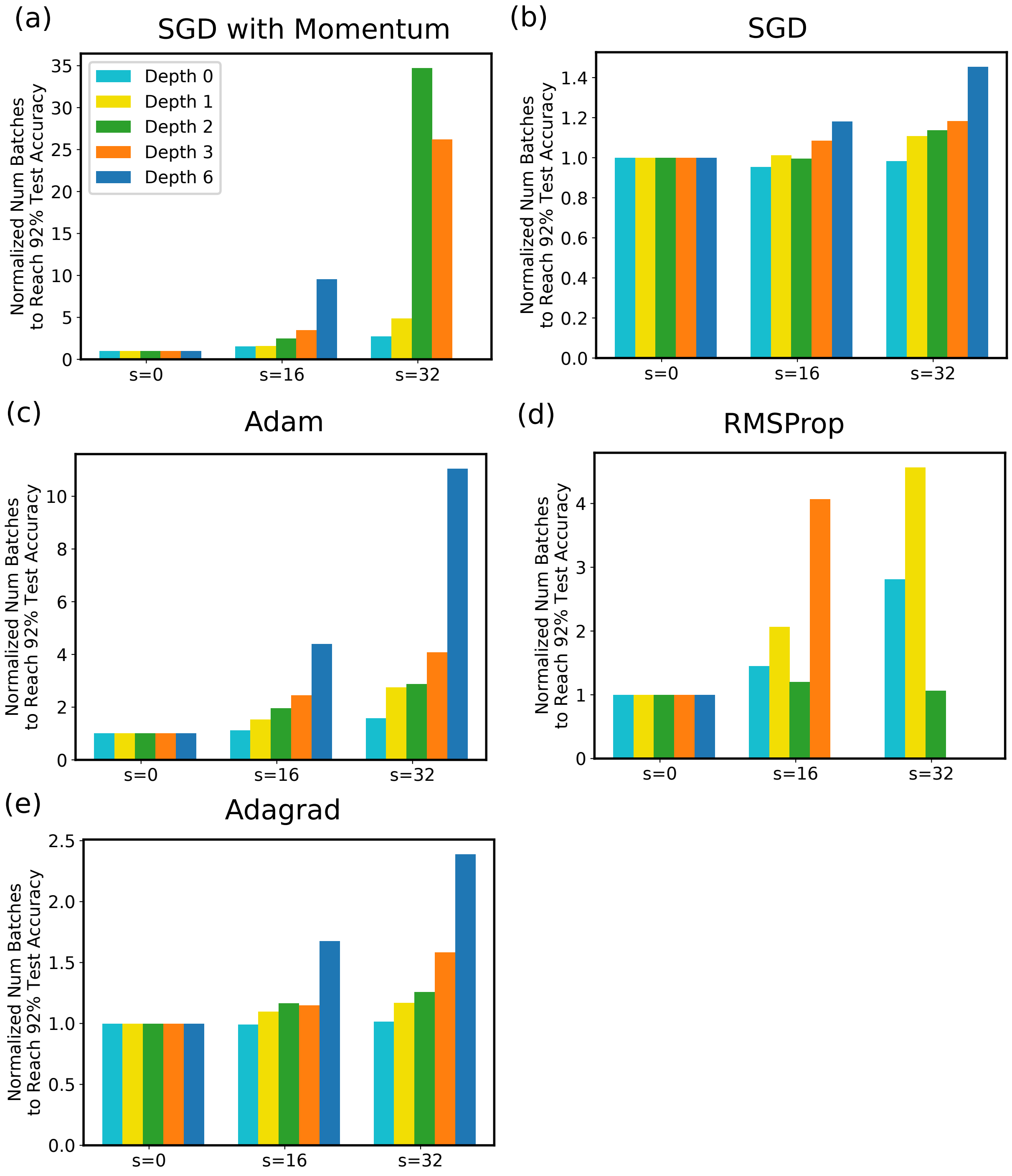


Figure 2.5: The number of batches to reach 92% test accuracy using Deep Neural Networks with varying numbers of hidden layers under 1 worker. We consider several variants of SGD algorithms (a)-(e). Note that with depth 0 the model reduces to multi-class logistic regression (MLR), which is convex. The numbers are averaged over 5 randomized runs. We omit the bars whenever convergence is not achieved within the experiment horizon (77824 batches), such as SGD with momentum at depth 6 and $s = 32$. We do not include FTRL result due to the unstable convergence. The unnormalized version is provided in the appendix (Fig. A.1).

can control the model complexity, as each hidden layer introduces an additional set of weights and non-linearity. In fact, for depth 0 (no hidden layer), we recover MLR, which is convex. We consider depth 0–6. Fig. 2.5 shows the number of batches needed to reach 92% test accuracy, under 1 worker, for varying depths and optimizers, each depth normalized by staleness 0, respectively.[7]

The convergence time for both SGD with momentum and RMSProp exhibits a high variance, which is shown in the unnormalized plot in the Appendix (Fig. A.3). Moreover, the variance in convergence speed is higher with the increasing staleness and the number of workers, which can be observed in Fig. A.2 (the unnormalized version of Fig. 2.5) in the Appendix. Focusing on SGD, Adam, and Adagrad in Fig. 2.5, we make a number of observations:

- For each depth, staleness generally increases convergence difficulty, manifested in the higher number of batches needed to reach the same test accuracy. The only exception is SGD, where staleness creates implicit momentum which actually speeds up the convergence [109].
- Higher staleness disproportionately impacts deeper networks than shallower ones. In particular, SGD, Adam, and Adagrad exhibit highly consistent patterns where the staleness-induced convergence difficulties increases with increasing depth.
- MLR (Depth 0) is much more robust to staleness than deeper networs. This is likely due to the (strong) convexity of MLR, which often leads to more coherent gradients between mini-batches. We will revisit this point in the sequel.

Fig. 2.6 shows the number of batches to convergence under Adam and SGD on 1, 8, 16 simulated workers, respectively normalized by staleness 0's values. The unnormalized version is in Appendix (Fig. A.2). SGD with momentum and RMSProp did not reach 92% test accuracy for all runs with staleness 16 and 32 on worker 8 or 16 (Fig. A.3), and thus we focus our observations on Adam and SGD:

- Compared with the single worker setting, multiple workers magnify the effect of staleness. For example, depth 3 under Adam needs 4x more batches to converge under $s = 32$ on a single worker, compared with $s = 0$. That multiple increases to 20x with 16 workers.
- With more workers, the variance in convergence also increases, which leads to the less consistent patterns in Adam optimization with 8 and 16 workers. We refer interested reader to the appendix (Fig. A.2) for the plot of the unnormalized convergence.
- MLR (depth 0) convergence difficulties caused by staleness increase under multi-worker such that reaching 92% is sometimes not possible within the experiment horizon. For example, Adam optimization on 16 workers under staleness 32 failed to achieve the target test accuracy, and similarly for SGD optimization on 8 and 16 workers across all staleness levels.

---

[7] We pick 92% test accuracy as target instead of the 95% test accuracy due to the limited model capacity of MLR, which can only achieve test accuracy about 92.7%.

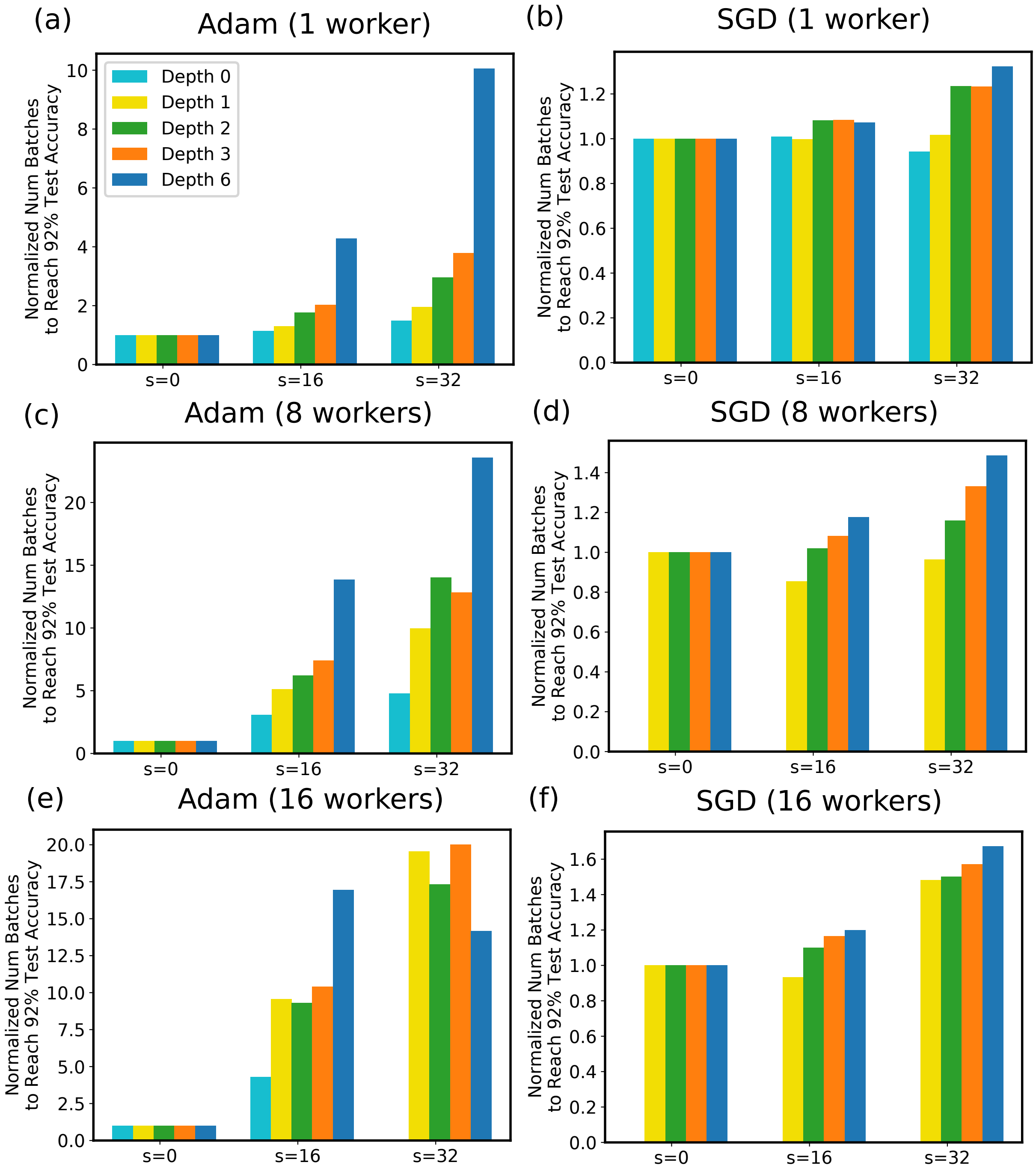


Figure 2.6: The number of batches to reach 92% test accuracy with Adam and SGD on 1, 8, 16 workers with varying staleness. Each depth is normalized by the staleness 0's values, respectively. The numbers are average over 5 randomized runs. Depth 0 under SGD with 8 and 16 workers did not converge within the experiment horizon (77824 batches) for all staleness values, and is thus not shown. The unnormalized version is in Appendix (Fig. A.2)

In light of these findings, we will focus on SGD and Adam in the subsequent experiments for several reasons: (1) They represent two very distinct responses to staleness. SGD is highly robust to staleness, while Adam is quite sensitive; (2) They have stable convergence under staleness and deeper architectures, which is not shared by other optimizers such as RMSProp and SGD with momentum. This is an important characteristics for our study as high variance results can be difficult to interpret (e.g. RMSProp); (3) The simplicity of SGD can be instrumental for our understanding, with the added benefit that SGD is well understood with a wealth of existing analyses. Adam, on the other hand, is an advanced gradient method that makes use of gradient history to modify the gradient updates. The diversity of these two methods can potentially reveal further insights; (4) SGD and Adam are two of the most commonly used optimization schemes.

2.3.5 Gradient Coherence

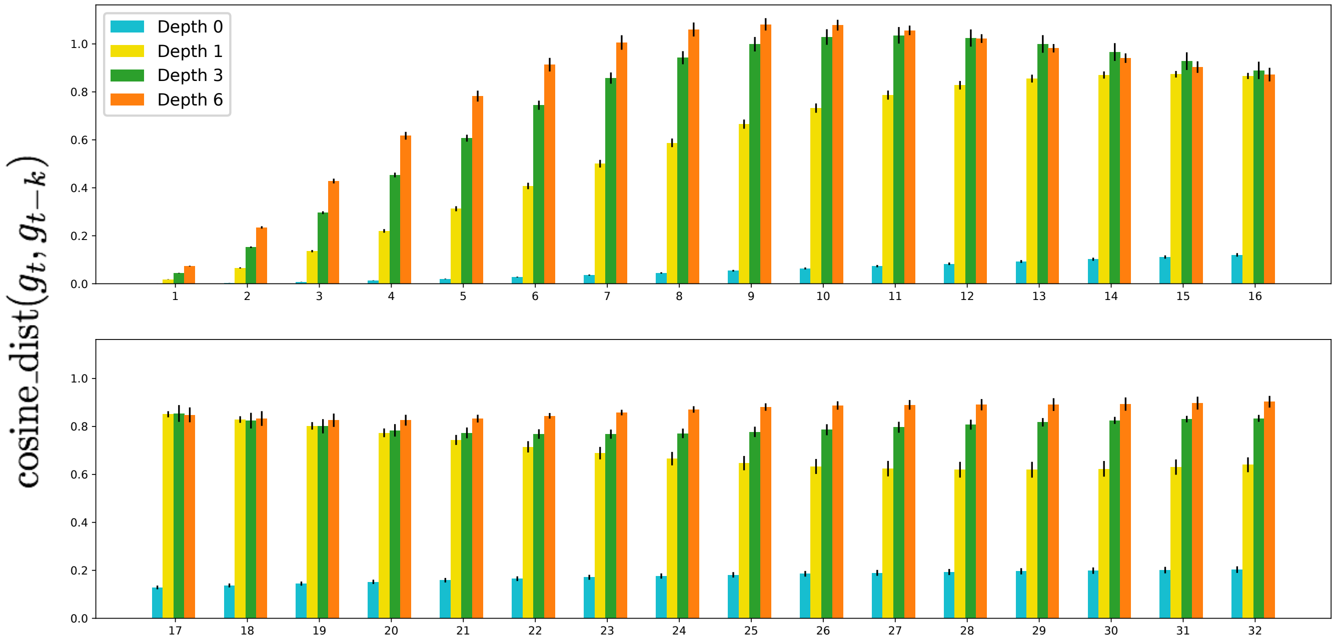
To better understand how model complexity interacts with staleness, we consider cosine distance to measure *gradient coherence* across the batches:

$$\begin{aligned}\text{cosine\_dist}(\boldsymbol{u}, \boldsymbol{v}) &:= 1 - \cos(\theta_{\boldsymbol{u},\boldsymbol{v}}) \\ &= 1 - \frac{\boldsymbol{u} \cdot \boldsymbol{v}}{||\boldsymbol{u}||_2 ||\boldsymbol{v}||_2}\end{aligned}$$

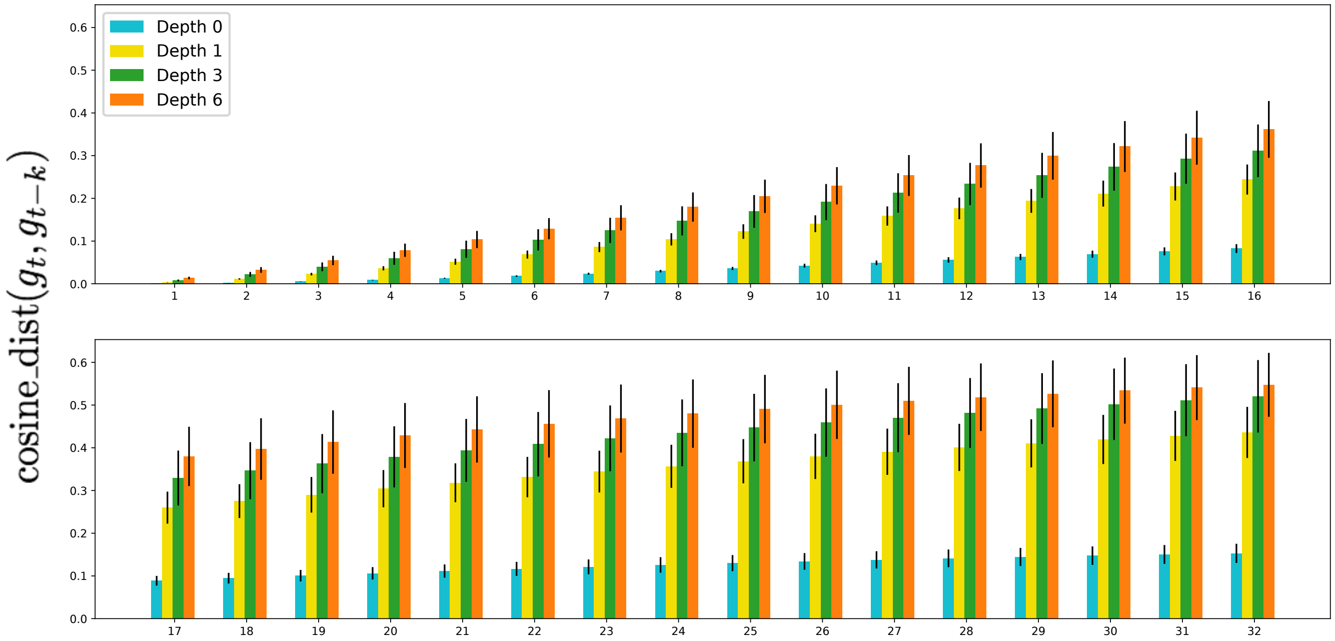
, where $\theta_{\boldsymbol{u},\boldsymbol{v}}$ is the angle between $\boldsymbol{u}, \boldsymbol{v}$. Cosine distance ranges from 0 to 2. A value greater than 1 implies $\theta_{\boldsymbol{u},\boldsymbol{v}} > \pi$. We want to measure $\text{cosine\_dist}(\boldsymbol{g}_t, \boldsymbol{g}_{t-k})$ for $k = 1, 2, ...$ which can indicate how quickly the gradient changes directions. Furthermore, this quantity is easy to measure empirically. In our experiments we compute the gradients on the first 1000 samples in the dataset in each iteration to serve as an estimate for the full gradient.

Fig. 2.7 and Fig. 2.8 show the profile of gradient coherence for Adam and SGD under sequential execution (1 worker, $s = 0$). Specifically, we compute the cosine distances between the current gradient $\boldsymbol{g}_t$ and $\boldsymbol{g}_{t-1}, ..., \boldsymbol{g}_{t-32}$ for DNNs with depth 0–6 optimized by Adam (Fig. 2.7) and SGD (Fig. 2.8). Fig. 2.7(a) and Fig. 2.8(a) are snapshots at the beginning of the algorithm while Fig. 2.7(b) and Fig. 2.8(b) are taken after the model has already converged. We make a number of observations:

- MLR (depth 0) exhibits much lower cosine distance (and hence much higher gradient coherence) across the mini-batches, compared with DNNs (depth 1, 3, 6). This is consistent with the fact that MLR is (strongly) convex, while all DNNs are non-convex. Convex functions do not have saddle points or other critical points except global optima, which generally leads to a more consistent gradient directions.
- The cosine distances in both Fig. 2.7 and Fig. 2.8 largely remain below 1, implying that the gradients are positively correlated: $\langle g_t, g_{t-k} \rangle > 0$. This may have theoretical implications, which we leave to the future work.



(a) Gradient Coherence: DNN with Adam on 1 worker (batch 0~512)



(b) Gradient Coherence: DNN with Adam on 1 worker (batch 25088~25600)

Figure 2.7: Gradient coherence for DNNs with varying depths (depth 0–6) optimized by the Adam optimization using 1 worker and no staleness ($s = 0$). The x-axis is $k = 1, ..., 32$. Here we show cosine distance up to 32 batches back. (a) is a snapshot taken from the first 512 batches, while (b) is taken from 512 batches starting from batch 25088 after the algorithm has converged. The error bars around the means represent 1 standard deviation computed from 5 randomized runs.

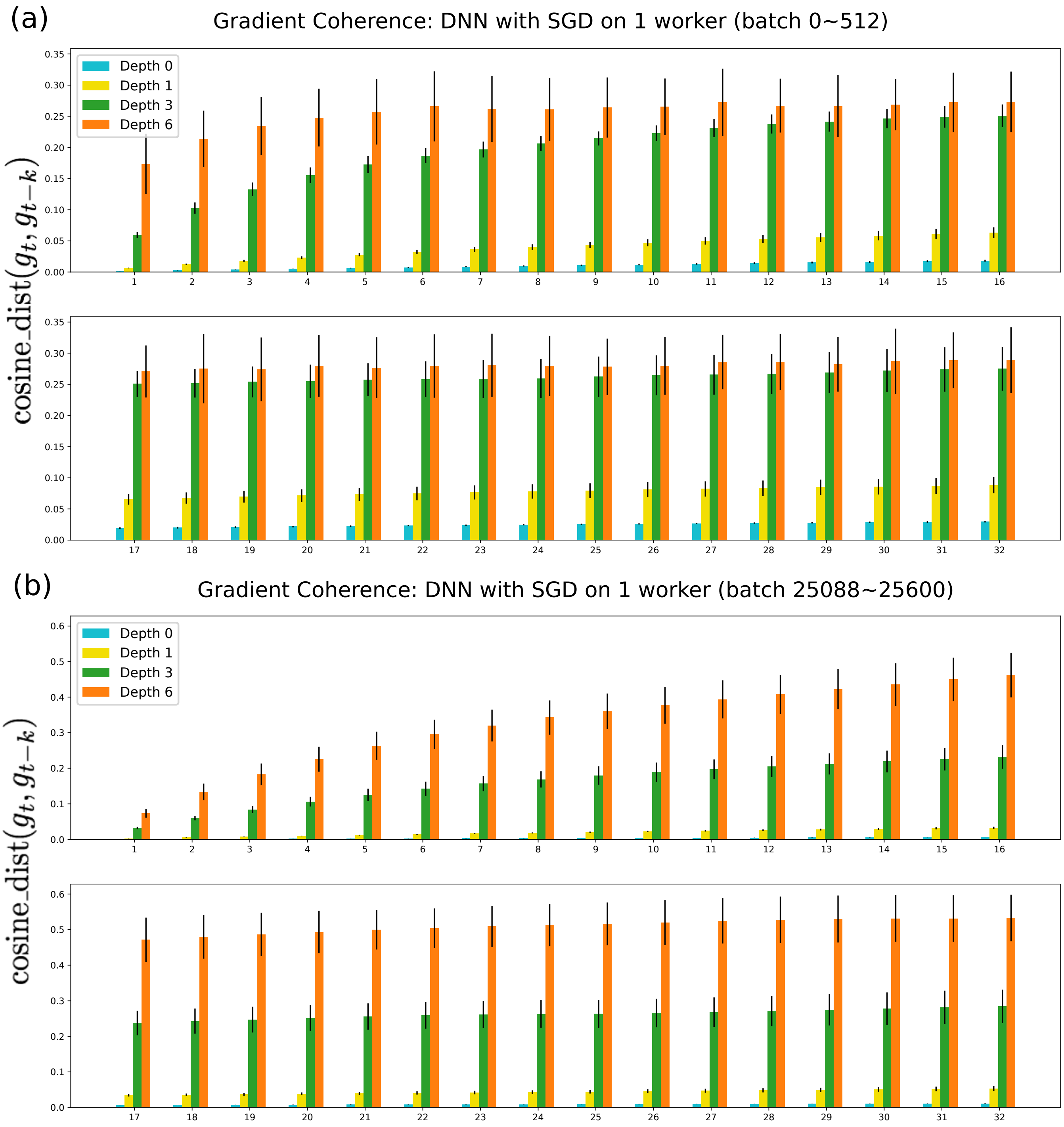


Figure 2.8: Gradient coherence for DNNs with varying depths (depth 0–6) optimized by SGD using 1 worker and no staleness ($s = 0$). The x-axis is $k = 1, ..., 32$. Here we show cosine distance up to 32 batches back. (a) is a snapshot taken from the first 512 batches, while (b) is taken from 512 batches starting from batch 25088 after the algorithms have converged (Fig. 2.7(b), Fig. 2.8(b)). The error bars around the means represent 1 standard deviation computed from 5 randomized runs.

- For DNNs (depth > 0), gradient coherence decreases with increasing model complexity, for both Adam (Fig. 2.7) and SGD (Fig. 2.8). This is evident from the fairly consistent increase of cosine distance with the increasing depth. The increase is not linear, however. In the case of SGD there is a substantial gap in cosine distance between depth 1 and depth 3. while in the case of Adam the gap is much smaller.
- In the case of Adam, the cosine distances are high at the beginning of the algorithm (Fig. 2.7(a)) for DNNs (depth > 0). The low gradient coherence at the beginning of Adam optimization may be the reason that staleness significantly impacts Adam's convergence, especially for deeper architectures, as observed in Fig. 2.4. In particular, for depth 3 and 6, some of the cosine distances even exceed 1, indicating an opposite direction of gradients across batches.
- In the case of SGD applied to DNNs (depth > 0), the gradient coherence actually *decreases* after the algorithm has converged, with the exception of the initial batches (g_{t-1} g_{t-5}). This is consistent with the common understanding that gradients close to the optima are generally less coherent than gradients far away from the optima or any critical point.
- In the case of Adam applied to DNNs (depth > 0), the gradient coherence improves drastically (i.e. lower cosine distances) after the algorithm has reached convergence (Fig. 2.7(b)). This does not imply that the gradient at close to optimum is more coherent, as we have observed the opposite in the case of SGD (see the last point). However, Adam performs a moving averaging over the gradient history, so it is likely that while gradient of each batch is not coherent, the average exhibits more coherence. Additionally, coordinates with higher variances are scaled to have smaller effective learning rate, minimizing their impacts on coherence. These are possible reasons that Adam exhibits lower cosine distance at convergence than SGD (Fig. 2.7(b) vs Fig. 2.8(b)).

We conclude our investigation of MLR and DNNs by noting that staleness in general negatively impacts the convergence. Furthermore, the impact couples with other factors, including the model complexity, the optimization methods, and the number of workers, among others. Staleness also increases the variance of convergence. When the variance is modest, we observe that the effects of staleness are overall consistent.

There are cases where modest staleness creates implicit momentum that actually accelerates the convergence, as observed in SGD, Adagrad, and FTRL optimization. This phenomenon, however, is largely limited to simpler models (e.g., MLR and depth 1 DNNs), low staleness settings (e.g., $s = 16$), or fewer workers (e.g., 1 worker). The negative impact of staleness quickly dominates with the increasing level of staleness and model complexity, and we are back to the regime where convergence difficulties increase quickly with the staleness levels.

2.3.6 Matrix Factorization with Staleness

We now turn to Matrix Factorization (MF) solved by SGD. MF is similar to DNNs in that they are both non-convex (though MF is bi-convex). They are different in that MF updates are sparse as each observation will only generate gradients updating a small subset of the model parameters (Eq. (2.2)). We use batch size of 25000 samples, which is 2.5% of the MovieLens dataset (1 million samples). Therefore, every 40 batches is 1 full pass over the data. We study staleness ranging up to $s = 50$ on 8 workers, which can be as stale as 8.75 data passes ($50 \times 2.5\% \times 7$). We track the following scaled loss function for convergence:

$$\frac{1}{|D|} \sum_{(i,j) \in D_{obs}} ||D_{ij} - \sum_{k=1}^{K} L_{ik} R_{kj}||^2 + \lambda(||L||_F^2 + ||R||_F^2)$$

where $|D|$ is the number of observations in the training dataset.

Fig. 2.9 shows the convergence against the number of batches for staleness 0–50 using 4 and 8 workers. Fig. 2.10 shows the number of batches to reach training loss of 0.5 on 4 and 8 workers under staleness 0 to 50. Fig. 2.11 shows the same metrics but normalized by the values of staleness 0 of each worker setting, respectively. We make a number of observations:

- Staleness has a larger impact on 8 workers than 4 workers, as can be seen in the growth of the number of batches to convergence in Fig. 2.11. In fact, the convergence slow-down in terms of the number of batches (normalized by the convergence for $s = 0$) on 8 workers is more than twice of the slow-down on 4 workers. For example, in Fig. 2.11 the slow-down at $s = 15$ is 3.4, but the slow down at the same staleness level on 8 workers is 8.2. This can be explained by the fact that additional workers amplifies the effect of staleness by (1) creating updates that will be subject to staleness, and (2) missing updates from other workers that are subject to staleness. In the sequel we will see how the number of workers amplify the effective staleness in theoretical analyses.
- Higher staleness leads to a higher variance in convergence. This is the most directly reflected in the convergence curves (Fig. 2.9). Furthermore, the number of workers also affects variance, given the same staleness level. For example, MF with 4 workers incur very low standard deviation up to staleness 20. In contrast, MF with 8 workers already exhibits a large variance at staleness 15. The amplification of staleness from increasing number of workers is similar to the previous bullet point.
- Staleness negatively impacts convergence. For both 4 workers and 8 workers setting, the number of batches needed to reach a certain training loss increases—quite consistently. This is consistent with the observations in DNNs (Section 2.3.4). Moreover, the increase in convergence difficulty is often non-linear in the level of staleness, such as the large jump from $s = 15$ to $s = 20$ on 4 workers, and from $s = 30$ to $s = 40$ on 8 workers.
- On 4 workers, there is a slight speedup when applying staleness 5, compared with staleness 0. This is consistent with the observation in the case of DNNs, where modest level of staleness creates a helpful momentum that accelerates the convergence for certain SGD optimization and its variants (Fig. 2.4).

The results from MF are quite consistent with those from DNNs. We continue to see that staleness negatively impacts the convergence, and its effects are amplified by the number of workers. Staleness also introduces higher variance in the convergence, similar to the case of DNNs.

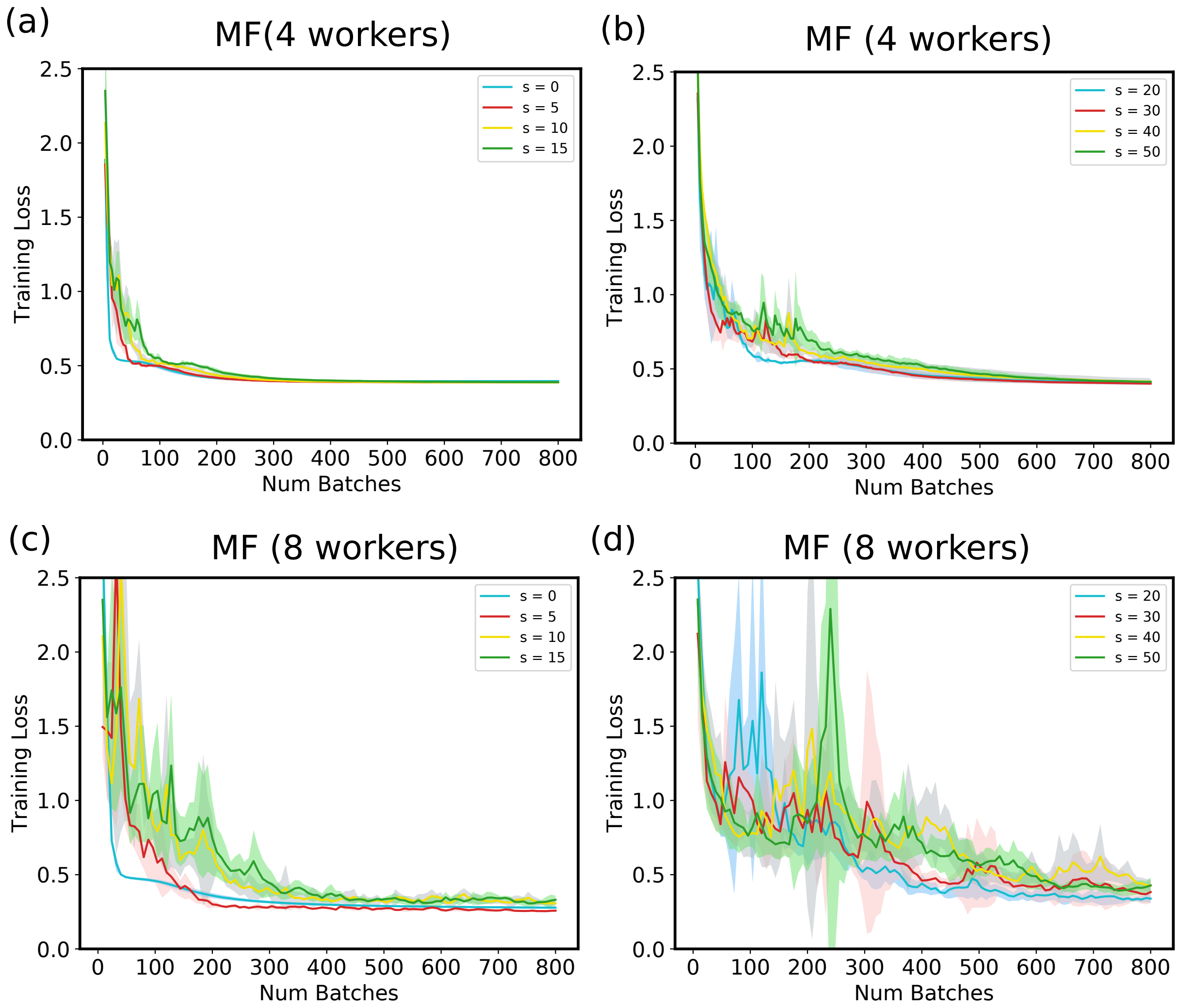


Figure 2.9: Convergence of Matrix Factorization (MF) using 4 and 8 workers, with staleness ranging from 0 to 50. We use the number of batches processed across all workers as the logical time. Shaded area represents 1 standard deviation around the means (represented by the curves) computed on 5 randomized runs.

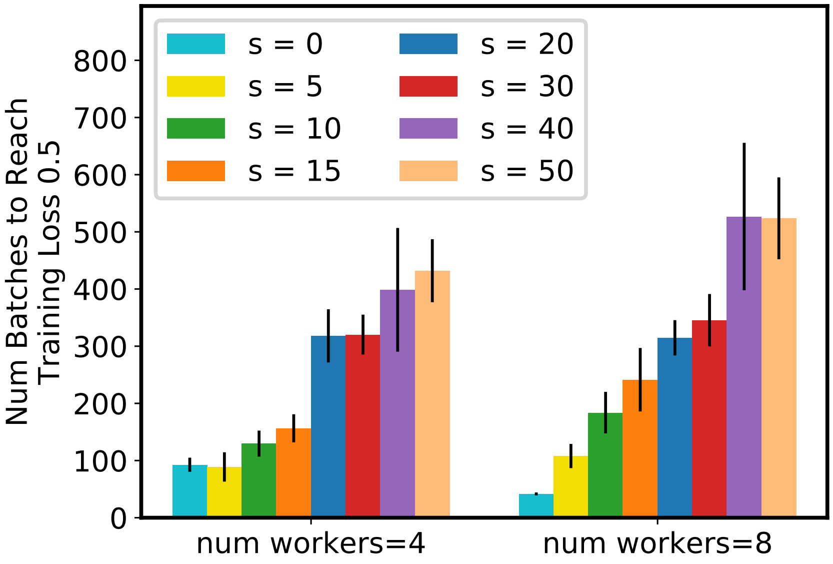


Figure 2.10: The number of batches to reach training loss of 0.5 for Matrix Factorization (MF) optimized by SGD. Mean and error bar (representing 1 standard deviation) are based on 5 randomized runs.

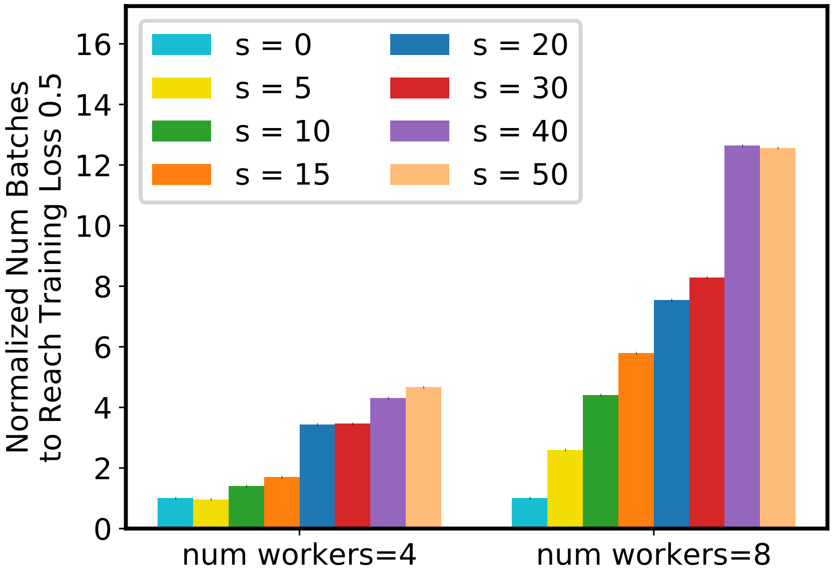


Figure 2.11: The number of batches to reach training loss of 0.5 for Matrix Factorization (MF), normalized by the values of staleness 0 of each worker setting, respectively. This is the normalized version of Fig. 2.10.

2.3.7 Variational Autoencoder with Staleness

Variational Autoencoders (VAEs), unlike MLR, DNNs and MF, are generally solved by variational inference methods that make use of a combination of optimization and sampling (Algorithm 4). Due to this unique hybrid, VAE's solution path has two sources of stochasticity: (1) the randomized mini-batch of data, and (2) the random variables independently sampled to generate gradient updates. We use symmetric encoder and decoder architecture, and therefore L hidden layer implies L hidden layers in the encoder network *and* L hidden layers in the decoder network, for a combined $2L$ weight layers. We use test loss 130 as the convergence target, which is empirically chosen based on the convergence curves (not shown).

Fig. 2.12 shows the number of batches to reach test loss 130 by Variational Autoencoders (VAEs) on 1 worker, under staleness 0 to 16 and 4 SGD variants. We consider VAEs with depth 1, 2, and 3 (the number of layers in the encoder and decoder networks). The number of batches are normalized by $s = 0$ for each VAE depth, respectively. We make a number of observations:

- Staleness increases convergence difficulty for most considered scenarios. This trend is most prominent in SGD and Adam, and mildly present in Adagrad. SGD with momentum exhibits a high variance (Fig. A.5 in the appendix) and thus does not show a consistent trend.
- Deeper VAE exhibits much higher sensitivity to staleness, especially in comparison with DNNs (Fig. 2.4). This is the case even considering that VAE with depth 3 has 6 weight layers, which has comparable number of model parameters and network architecture to DNNs with 6 layers. For example, when optimized by the Adam optimizer, VAEs with depth 3 under $s = 8$ requires 17.5x more batches to converge than under $s = 0$. On the other hand, DNNs with 6 layers optimized by the Adam optimizer takes only <5x more batches to converge under $s = 16$ than under $s = 0$ (Fig. 2.5). This high sensitivity of Adam on VAE is a robust phenomenon, since the error bars in the unnormalized plots (Fig. A.5 in the appendix).
- Adagrad interestingly is not able to reach the target test loss (it only reaches $\sim$136.3 within the experiment window). This is likely due to two factors: (1) Adagrad shrinks learning rate more aggressively than SGD and Adam. With higher level of stochasticity in VAEs (due to the additional sampling stochasticity) this may lead to overly-diminished learning rates. (2) VAE with depth 1 has limited model capacity compared with depth 2 and 3. While deeper networks are more challenging to optimize, they can achieve lower loss when properly optimized. The trade-off may lead to the phenomenon that Adagrad reaches target test loss with depth 2 and 3 but not 1.

Similar to DNNs (Section 2.3.5), we investigate the gradient coherence during the convergence. Fig. 2.13 shows the gradient coherence of VAEs with varying depths optimized by SGD on 1 worker with no staleness ($s = 0$). (We provide a similar plot for the Adam optimizer in the appendix Fig. A.4). Here are a number of observations:

- Similar to DNNs, deeper architecture's higher sensitivity to staleness (Fig. 2.12(a)) is con-

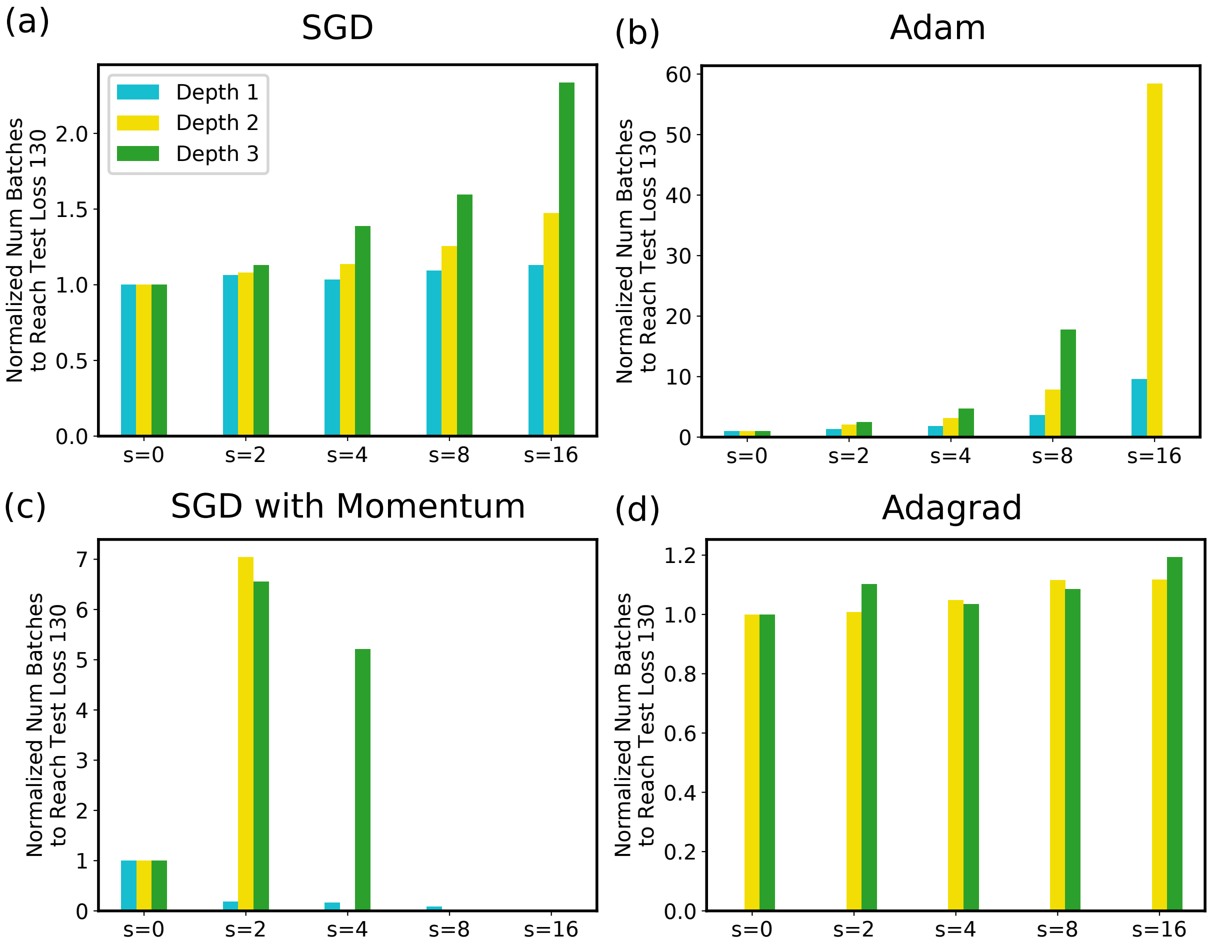


Figure 2.12: The number of batches to reach test loss 130 by Variational Autoencoders (VAEs) on 1 worker, under staleness 0 to 16. We consider VAEs with depth 1, 2, and 3 (the number of layers in the encoder and the decoder networks). The numbers of batches are normalized by $s = 0$ for each VAE depth, respectively. Configurations that do not converge to the desired test loss are omitted, such as Adam optimization for VAE with depth 3 and $s = 16$. The unnormalized version is provided in the appendix (Fig. A.5).

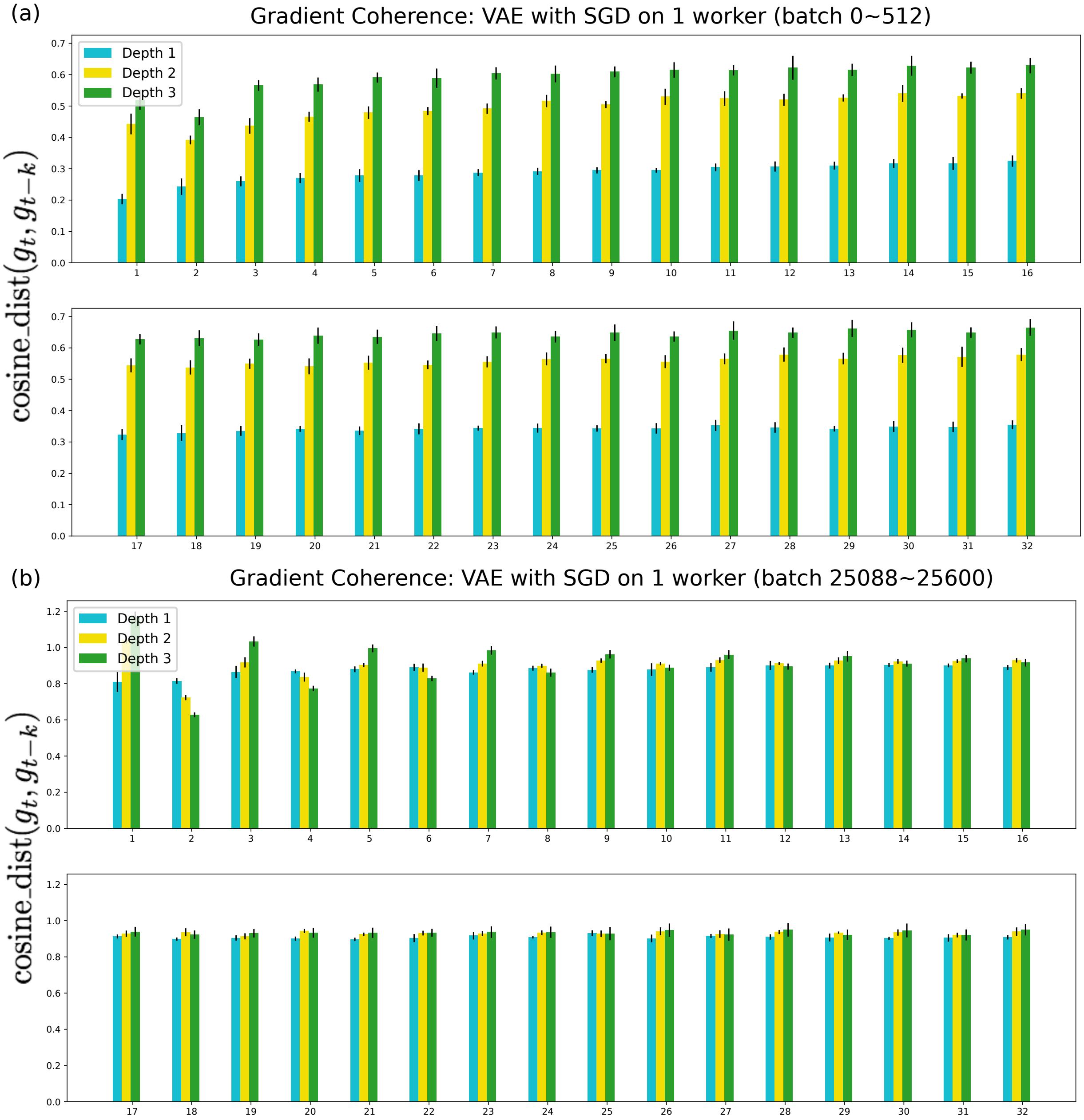


Figure 2.13: Gradient coherence for VAEs with varying depths (depth 1∼3) optimized by SGD optimization using 1 worker with no staleness ($s = 0$). The x-axis is $k = 1, ..., 32$. Here we show cosine distance up to 32 batches back. (a) is a snapshot taken from the first 512 batches, while (b) is taken from 512 batches starting from batch 25088 after algorithms have converged. The error bars around the means represent 1 standard deviation computed from 5 randomized runs.

sistent with the lower gradient coherence observed in Fig. 2.13(a).

- The cosine distance is rather stable across the time steps k, both for the initial phase of the algorithm (Fig. 2.13(a)) and after convergence (Fig. 2.13(b)). This is largely similar to the observation for DNNs (Fig. 2.8). However, the magnitude is much larger in the case of VAEs than DNNs. For example, after convergence has taken place, the cosine distances are generally around $0.8 \sim 1$ for VAEs, whereas the largest magnitude for DNNs is around 0.5 (Fig. 2.8). This is consistent with the previous findings that VAEs are more sensitive to staleness than DNNs, given the same depths. The lower gradient coherence is due to the additional source of stochasticity from sampling.

- Unsurprisingly, at the beginning of the algorithm, the gradient coherence exhibits a clear gap between various depths, with depth 1 the lowest and depth 3 the highest. This is similar to the findings from DNNs. However, after convergence, this differences vanishes, resulting in similar level of cosine distances across varying depths. The same pattern can be observed for VAE optimized by Adam (Fig. A.4 in the appendix). This may be a unique property of gradients derived from variational inference. Indeed, after the algorithms have largely converged, the primary source in the variance of gradients comes from the stochasticity in sampling, which is independent of the model architecture. As a result, the gradient coherence becomes independent of the depths of the networks.

VAEs reveal interesting similarity and differences compared with MLR, DNNs, and MF. Like MLR, DNNs, and MF, VAE's convergence is negatively impacted by staleness, to a higher degree than DNNs of comparable model capacity. The higher sensitivity to staleness due to the increasing depths of the model can be explained by the lower gradient coherence, similar to DNNs. On the other hand, the gaps in gradient coherence due to different depths of the network vanish after convergence happens. This appears to be a unique property due to the variational inference that introduces sampling in the optimization routine.

2.3.8 Latent Dirichlet Allocation with Staleness

All previous models rely on variants of SGD optimization. We now turn to Latent Dirichlet Allocation (LDA) and apply Gibbs sampling, which is the predominant method to estimate LDA parameters and variables.

A key hyperparameter in LDA is the number of topics K, and the model size is linear in K. We use $\frac{D}{\text{num\_iterations} \times P}$ as the batch size, where D is the number of documents, P is the number of workers, and num_iterations $= 10$. In this way we clock 10 times for each data pass, regardless of the number of workers. Given this iteration schedule, the maximum staleness $s = 20$ implies that updates can be up to 2 data passes behind.

Fig. 2.14 and Fig. 2.15 show the convergence of LDA log likelihood using 10 and 100 topics, respectively, with respect to the number of documents processed by Gibbs sampling, with varying staleness and the number of workers. We make a number of observations:

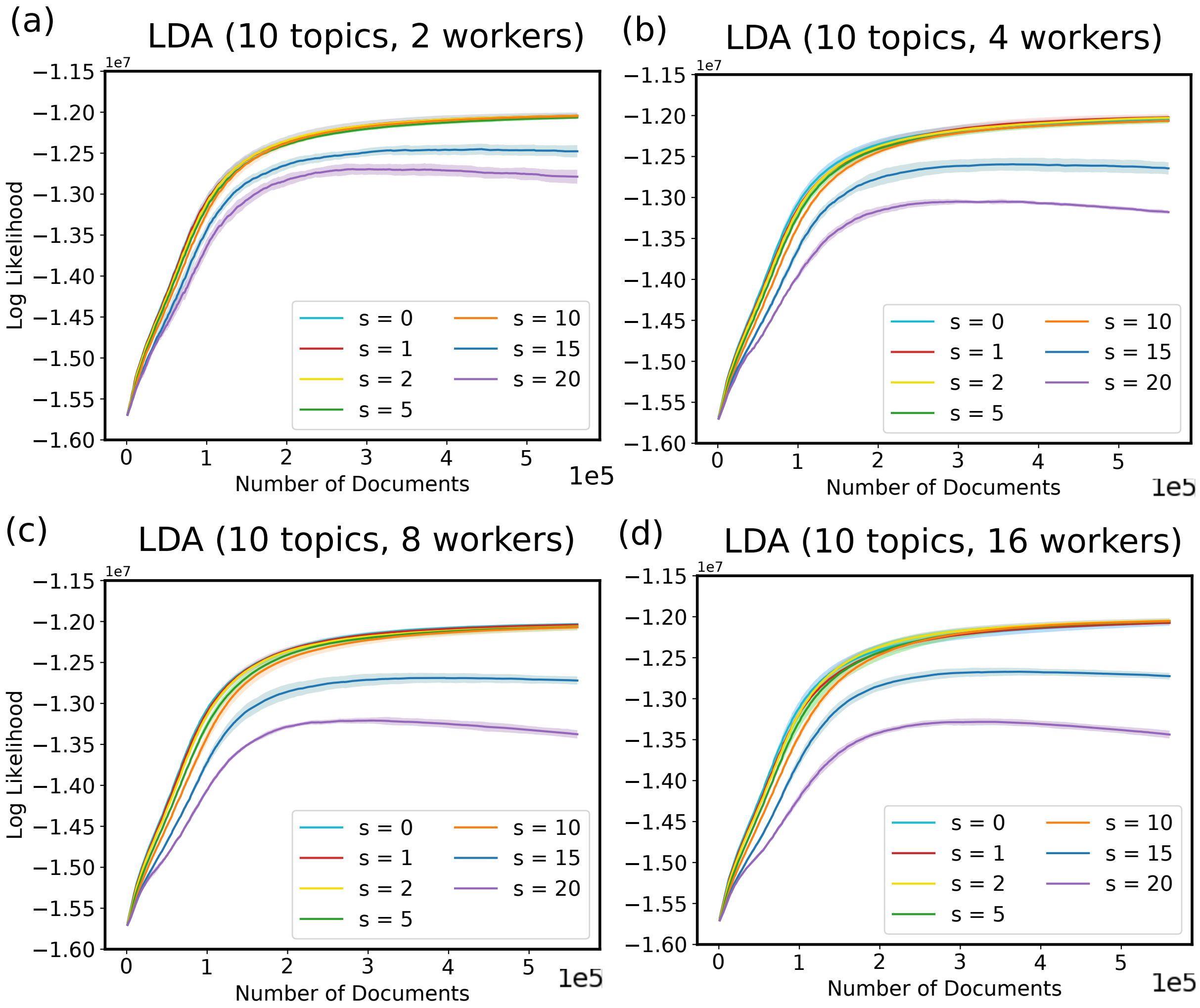


Figure 2.14: Convergence of LDA log likelihood using 10 topics with respect to the number of documents processed by Gibbs sampling, with varying staleness and number of workers. The shaded regions are 1 standard deviation around the means (curves) based on 5 randomized runs.

- Unlike SGD-based algorithms, the convergence curves of Gibbs sampling are highly smooth, even under high staleness and large number of workers. This can be attributed to the log likelihood objective function, which is the sum over many terms (Eq. (2.1)). Since in each sampling we only change the count statistics n_k^i, n_k^w based on a portion of the corpus, the objective value will generally change smoothly.

- Staleness levels under a certain threshold ($s \leq 10$ all result to convergence, following indistinguishable log likelihood trajectory, regardless of the number of topics ($K = 10, 100$) or the number of workers (2–16 workers). Also, there is very minimal variance in those trajectories. However, for higher levels of staleness ($s \geq 15$), Gibbs sampling fails to converge to the same asymptote. The convergence trajectories are distinct, and are sensitive to the number of topics and the number of workers. There appears to be a “phase transition” at a certain staleness level, possibly independent of number of workers and number

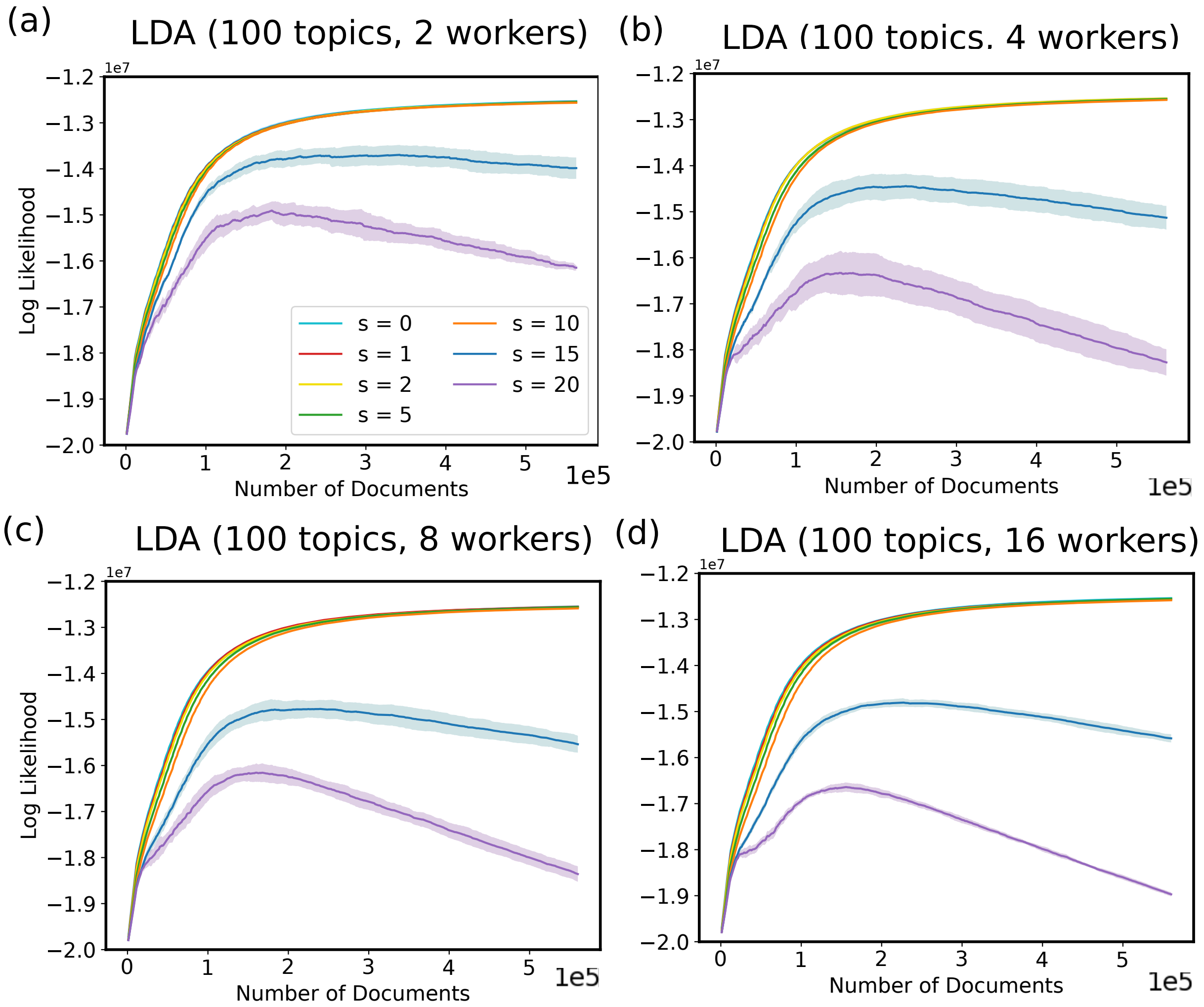


Figure 2.15: Convergence of LDA log likelihood using 100 topics with respect to the number of documents processed by Gibbs sampling, with varying staleness and the number of workers. The shaded regions are 1 standard deviation around the means (curves) based on 5 randomized runs.

of topics, that creates two distinct categories of convergence behaviors.[8]

- The way convergence fails is also interesting. All failures reach certain apex in log likelihood, before degrading in a linear fashion. The value of the inflection point appears to be influenced by the number of workers, as evident in Fig. 2.15. There is also a point at log likelihood value around -1.8×10^7 where the failure cases depart from the converging cases. Prior to that point the convergence trajectories are virtually indistinguishable. These warrant further investigation into the dynamics of Monte Carlo Markov Chain (MCMC) and are beyond the scope of this thesis.

---

[8] We leave the investigation into such a transition as future work.

2.4 Staleness and ML Algorithms

Staleness is a key parameter that governs the convergence of all the studied ML models and algorithms, and possibly all ML algorithms that are based on fixed point iterations. The effects of staleness are also highly problem dependent. In Deep Neural Networks, the staleness slows down deeper models much more than shallower counterparts. When DNNs reduce to MLR, a convex objective, staleness in most parts has minimal effects. For problems with more complex structures like Variational Autoencoders, the convergence slow down due to staleness is much more prominent. Certain stochastic algorithms are also much more sensitive to staleness than others. For example, Adam optimization performs well under low staleness, but can be drastically slowed down by high staleness. SGD, on the contrary, are overall robust against staleness, though its absolute convergence speed is not excellent compared with Adam and other advanced learning rate schedules. Outside of optimization, Gibbs sampling on Latent Dirichlet Allocation (LDA) appears to be highly resistant to staleness up to a certain level, and then undergoes a rapid degradation.

The implications for distributed ML systems are clear from our findings. To achieve actual speed-up, as measured by the wall clock time to convergence, in ML convergence, any distributed ML system needs to overcome the slow-down from staleness. In other words, they must carefully trade off between system throughput gains against the negative impacts of staleness. Many ML methods indeed demonstrate certain robustness against low staleness, which should offer optimization opportunities for system designs. Furthermore, it is paramount to implement the system in a way that minimizes the staleness, such as using a more suitable communication protocol or network bandwidth management. We shall explore some of these ideas in the sequel, as well as providing further theoretical understanding of the impacts of staleness on classes of ML algorithms.

Chapter 3

Analysis of Consistency Models

In distributed systems, a consistency model is a contract specifying the system behaviors which programmers can use to reason the distributed execution. These consistency models are usually defined in terms of the ordering of the operations, such as read and update operations. Conventional consistency models ranges from strong consistency (reader always gets all the previous updates) to the very weak eventual consistency (reader will get some of the updates but no guarantee on which ones or any), with various models in between. Consistency models are critical in distributed system design as they offer trade-offs between consistency, system performance, and availability.

A key idea to achieve efficient large-scale distributed ML is to carefully trade off parameter consistency for increased parameter read throughput (and thus faster algorithm execution), in a manner that guarantees the final output of an ML algorithm is still correct (meaning that it has reached a locally-optimal answer). This is possible because ML algorithms are *error-tolerant*: ML algorithms will converge to a local optimum even when there are errors in the algorithmic procedure itself (such as stochasticity in randomized methods).

In this chapter we present the consistency models commonly used in data parallel ML. We provide extensive analysis of the ML convergence behaviors under the studied consistency models, beyond the existing correctness guarantees. We then use the gleaned insights to improve a consistency model to enables ML programs to reach their solution more quickly.

3.1 Preliminaries

Due to the iterative nature of ML algorithms, ML programs have natural logical clocks, such as a mini-batch or an iteration of the algorithm. Most of the consistency models are defined according to this logical clock, distinct from the global wall clock time (Fig. 3.1).

We use the term "worker" to denote a logical computing unit, which can be a thread or a machine or other suitable computing devices.

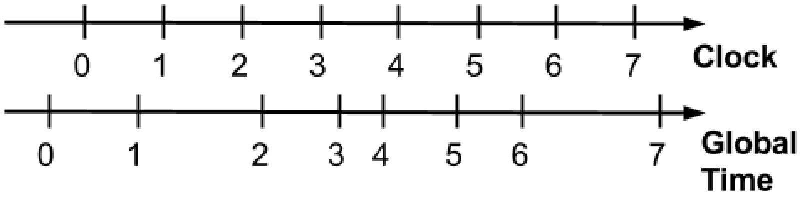


Figure 3.1: Logical clock is distinct from global time.

A key concept in our study is the *staleness*. In the sequential execution, any update to parameters is immediately visible to the subsequent computation. Under the distributed execution, however, it is highly inefficient to enforce such a requirement. Therefore, when the model parameters are shared and updated by multiple workers, there are pending updates generated by one worker that are not visible to all workers. This is the staleness inevitable in distributed ML execution. Staleness, however, does not necessarily leads to inconsistent views of the shared model parameters. In the sequel we will see that some consistency models like Bulk Synchronous Parallel maintain globally consistent parameter views through global synchronization points which communicate and apply these pending updates all at once to all workers.

It is also possible that the staleness leads to inconsistent views of the shared parameters. The insight is that, to an iterative-convergent ML algorithm, inconsistent parameter reads have essentially the same effect as errors due to the algorithmic procedure — implying that convergence to local optima can still happen even under inconsistent reads, *provided the the degree of inconsistency is carefully controlled.*

3.2 Consistency Models for Parameter Servers

We now introduce the consistency models commonly used in distributed ML execution.

3.2.1 Bulk Synchronous Parallel (BSP)

The Bulk Synchronous Parallel (BSP) model, one of the most commonly used synchronization models, requires all workers to see the updates from all other workers in previous clocks before proceeding to the next clock. This is illustrated in Fig. 3.2. This guarantees that all workers have a consistent view of the shared model parameters at the beginning of each clock. Note that this is not the same as sequential execution, as updates produced in the current clock is not visible in the current clock, but only become available in the next clock.

Even though BSP is conceptually simple, it faces certain challenges. First, because the updates generated in a clock is usually communicated after the computation ends, and the next clock cannot start until a worker has received all other workers' updates, the communication in general

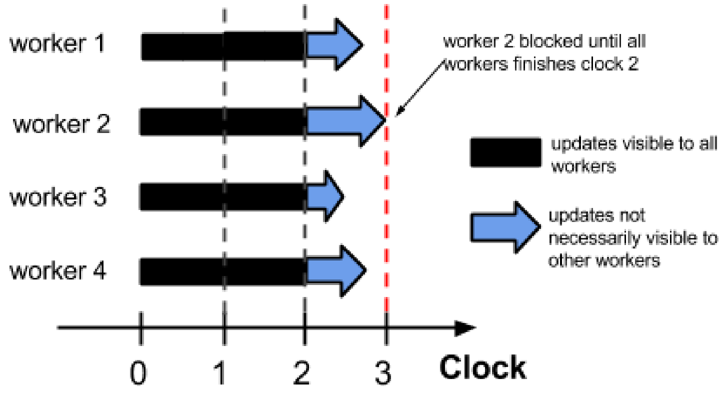

worker 1
worker 2
worker 3
worker 4
worker 2 blocked until all workers finishes clock 2
updates visible to all workers
updates not necessarily visible to other workers
0
1
2
3
Clock

Figure 3.2: An illustration of Bulk Synchronous Parallel (BSP) execution. Worker 2 is blocked at the end of clock 2 as other workers have not completed and communicated updates for clock 2.

does not overlap with the computation and can block the computation. Furthermore, BSP requires the workers to move in lock steps, which can suffer when one of the workers is a straggler. This is a common problem especially in a large cluster, and is well-documented [134]. Frameworks that uses BSP include MapReduce frameworks such as Hadoop [8] and Spark [154] and certain key-value stores [116]. Certain distributed graph computation engines such as GraphLab [101] and Pregel [102] also employs BSP.

3.2.2 Total Asynchronous Parallel (TAP)

Under total asynchronous parallel (TAP) execution, all workers run at their own pace. This eliminates any opportunity for blocking and can fully overlap computation with communication (since computation can proceed without any requirement on the updates being communicated). However, since there can be arbitrary delays in the network, it is difficult to reason the correctness of ML programs. In fact, some TAP systems assumes some bounds on the update delay in order to derive theoretical correctness [120]. Certain systems indeed employ TAP execution [6, 100] with some empirical success. These systems rely on implicitly bounded delays during the execution which is not part of the formal system specification.

3.2.3 Stale Synchronous Parallel (SSP)

Stale Synchronous Parallel (SSP) is a class of *bounded-staleness* consistency models. SSP interpolates between BSP and TAP by providing a looser but still bounded staleness guarantee. Given P workers, SSP assigns each worker a clock c_p that is initially zero. Then, each worker repeats the following operations: (1) perform computation using shared parameters $\mathbf{x}$ stored in the PS, (2) make additive or communicative updates u to the PS, and (3) advance its own clock c_p by

1. SSP allows the fastest worker to be ahead of the slowest worker by no more than *staleness* clocks. In other words, given a worker at logical clock c, reading any parameter must return a value that includes all updates computed before and at clock $c - s - 1$, where s is the staleness threshold. Fig. 3.3 illustrates the SSP execution. SSP subsumes both BSP (staleness $s = 0$) and TAP ($s = \infty$). For a given network condition and size of shared parameters, small staleness blocks computation more often, but maintains less stale parameter views, while large staleness in general reduces wait time for communication. By tuning the staleness bound, SSP can trade off between parameter freshness and system throughput.

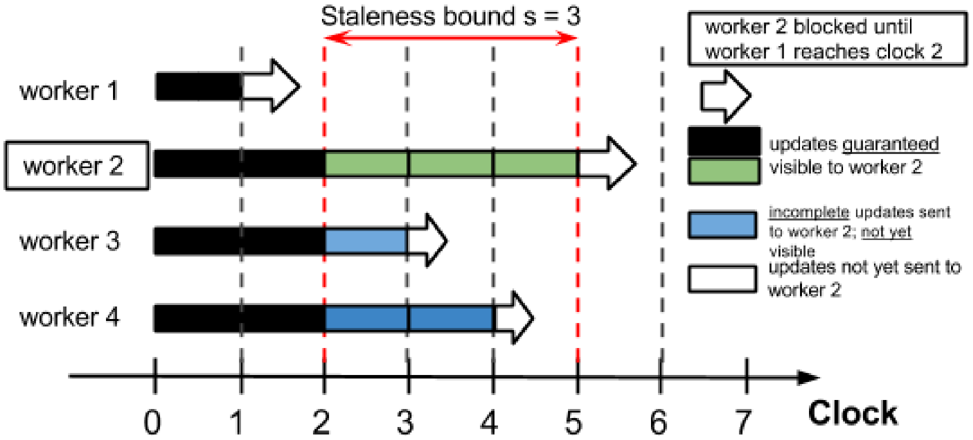

Staleness bound s = 3
worker 2 blocked until worker 1 reaches clock 2
worker 1
worker 2
worker 3
worker 4
updates guaranteed visible to worker 2
incomplete updates sent to worker 2; not yet visible
updates not yet sent to worker 2
0
1
2
3
4
5
6
7
Clock

Figure 3.3: An illustration of Stale Synchronous Parallel (SSP) execution with a staleness bound $s = 3$. The black and green blocks denote the updates that are visible to worker 2; the green updates are visible due to read-my-write consistency. The blue updates are not necessarily visible to worker 2 under SSP. In order to satisfy SSP constraint, worker 2 is blocked at the end of clock 4 because worker 1 has not finished clock 1.

Eager Stale Synchronous Parallel (ESSP)

There are multiple update communication strategies that meet the SSP condition. For example, the communication can occur lazily, only when the computation is blocked. We present *Eager SSP (ESSP)* as a class of implementations that eagerly propagate the updates to reduce empirical staleness beyond what is required by SSP. Fig. 3.4 illustrates the execution under ESSP. ESSP does not provide new guarantees beyond SSP, but in the sequel we will show that by reducing the average staleness ESSP achieves faster convergence, both theoretically and empirically.

3.2.4 Value-bounded Asynchronous Parallel (VAP)

Value-bounded Asynchronous Parallel (VAP) is an idealized consistency model that approximates the strong consistency by bounding the difference in *magnitude* between the strongly consistent view of values on the parameter server and the actual parameter views on the workers.

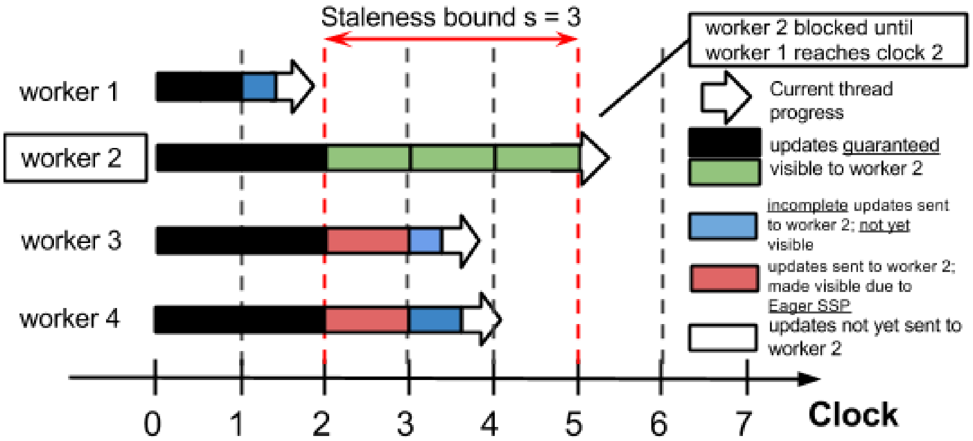

Staleness bound s = 3
worker 2 blocked until worker 1 reaches clock 2
worker 1
worker 2
worker 3
worker 4
Current thread progress
updates guaranteed visible to worker 2
incomplete updates sent to worker 2; not yet visible
updates sent to worker 2; made visible due to Eager SSP
updates not yet sent to worker 2
0
1
2
3
4
5
6
7
Clock

Figure 3.4: An illustration of Eager Stale Synchronous Parallel (ESSP) execution. The execution is similar to that of SSP (Fig. 3.3), except that updates are communicated eagerly as shown in the red blocks.

Formally, let $\mathbf{x}$ represent all model parameters, and assume that each worker in the ML algorithm produces additive updates ($\mathbf{x} \leftarrow \mathbf{x} + \mathbf{u}$, where $\mathbf{u}$ is the update).[1] Given P workers, we say that an update $\mathbf{u}$ is *in transit* if $\mathbf{u}$ has been seen by $P-1$ or fewer workers — in other words, it is yet visible by all workers. Conversely, update $\mathbf{u}$ is no longer in transit once seen by all workers. The VAP requires the following condition:

VAP condition: Let $\mathbf{u}_{p,c}$ be the updates from worker p in clock c that are in transit, and $\mathbf{u}_p := \sum_i \mathbf{u}_{p,i}$. VAP requires that, whenever any worker performs a computation involving the model variables $\mathbf{x}$, the condition $||\sum_p \mathbf{u}_p||_\infty \leq v_{thr}$ holds for a specified (and possibly time-varying) value bound parameter v_{thr} for all workers p, where $||\mathbf{a}||_\infty := \max_j a_j$ is the max-norm. In other words, the aggregated in-transit updates from all workers cannot be too large.

The VAP condition is difficult to be directly implemented efficiently in practice: before any worker can perform computation on $\mathbf{x}$, it must ensure that the in-transit updates from all other workers sum to at most v_{thr} component-wise due to the max-norm. This poses a chicken-and-egg conundrum: for a worker to ensure the VAP condition holds, it needs to know the updates from all other workers — which, in general, requires the same amount of communication as strong consistency. While it may be possible to resort to clock-based mechanism to approximate VAP, direct implementation of VAP is difficult to achieve for a generic PS.

---

[1] This is common in algorithms such as gradient descent ($\mathbf{u}$ being the gradient) and certain sampling methods.

3.3 Theoretical Analysis

In this section, we theoretically analyze VAP and ESSP, and show how they affect ML algorithm convergence. Since ESSP maintains the same guarantees as SSP, our ESSP results naturally extend to the SSP settings. The detailed proofs are presented in the appendix. We ground our analysis on stochastic gradient descent (SGD), as SGD and its variants are the main algorithms for large-scale optimization programs. Our results characterize the convergence of SGD under VAP and ESSP.

3.3.1 SGD for Low Rank Matrix Factorization

We present SGD in the context of a matrix factorization (MF) problem. MF involves decomposing an $N \times M$ matrix D into two low rank matrices $L \in \mathbb{R}^{N\times K}$ and $R \in \mathbb{R}^{K\times M}$ such that $LR \approx D$ gives the prediction of missing entries in D, where $K \ll \min\{M, N\}$ is a user-specified rank. Section 2.2.1 provides an overview of MF formulation and the SGD updates. Since L, R are being updated by each gradient, we store them in the parameter server to allow all workers access them and make additive updates. The data D_{obs} are partitioned into worker nodes and stored locally. See Algorithm 3 for further details.

3.3.2 Preliminaries

We consider the problem in the online learning framework. At each step t the algorithm plays a parameter estimate $\mathbf{x}_t$, and afterwards a loss is revealed $f_t(\mathbf{x})$, which depends on the loss function and the presented data. The familiar empirical risk minimization problem $\min_{\mathbf{x}} \sum_i f_i(\mathbf{x})$ can be cast to the online learning setting by observing that SGD algorithm simply iterates through the data and receives losses for each data based on the current parameter estimate $\mathbf{x}_t$, analogous to the online learning framework.

In this online learning framework we are interested in the difference between the actual incurred loss and the loss achieved by the best possible static parameter estimate $\mathbf{x}^*$. We formalize this notion as *regret*, defined over a sequence $X = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_T\}$:

$$R[X] := \underbrace{\sum_{t=1}^{T} f_t(\mathbf{x}_t)}_{\text{actual incurred loss}} - \underbrace{\inf_{\mathbf{x}} \sum_{t=1}^{T} f_t(\mathbf{x})}_{\text{loss by best static predictor}} = \sum_{t=1}^{T} f_t(\mathbf{x}_t) - \sum_{t=1}^{T} f_t(\mathbf{x}^*)$$

where we assume the sequence length T. When the regret grows sublinearly, such as $R[X] = \mathcal{O}(\sqrt{T})$, then we have the average loss at each step (each datum) $\frac{R(X)}{T} \to 0$ asymptotically.

3.3.3 Theorems for VAP Consistency

We formally establish VAP computation model as follows: given P workers that produce updates at regular intervals which we call "clocks", and let $\mathbf{u}_{p,c} \in \mathbb{R}^n$ be the update from worker p at clock c applied to the system state $\mathbf{x} \in \mathbb{R}^n$ via $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{u}_{p,c}$. Consider the update sequence $\hat{\mathbf{u}}_t$ that orders the updates based on the global time-stamp they are generated. We can define "real-time sequence" $\hat{\mathbf{x}}_t$ as

$$\hat{\mathbf{x}}_t := \mathbf{x}_0 + \sum_{t'=1}^{t} \hat{\mathbf{u}}_{t'}$$

assuming all workers start from the agreed-upon initial state $\mathbf{x}_0$. (Note that $\hat{\mathbf{x}}_t$ is different from the parameter server view as the updates from different workers can arrive the server in a different order due to the network.) Let $\breve{\mathbf{x}}_t$ be the noisy view some worker w sees when generating update $\hat{\mathbf{u}}_t$, i.e., $\hat{\mathbf{u}}_t := G(\breve{\mathbf{x}}_t)$ for some function G. The VAP condition guarantees

$$||\breve{\mathbf{x}}_t - \hat{\mathbf{x}}_t||_\infty \leq v_t = \frac{v_0}{\sqrt{t}} \tag{3.1}$$

where we require the value bound v_t to shrink over time from the initial bound v_0. Notice that $\breve{\mathbf{x}}_t - \hat{\mathbf{x}}_t$ is exactly the updates *in transit* with respect to worker w. We make mild assumptions to avoid pathological cases.[2] We now present the main theorems for SGD under VAP consistency.

Theorem 3.1 (SGD under VAP, convergence in expectation)**.** *Given convex function* $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$ *such that components* f_t *are also convex. We search for minimizer* $\boldsymbol{x}^*$ *via stochastic gradient descent on each component* ∇f_t *with step-size* $\breve{\eta}_t$ *close to* $\eta_t = \frac{\eta}{\sqrt{t}}$ *such that the update* $\hat{\boldsymbol{u}}_t = -\breve{\eta}_t \nabla f_t(\breve{\boldsymbol{x}}_t)$ *is computed on noisy view* $\breve{\boldsymbol{x}}_t$. *The VAP bound follows the decreasing* v_t *described above. Under suitable conditions (*f_t *are* L*-Lipschitz and bounded diameter* $D(x\|x') \leq F^2$*),*

$$R[X] := \sum_{t=1}^{T} f_t(\breve{\boldsymbol{x}}_t) - f(\boldsymbol{x}^*) = \mathcal{O}(\sqrt{T})$$

and thus $\frac{R[X]}{T} \to 0$ *as* $T \to \infty$.

Theorem 3.1 implies that the worker's noisy VAP view $\breve{\mathbf{x}}_t$ converges to the global optimum $\mathbf{x}^*$, as measured by f, in expectation at the rate $\mathcal{O}(T^{-1/2})$. The analysis is similar to that in [65], but we use the real-time sequence $\hat{\mathbf{x}}_t$ as our reference sequence and VAP condition instead of SSP. The proof is presented in Appendix. Loosely speaking, Theorem 3.1 shows that VAP execution is unbiased. We now present a new bound on the variance of the convergence.

---

[2] To avoid pathological cases where a worker is delayed indefinitely, we assume that each worker's updates are finitely apart in sequence $\hat{\mathbf{u}}_t$. In other words, all workers generate updates with a sufficient frequency. For SGD, we further assume that each worker updates its step-sizes sufficiently often that the local step-size $\breve{\eta}_t = \frac{\eta}{\sqrt{t-r}}$ for some bounded drift $r \geq 0$ and thus $\breve{\eta}_t$ is close to the global step size schedule $\eta_t = \frac{\eta}{\sqrt{t}}$.

Theorem 3.2 (SGD under VAP, bounded variance)**.** *Assuming $f(\boldsymbol{x})$, $\breve{\eta}_t$, and v_t similar to theorem 3.1 above, and further assume that $f(\boldsymbol{x})$ has bounded and invertible Hessian, Ω^* defined at optimal point $\boldsymbol{x}^*$. Let $\mathrm{Var}_t := \mathbb{E}[\breve{\boldsymbol{x}}_t^2] - \mathbb{E}[\breve{\boldsymbol{x}}_t]^2$, and $\breve{\boldsymbol{g}}_t = \nabla f_t(\breve{\boldsymbol{x}}_t)$ be the gradient, then:*

$$\begin{aligned} \mathrm{Var}_{t+1} &= \mathrm{Var}_t - 2cov(\hat{\boldsymbol{x}}_t, \mathbb{E}^{\Delta_t}[\breve{\boldsymbol{g}}_t]) + \mathcal{O}(\delta_t) & (3.2) \\ &+ \mathcal{O}(\breve{\eta}_t^2 \rho_t^2) + \mathcal{O}^*_{\delta_t} & (3.3) \end{aligned}$$

near the optima $\boldsymbol{x}^*$. The covariance $cov(\boldsymbol{a}, \boldsymbol{b}) := \mathbb{E}[\boldsymbol{a}^T \boldsymbol{b}] - \mathbb{E}[\boldsymbol{a}^T]\mathbb{E}[\boldsymbol{b}]$ uses inner product. $\delta_t = ||\boldsymbol{\delta}_t||_\infty$ and $\boldsymbol{\delta}_t = \breve{\boldsymbol{x}}_t - \hat{\boldsymbol{x}}_t$. $\rho_t = ||\breve{\boldsymbol{x}}_t - \boldsymbol{x}^*||$. Δ_t is a random variable capturing the randomness of update $\hat{\boldsymbol{u}}_t = -\eta_t \breve{\boldsymbol{g}}_t$ conditioned on $\hat{\boldsymbol{x}}_t$ (see the appendix).

$cov(\hat{\mathbf{x}}_t, \mathbb{E}^{\Delta_t}[\breve{\boldsymbol{g}}_t]) \geq 0$ in general as the change in $\hat{\mathbf{x}}_t$ and average gradient $\mathbb{E}^{\Delta_t}[\breve{\boldsymbol{g}}_t]$ are of the same direction. Theorem 3.2 implies that under VAP the variance decreases in successive iterations for sufficiently small δ_t, which can be controlled via VAP threshold v_t. However, the VAP condition requires tight synchronization as $\delta_t \to 0$. This motivates our following analysis of the SSP model.

3.3.4 Theorems for SSP Consistency

We return to the (p, c) indexing such that $\mathbf{u}_{p,c}$ is the update generated by worker p at clock c. Under the SSP worker p at clock c only has access to a noisy view $\tilde{\mathbf{x}}_{p,c}$ of the system state ($\tilde{\mathbf{x}}$ is different from $\breve{\mathbf{x}}$, the noisy view in VAP). Update $\mathbf{u}_{p,c} = G(\tilde{\mathbf{x}}_{p,c})$ is computed on the noisy view $\tilde{\mathbf{x}}_{p,c}$ for some function $G(\cdot)$. Assuming all workers start from the agreed-upon initial state $\mathbf{x}_0$, the SSP condition is:

SSP Bounded-Staleness (formal): For a fixed staleness parameter s, the noisy state $\tilde{\mathbf{x}}_{p,c}$ is equal to

$$\tilde{\mathbf{x}}_{p,c} = \mathbf{x}_0 + \underbrace{\left[\sum_{c'=1}^{c-s-1} \sum_{p'=1}^{P} \mathbf{u}_{p',c'}\right]}_{\text{guaranteed pre-window updates}} + \underbrace{\left[\sum_{(p',c') \in \mathcal{S}_{p,c}} \mathbf{u}_{p',c'}\right]}_{\text{best-effort in-window updates}},$$

for some $\mathcal{S}_{p,c} \subseteq \mathcal{W}_{p,c} = \{1, ..., P\} \times \{c-s, ..., c+s-1\}$ which is some subset of updates in the $2s$ window issued by all P workers during clock $c-s$ to $c+s-1$. The noisy view consists of (1) guaranteed pre-window updates for clock 1 to $c-s-1$ (the black updates in Fig. 3.3), and (2) best-effort updates indexed by $\mathcal{S}_{p,c}$ (the red updates in Fig. 3.4).[3] We introduce a clock-major index t (Fig. 3.5):

$$\tilde{\mathbf{x}}_t := \tilde{\mathbf{x}}_{(t \bmod P), \lfloor t/P \rfloor} \qquad\qquad \mathbf{u}_t := \mathbf{u}_{(t \bmod P), \lfloor t/P \rfloor}$$

---

[3] In contrast to [65], we do not assume read-my-write.

and analogously for $\mathcal{S}_t$ and $\mathcal{W}_t$. We can now define a reference sequence (distinct from $\hat{\mathbf{x}}_t$ in VAP) which we informally refers to as the "true" sequence:

$$\mathbf{x}_t = \mathbf{x}_0 + \sum_{t'=0}^{t} \mathbf{u}_{t'} \tag{3.4}$$

The sum loops over workers ($t \bmod P$) and clocks $\lfloor t/P \rfloor$. Notice that this sequence is distinct to the server view.

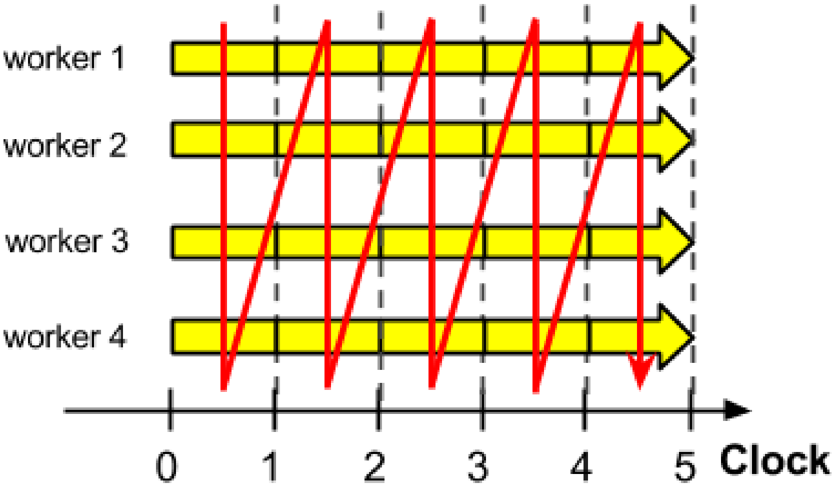


Figure 3.5: The ordering of the updates for analyzing SSP.

Given the update sequence, we now present the main theorems for SSP.

Theorem 3.3 (SGD under SSP, convergence in expectation [65], Theorem 1)**.** *Given convex function* $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$ *with suitable conditions as in Theorem 3.1, we use gradient descent with updates* $\boldsymbol{u}_t = -\eta_t \nabla f_t(\tilde{\boldsymbol{x}}_t)$ *generated from noisy view* $\tilde{\boldsymbol{x}}_t$ *and* $\eta_t = \frac{\eta}{\sqrt{t}}$*. Then*

$$R[X] := \sum_{t=1}^{T} f_t(\tilde{\boldsymbol{x}}_t) - f(\boldsymbol{x}^*) = \mathcal{O}(\sqrt{T})$$

and thus $\frac{R[X]}{T} \to 0$ *as* $T \to \infty$*.*

Theorem 3.3 is the SSP counterpart of Theorem 3.1 for VAP. The analysis of Theorem 3.3 only uses the worst-case SSP bounds. However, in practice many updates are much less stale than the SSP bound, which we empirically verify in the sequel.

We now use moment statistics to further characterize the convergence. We begin by decomposing $\tilde{\mathbf{x}}_t$. Let $\bar{u}_t := \frac{1}{P(2s+1)} \sum_{t' \in \mathcal{W}_t} ||\mathbf{u}_{t'}||_2$ be the average of ℓ_2 norm of the updates. We can write the noisy view $\tilde{\mathbf{x}}_t$ as

$$\tilde{\mathbf{x}}_t = \mathbf{x}_t + \bar{u}_t \boldsymbol{\gamma}_t \tag{3.5}$$

where $\boldsymbol{\gamma}_t \in \mathbb{R}^d$ is a vector of random variables whose randomness lies in the network communication. Note that the decomposition in Eq. (3.5) is always possible since $\bar{u}_t = 0$ iff $\mathbf{u}_{t'} = \mathbf{0}$ for

all updates $\mathbf{u}_{t'}$ in the $2s$ window. Given the SSP condition and L, the Lipschitz constant such that f_t are L-Lipschitz, we can bound $\bar{u}_t$ and $\boldsymbol{\gamma}_t$:
Lemma 3.1. *$\bar{u}_t \leq \frac{\eta}{\sqrt{t}} L$ and $\gamma_t := ||\boldsymbol{\gamma}_t||_2 \leq P(2s+1)$.*

Therefore $\mu_\gamma = \mathbb{E}[\gamma_t]$ and $\sigma_\gamma = var(\gamma_t)$ are well-defined. We now provide an non-asymptotic exponential tail-bound characterizing convergence in *finite* steps.
Theorem 3.4 (SGD under SSP, convergence in probability)**.** *Given convex function $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$ such that components f_t are also convex. We search for minimizer $\boldsymbol{x}^*$ via gradient descent on each component ∇f_t under SSP with staleness parameter s and P workers. Let $\boldsymbol{u}_t := -\eta_t \nabla_t f_t(\tilde{\boldsymbol{x}}_t)$ with $\eta_t = \frac{\eta}{\sqrt{t}}$. Under suitable conditions (f_t are L-Lipschitz and bounded divergence $D(x||x') \leq F^2$), we have*

$$P\left[\frac{R[X]}{T} - \frac{1}{\sqrt{T}}\left(\eta L^2 + \frac{F^2}{\eta} + 2\eta L^2 \mu_\gamma\right) \geq \tau\right] \\ \leq \exp\left\{\frac{-T\tau^2}{2\bar{\eta}_T \sigma_\gamma + \frac{2}{3}\eta L^2 (2s+1) P \tau}\right\}$$

where $R[X] := \sum_{t=1}^{T} f_t(\tilde{x}_t) - f(x^*)$, and $\bar{\eta}_T = \frac{\eta^2 L^4 (\ln T + 1)}{T} = o(T)$.

This means that $\frac{R[X]}{T}$ converges to $O(T^{-1/2})$ in probability with an exponential tail-bound. Also note that the convergence is faster for smaller μ_γ and σ_γ. This key result implies that the rate of convergence is faster when there is less staleness and when the degree of staleness across time is more concentrated (smaller variance). Therefore if the underlying implementation can systematically reduce the staleness, the average empirical staleness will be small and concentrated on small staleness values. This theoretical result is consistent with the intuition that less staleness is better. We point out that this result demonstrates that convergence quality depends on the runtime staleness distributions, manifested in $\mu_\gamma, \sigma_\gamma$, and not simply the maximal bound on the staleness (the staleness parameter s). In that sense this is a stronger statement than the worst-case bounds in Theorem 3.3.

To derive the variance bounds, we need a few mild assumptions on the staleness γ_t:
Assumption 3.1. *γ_t are i.i.d. random variable with well-defined mean μ_γ and variance σ_γ.*
Assumption 3.2. *γ_t is independent of $\boldsymbol{x}_t$ and $\boldsymbol{u}_t$.*

Assumption 1 is satisfied by Lemma 3.1, while Assumption 2 is valid since γ_t are only influenced by the computational load and network bandwidth at each machine, which are themselves independent of the actual values of the computation ($\mathbf{u}_t$ and $\mathbf{x}_t$). We now present an SSP variance bound.
Theorem 3.5 (SGD under SSP, decreasing variance)**.** *Given the setup in Theorem 3.4 and assumption 1-2. Further assume that $f(\boldsymbol{x})$ has bounded and invertible Hessian Ω^* at optimum $\boldsymbol{x}^*$ and γ_t is bounded. Let $\mathrm{Var}_t := \mathbb{E}[\tilde{\boldsymbol{x}}_t^2] - \mathbb{E}[\tilde{\boldsymbol{x}}_t]^2$, $\boldsymbol{g}_t = \nabla f_t(\tilde{\boldsymbol{x}}_t)$, η_t be the learning rate, then for $\tilde{\boldsymbol{x}}_t$ near the optima $\boldsymbol{x}^*$ such that $\rho_t = ||\tilde{\boldsymbol{x}}_t - \boldsymbol{x}^*||$ and $\xi_t = ||\boldsymbol{g}_t|| - ||\boldsymbol{g}_{t+1}||$ are small:*

$$\mathrm{Var}_{t+1} = \mathrm{Var}_t - 2\eta_t cov(\boldsymbol{x}_t, \mathbb{E}^{\Delta_t}[\boldsymbol{g}_t]) + \mathcal{O}(\eta_t \xi_t) \tag{3.6}$$

$$+ \mathcal{O}(\eta_t^2 \rho_t^2) + \mathcal{O}^*_{\gamma_t} \tag{3.7}$$

where covariance $cov(\boldsymbol{a}, \boldsymbol{b}) := \mathbb{E}[\boldsymbol{a}^T\boldsymbol{b}] - \mathbb{E}[\boldsymbol{a}^T]\mathbb{E}[\boldsymbol{b}]$ *uses inner product.* $\mathcal{O}^*_{\gamma_t}$ *are high order* ($\geq$ *5th) terms involving* $\gamma_t = ||\boldsymbol{\gamma}_t||_\infty$*.* Δ_t *is a random variable capturing the randomness of update* $\boldsymbol{u}_t$ *conditioned on* $\boldsymbol{x}_t$*.*

where Δ_t is defined in the Appendix. As argued before, $cov(\mathbf{x}_t, \mathbb{E}^{\Delta_t}[\boldsymbol{g}_t]) \geq 0$ in general. Therefore the theorem is a key stability result that implies that Var_t monotonically decreases when SGD is close to an optimum (small ρ_t and ξ_t). This is important to ensure that the convergence will experience decreasing fluctuation from both the algorithm and system delays.

3.3.5 Comparison of VAP and ESSP

From Theorem 3.2 and 3.5 we see that both VAP and (E)SSP generally decrease the variance of the model parameters over iterations. However, VAP convergence is much more sensitive to its tuning parameter (the VAP threshold) than (E)SSP, whose tuning parameter is the staleness s. This is evident from the $O(\delta_t)$ term in Eq. 3.3, which is bounded by the VAP threshold. In contrast, (E)SSP's variance only involves staleness γ_t in high order terms $\mathcal{O}^*_{\gamma_t}$ (Eq. 3.7), where γ_t is bounded by the SSP staleness parameter (Lemma 3.1). This implies that staleness-induced variance vanishes quickly in (E)SSP. The main reason for (E)SSP's weak dependency on staleness is because it "factors in" the SGD step size: as the algorithm approaches an optimum, the updates automatically become more fine-grained (i.e. their magnitude decreases), which is conducive for lowering variance. On the other hand, the VAP threshold forces a minimum size on updates, and without adjusting this threshold accordingly, the VAP updates cannot become more fine-grained.

An intuitive analogy is that of postmen: VAP is like a postman who only ever delivers mail above a certain weight threshold δ. (E)SSP, on the other hand, is like a postman who delivers mail late, but no later than s days. Intuitively, the (E)SSP postman is more reliable than the VAP postman due to his regularity. The only way for the VAP postman to be reliable, is to decrease the weight threshold. This is important when the algorithm approaches convergence, because the algorithm's updates become diminishingly small. However, there are two drawbacks to decreasing δ: first, much like step-size tuning, it must be done at a carefully controlled rate — this requires either specific knowledge about the ML problem, or a sophisticated, automatic scheme (that may also be domain-specific). Second, as δ decreases, VAP can produce more frequent communication and reduce the throughput.

In contrast to VAP, ESSP does not suffer as much from these drawbacks, because: (1) the SSP family has a weaker theoretical dependency on the staleness threshold (than VAP does on its value-bound threshold), thus it is usually unnecessary to decrease the staleness as the ML algorithm approaches convergence. This is evidenced by [65], which achieved stable convergence even though they did not decrease staleness gradually during ML algorithm execution. (2) Because ESSP proactively pushes out fresh parameter values, the empirical distribution of stale reads usually exhibit very low staleness, regardless of the actual staleness threshold used, a subject we shall revisit in the sequel. Hence, fine-grained tuning of the staleness threshold is rarely necessary under ESSP.

`Get(key)` Read a parameter indexed by `key`.
`GetRow(row_key)` Read a row of parameters indexed by `row_key`. A row consists of a static group of parameters.
`Inc(key, delta)` Increment the parameter `key` by `delta`.
`IncRow(row_key, deltas)` Increment the row `row_key` by `deltas`.
`Clock()` Signal end of iteration.

Table 3.1: **Bösen Client API.** The API is similar to key-value interfaces. The user program can read parameters via `Get` and `GetRow` (batched reads) and make (additive) updates via `Inc` and `IncRow` (batched updates). Since bounded staleness is defined with respect to logical clock, the user program needs to signal the completion of a logical unit of work via `Clock`.

3.4 Bösen System Overview

The theory suggests that an ESSP implementation using eager parameter updates should outperform a SSP implementation using stalest parameters, when network bandwidth is sufficient. To verify this, we implement ESSP in Parameter Server framework (Section 1.2.2) called Bösen. Bösen is a parameter server (PS) with bounded staleness parallel (Section 3.2.3).[4] Because Bösen also satisfies bounded staleness model's requirements, Bösen inherits the formal convergence guarantees, and enjoys high iteration throughput that is better than Bulk Synchronous Parallel and close to Totally Asynchronous Parallel (TAP) systems.

3.4.1 API and Bounded Staleness Consistency

Bösen PS consists of a *client library* and *parameter server partitions* (Figure 3.6). The client library provides the Application Programming Interface (API) for reading/updating model parameters, while the PS partition stores and maintains the model parameters. In terms of usage, Bösen closely follows other key-value stores: once a ML program process is linked against the client library, any thread in that process may read/update model parameters concurrently. The user runs a Bösen ML program by invoking as many server partitions and ML application *compute processes* (which use the client library) as needed, across multiple machines.

---

[4] "The system is named after the piano maker Bösendorfer.

Bösen Client API

Bösen's API abstracts consistency management and networking operations away from application, and presents a simple key-value interface (Table 3.1). The application may use `Get()` read parameters, and `Inc()` to increment the parameter by some delta. To signal progress with respect to logical clock, the client application invokes `Clock()` at the end of a unit of work. Each thread is considered a worker and calls `Clock()` separately. We use the term "clock" and "iteration" interchangeably in this chapter. In order to exploit locality in ML applications and thus amortize the overhead of operating on concurrent data structures and network messaging, Bösen allows applications to statically partition the parameters into batches called *row*s. A row is a set of parameters that are usually accessed together. A row can also be the unit of communication between client and server: `RowGet()` is provided to read a row by its key, and `RowInc()` applies a set of deltas to multiple elements in a row. ML applications can use one of the built-in dense and sparse row data types, or customize the row data structure for maximal flexibility. A row of parameters can flexibly represent different objects in ML applications. For example, in LDA, rows are word-topic vectors. In MF, rows in a table are rows in the factor matrices.

Bounded Staleness Consistency

Bösen client library maintains model parameter cache $\tilde{A}$ on each worker machine locally to avoid repeated remote reads of the model parameters A on the server. These synchronization of model replicas are maintained by consistency managers on the worker nodes (client library) and the server. Together they enforce consistency requirements on the local parameter image $\tilde{A}$, thus ensuring correct ML program execution even under the *worst-case* delays. The ML program's tolerance to staleness is specified as the *staleness threshold* s, a non-negative integer defined by the user.

The consistency manager works by blocking client process worker threads when reading parameters, until the local model image $\tilde{A}$ has been updated to meet the consistency requirements. Bounded staleness puts constraints on parameter age; Bösen will block if $\tilde{A}$ is older than the worker's current iteration by s or more (i.e., $c_p - Age(\tilde{A}) > s$), where c_p is the worker's clock. $\tilde{A}$'s age is defined as the largest iteration such that all updates generated at and before that iteration are reflected in $\tilde{A}$. The resulting effect is that a `GET` issued by a worker at clock c_p is guaranteed to observe all updates generated in clock $[0, c_p - s - 1]$.

Bulk Synchronous Parallel (BSP). When the staleness threshold is set to 0, bounded staleness consistency reduces to the classic BSP model. The BSP model is a gold standard for correct ML program execution; it requires all updates computed in previous iterations to be made visible before the current iteration starts. A conventional BSP implementation may use a global synchronization barrier; Bösen's consistency manager achieves the same result by requiring calls to `PS.Get()` and `PS.GetRow()` at iteration t to reflect all updates, made by any thread, before its $(t-1)$-th call to `PS.Clock()`. Otherwise, the call to `PS.Get()` or `PS.GetRow()` blocks until the required updates are received.

3.4.2 System Architecture

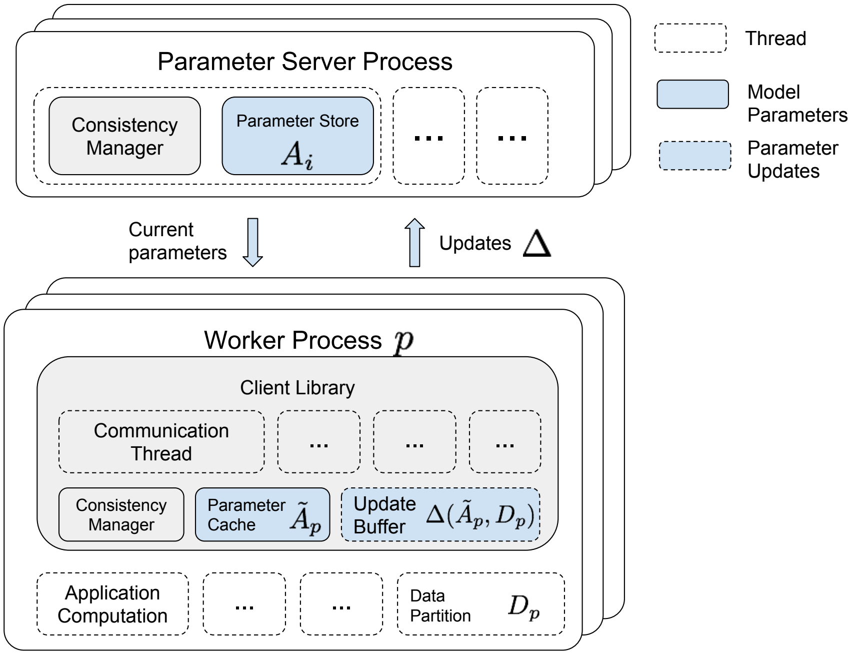

Parameter Server Process
Consistency Manager
Parameter Store A_i
...
...
Thread
Model Parameters
Parameter Updates
Current parameters
Updates Δ
Worker Process p
Client Library
Communication Thread
...
...
...
Consistency Manager
Parameter Cache $\tilde{A}_p$
Update Buffer $\Delta(\tilde{A}_p, D_p)$
Application Computation
...
...
Data Partition D_p

Figure 3.6: **Bösen Parameter Server Architecture.** The PS consists of client processes (bottom) and the server partitions (top). The Bösen client library maintains cached image of the server parameters $\tilde{A}_p$ (p indexes the worker processes). The user application instantiates compute threads, which have access to data partition D_p, and generates updates $\Delta(\tilde{A}_p, D_p)$ that are buffered by the Bösen client library. The user program's access to the parameter cache is modulated by the consistency manager to ensure bounded staleness conditions. The communication with the server is performed by background communication threads, separate from compute threads. The server processes maintain partitioned master copy of parameters A_i, where i indexes the server threads.

We now describes Bösen's system architecture and focus on its realization of the bounded staleness consistency.

Client Library

The client library provides access to the model parameters A. This can come from the locally cached version $\tilde{A}$, or from the server when $\tilde{A}$ does not satisfy the staleness requirement. This is done through three components (Fig. 3.6): (1) a *parameter cache* that caches a partial or complete image of the model, $\tilde{A}$, at the client, in order to serve read requests made by compute threads; (2) an *update buffer* that buffers updates applied by compute threads via `PS.Inc()` and `PS.RowInc()`; (3) a group of *client communication threads* (distinct from compute threads) that perform synchronization of the local model cache and buffered updates with the servers' master copies, concurrently as the compute threads execute the application algorithm.

The parameters cached at a client are hash partitioned among the client communication threads. Each client communication thread needs to access only its own parameter partition when reading the computed updates and applying up-to-date parameter values to minimize lock contention. The client parameter cache and update buffer allow concurrent reads and writes from worker threads. Similar to [37], the cache and buffer use static data structures, and pre-allocate memory for repeatedly accessed parameters to minimize the overhead of maintaining a concurrent hash table.

In each compute process, locks are needed for shared access to parameters and buffered update entries. The buffered updates are coalesced since they are commutative and associative. In order to amortize the runtime cost of concurrency control, we allow applications to define parameter key ranges we call *rows* (Section 3.4.1). Parameters in the same row share one lock for access to their parameter caches, and one lock for access to their update buffers.

When serving read requests (`Get()` and `RowGet()`) from worker threads, the client parameter cache is searched first. A read request is sent to the server processes only if either the requested parameter is not in the cache or the cached parameter's age does not satisfy the staleness requirement. The reading compute thread blocks until the parameter's staleness is within the threshold. When writes are invoked by the application thread, updates are inserted into the update buffer, and, optionally, the client's own parameter cache is also updated.

Once all compute threads in a client process have called `PS.Clock()` to signal the end of a unit of work (e.g. an iteration), the client communication threads release buffered model updates to servers.

Server Partitions

The master copy of the model's parameters, A, is hash partitioned, and each partition is assigned to one server thread. The server threads may be distributed across multiple server processes and physical machines. As model updates are received from client processes, the addressed server thread updates the master copy of its model partition. When a client read request is received, the corresponding server thread registers a callback for that request; once a server thread has applied all updates from all clients for a given clock, it walks through its callbacks and sends the up-to-date model parameter values.

Ensuring Bounded Staleness

Bounded staleness is ensured by coordination of clients and server partitions using *clock messages*. When a client request a table-row for the first time, it registers a callback on the server. This is the only time the client makes read request to the server. Subsequently, when a server table's clock advances from getting the clock tick from all clients, it pushes out the table-rows to the respective registered clients. This differs from the SSPTable in [65] where the server passively sends out updates upon client's read request (which happens each time a client's local cache becomes too stale). The callback mechanism exploits the fact that computation threads often revisit the same parameters in iterative-convergent algorithms, and thus the server can push out table-rows to registered clients without clients' explicit request. Our server-push model causes more eager communication as specified in ESSP.

More specifically, on an individual client, as soon as all updates generated before and in iteration t are sent to server partitions and no more updates before or in that iteration can be generated (because all compute threads have advanced beyond that iteration), the client's communication threads send an client clock message to each server partition, indicating that "all updates generated before and in iteration t by this client have been made visible to this server partition". Note that this assumes reliable ordered message delivery, which is the case in most existing networks.

After a server partition sends out all dirty parameters modified in iteration t, it sends an server clock message to each client communication thread, indicating that 'all updates generated before and in iteration t in the parameter partition have been made visible to this client". Upon receiving such a clock message, the client communication thread updates the age of the corresponding parameters and permits the relevant blocked compute threads to proceed on reads, if any.

Fault Tolerance

Bösen provides fault tolerance by checkpointing the server model partitions; in the event of a failure, the entire system is restarted from the last checkpoint. A valid checkpoint contains the model state *strictly* right after iteration t — the model state includes all model updates generated before and during iteration t, and excludes all updates after the t-th `PS.Clock()` call by any worker thread. With bounded staleness, clients may asynchronously enter new iterations and begin sending updates; thus, whenever a checkpointing clock event is reached, each server model partition will copy-on-write to protect the checkpoint's parameter values until that checkpoint has been successfully copied externally. Since taking a checkpoint can be slow, a checkpoint will not be created for every iteration, or even every few iterations. A good estimate of the amount of time between taking checkpoints is $\sqrt{2T_sT_f/N}$ [148], where T_s is the mean time to save a checkpoint, N is the number of machines involved and T_f is the mean time to failure (MTTF) of a machine, typically estimated as the inverse of the average fraction of machines that fail each year.

As Bösen targets offline batch training, restarting the system (disrupting its availability) is not critical. With tens or hundreds of machines, such training tasks typically complete in hours or tens of hours. Considering the MTTF of modern hardware, it is not necessary to create many checkpoints and the probability of restarting is low. In contrast, a replication-based fault tolerance mechanism inevitably costs $2\times$ or even more memory on storing the replicas and additional network bandwidth for synchronizing them.

3.5 Evaluation

We show that ESSP improves the speed and quality of convergence (compared with SSP) for collapsed Gibbs sampling in topic model and stochastic gradient descent (SGD) in matrix factorization. Furthermore, ESSP is robust against the staleness setting, relieving the user from worrying about the additional tuning parameter s. Since (E)SSP subsumes both BSP and fully asynchronous consistency, we approximate these consistency with varying staleness level. As discussed in Section 3.2.4, we do not evaluate VAP model due to the difficulty in directly implementing VAP model.

3.5.1 Experiment Details

ML Models and algorithms: We use the sparsified collapsed Gibbs sampling in [147] for LDA topic modeling, and SGD for matrix factorization [77]). Unless stated otherwise, we use rank $K = 100$ and regularization parameter $\lambda = 0.1$. Both algorithms are implemented using Bösen's interface. We use log-likelihood as the measure of training quality for LDA, and the squared loss without the ℓ_2-penalized loss as the objective for MF. To speed up convergence, the step size for MF is chosen to be as large as possible while the algorithm still converges with staleness 0.

Datasets: For topic model: New York Times ($N = 100m$ tokens, $V = 100k$ vocabularies, and $K = 100$ topics). We use 50% of the dataset as the minibatch size in each `Clock()` call. For Matrix factorization: Netflix dataset ($480k$ by $18k$ matrix with $100m$ nonzeros.) We use 1% and 10% of the whole data as the minibatch size in each `Clock()` call.

Compute cluster: Matrix factorization experiments were run on 64 nodes, each with 2 cores and 16GB RAM, connected via 1Gbps ethernet. LDA experiments were run on 8 nodes, each with 64 cores and 128GB memory, connected via 1Gbps ethernet.

3.5.2 System Evaluations

Empirical Staleness. We offer a brief system evaluation to help understand the ensuing ML evaluations. To demonstrate how our ESSP implementation reduces the staleness of parameter read, we empirically measure the runtime staleness of parameter reads during an execution. Fig. 3.7 shows the distribution of parameter runtime staleness observed in matrix factorization run with SSP and ESSP communication protocols. We measure the runtime staleness using *clock differential*. When a worker reads a parameter, the read parameters reflect updates from other worker 0 or more clocks behind.[5] Clock differential is defined as this (non-positive) clock difference.

Under SSP, the distribution of clock differentials is nearly uniform, because SSP "waits until the last minute" to update the local parameter cache. On the other hand, ESSP initiates communication to updates the local parameter caches via its eager communication, which reduces the negative tail in clock differential distribution. This improved staleness profile is ESSP's most salient advantage over SSP, and is the foundation of the improved performance observed in subsequent evaluations.

---

[5] For simplicity, we do not consider reads containing updates from future clocks from other workers.

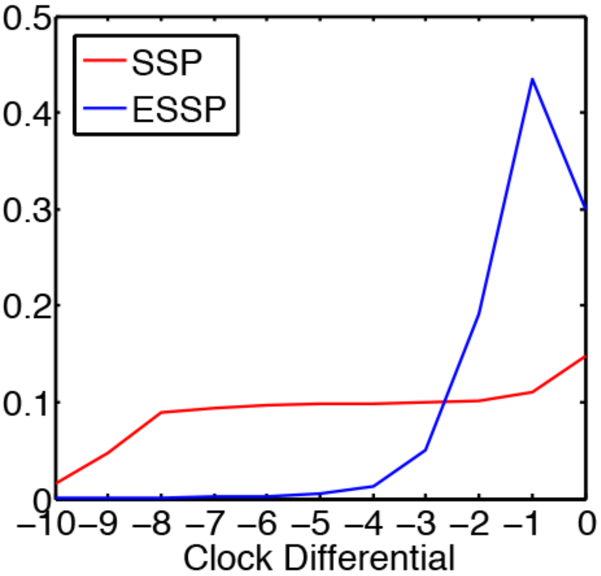


Figure 3.7: Empirical staleness distribution from matrix factorization. X-axis is (parameter age - local clock), *i.e.*, the clock differential. Y-axis is the distribution of the clock differentials observed in parameter reads. Note that in Bulk Synchronous Parallel (BSP) system such as MapReduce, the staleness is always -1. We use rank 100 for matrix factorization, and each clock is 1% minibatch (i.e., a minibatch corresponds to 1% of the dataset). The experiment is run on a 64 node cluster.

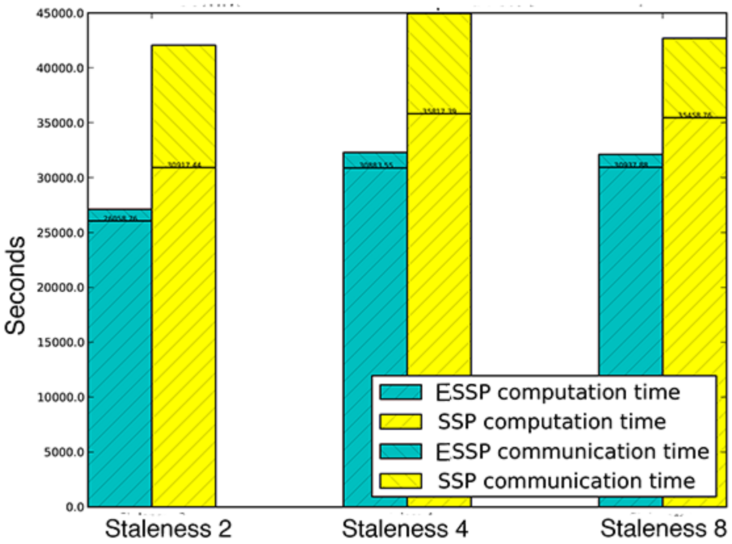


Figure 3.8: Communication and Computation breakdown for LDA for SSP and ESSP with staleness $s = 2, 4, 8$. The lower part of the bars are computation, and the upper part is communication.

Computation v.s. Communication. We now turn to analyze the time breakdown between computation and communication. We consider the breakdown from the worker perspective: the time a worker spent on the actual ML computation is the computation time, while the time a worker is spent on waiting for the communication is considered the communication time. Importantly, communication that takes place in the background (in parallel to the computation) is hidden in this accounting. This breakdown definition captures well the communication overheads encountered when going from a single worker implementation to a multi-worker one.

Fig. 3.8 shows the breakdown of communication and computation time for varying staleness for LDA to reach a pre-defined training quality. There are two important observations: (1) The computation time for ESSP is shorter than that for SSP for all considered staleness settings. This is because computation of ESSP utilizes fresher parameters (see Fig. 3.7) and thus the computation makes more progress per iteration and thus requires fewer iterations. (2) The communication time for ESSP is significantly lower than that for SSP under all considered settings. By sending updates preemptively, ESSP not only reduces the staleness but also reduces the chance of client workers being blocked to wait for the updates. Essentially, *ESSP is a more pipelined version of SSP*. Both factors—more effective computation and lower communication wait time—contribute to the overall convergence per wall clock time, as evident in Fig. 3.8 under all considered settings. We will revisit this overall reduction of wall clock time in the sequel.

3.5.3 ML Evaluation and Discussions

We now turn to the evaluation of ML training convergence, which is the crux of our theoretical analyses. Fig. 3.9 shows the objective over clock and wall clock time for LDA and matrix factorization, for a wide range of staleness values. Since clock measures the logical work, convergence per clock reveals how much algorithmic progress the computation makes in the same amount of work. This isolates the convergence properties from the system throughput issues. Convergence per wall clock time (in seconds), on the other hand, measures the ultimate goal of training: to reach the optimal solution as quickly as possible. This measure factors in both the algorithmic progress as well as the system throughput.

We do not consider test performance, which depends on many other factors such as the suitability of the specified model and the distribution drift between training and test data, all of which can compound the study of ML convergence. For MF due to the small minibatch sizes (1% and 10% of the whole dataset) we consider higher staleness ($s = 100, 10$) such that the program can miss updates for up to one full pass over the dataset.

We study the results from the following perspectives:

Convergence per clock: Convergence per clock shows the algorithmic progress per unit work of computation. We study the convergence per clock for LDA and MF (with minibatch sizes at 1% and 10% of the dataset) in the left column of Fig. 3.9. Under all studied scenarios, ESSP executions converge comparably to or faster than SSP counterparts. This speed-up is due to the reduced staleness as evident in the staleness profile (Fig. 3.7). This is also consistent with the preceding theoretical analyses that, under SSP, computation using fresher model parameters achieves faster convergence (see Theorem 3.4).

Convergence per wall clock time: Compared with the per clock convergence, convergence over wall clock time (Fig. 3.9 right column) factors in the system throughput, which "separates out" the curves in convergence per clock plots. For both SSP and ESSP, higher staleness incurs lower communication wait time, which improves system throughput up to certain staleness values. Furthermore, ESSP executions converge comparably to or faster than the SSP counterparts with respect to wall clock time. In particular, for LDA with staleness 8, ESSP reduces the time to reach objective function value -7.32×10^9 by 4.04x. As discussed earlier, there are primarily two sources for this speed-up: (1) Faster convergence per clock, and (2) higher system throughput due to the pipelining effect of ESSP that reduces stalling due to communication (Section 3.5.2). As a result, the gaps between ESSP and SSP convergence over wall clock time widen compared with the per clock convergence speed.

Robustness to Staleness: One challenge in applying SGD algorithms is the sensitivity to step size. Step sizes that are too small lead to slow convergence, while step sizes that are too large cause divergence. The problem of step size tuning is aggravated in the distributed settings, where staleness could aggregate the updates in a non-deterministic manner, causing unpredictable performance (dependent on network congestion and the machine speeds, for example). In the case of MF, SSP diverges under high staleness (such as $s = 5, 10$ for MF with 10% minibatch), as staleness effectively increases the step size. However, ESSP is robust across all investigated staleness values due to the concentrated staleness profiles (Fig. 3.7) that are insensitive to the actual staleness bound s. For some high SSP staleness, such as $s = 15$ for MF with 1% minibatch, the convergence is "shaky" due to the variance introduced by staleness. ESSP produces lower variance for all staleness settings, consistent with our theoretical analyses. This improvement largely removes the need for user to tune the staleness parameter introduced in SSP, making algorithms much more robust under SSP execution.

In summary, our analyses and experiments show that ESSP combines the strengths of VAP and SSP: (1) ESSP achieves strong theoretical properties comparable to VAP; (2) ESSP can be efficiently implemented, with excellent empirical performance on two ML applications: matrix completion using SGD, and topic modeling using sampling. We also show that ESSP achieves higher throughput than SSP, thanks to system optimizations exploiting ESSP's aggressive scheduling.

3.6 Additional Related Work

Existing software that is tailored towards *distributed* (rather than merely single-machine parallel), scalable ML can be roughly grouped into two categories: general-purpose, programmable libraries or frameworks such as GraphLab [101] and Parameter Servers (PSes) [65, 93], or special-purpose solvers tailored to specific categories of ML applications: CCD++ [151] for matrix factorization, Vowpal Wabbit for regression/classification problems via stochastic optimization [85], and Yahoo LDA as well as Google plda for topic modeling [142].

As discussed in Chapter 1, the primary differences between the general-purpose frameworks (including this work) and the special-purpose solvers are: (1) The former are user-programmable and can be extended to handle arbitrary ML applications, while the latter are non-programmable and restricted to predefined ML applications; (2) Because the former must support arbitrary ML programs, their focus is on improving the "systems" code (notably, communication and synchronization protocols) to increase the efficiency of *all ML algorithms*, particularly through the careful design of consistency models (e.g., the graph consistency in GraphLab and the iteration/value-bounded consistency in PSes) — in contrast, the special-purpose systems combine both system code improvements and *algorithmic* (i.e. mathematical) improvements tailor-made for their specific category of ML applications.

In [93], the authors propose and implement a PS consistency model that has similar theoretical guarantees to the ideal VAP model presented in this chapter. However, we note that their implementation does not strictly enforce the conditions of their consistency model. Their theoretical analyses assume zero latency for transmission over network in the implementation, while in a real cluster, there could be arbitrarily long network delay. In their system, reads do not wait for delayed updates, so a worker may compute with highly inconsistent parameters in the case of congested network.

On the wider subject of Big Data, Hadoop [8] and Spark [154] are popular programming frameworks which ML applications have been developed on. To our knowledge, there is no work showing that Hadoop or Spark have superior ML algorithm performance compared to frameworks designed for ML like GraphLab and PSes (let alone the special-purpose solvers). The main difference is that Hadoop/Spark only feature strict consistency, and do not support flexible consistency models like graph- or bounded-consistency; but Hadoop and Spark ensure program portability, reliability and fault tolerance at a level that GraphLab and PSes have yet to match.

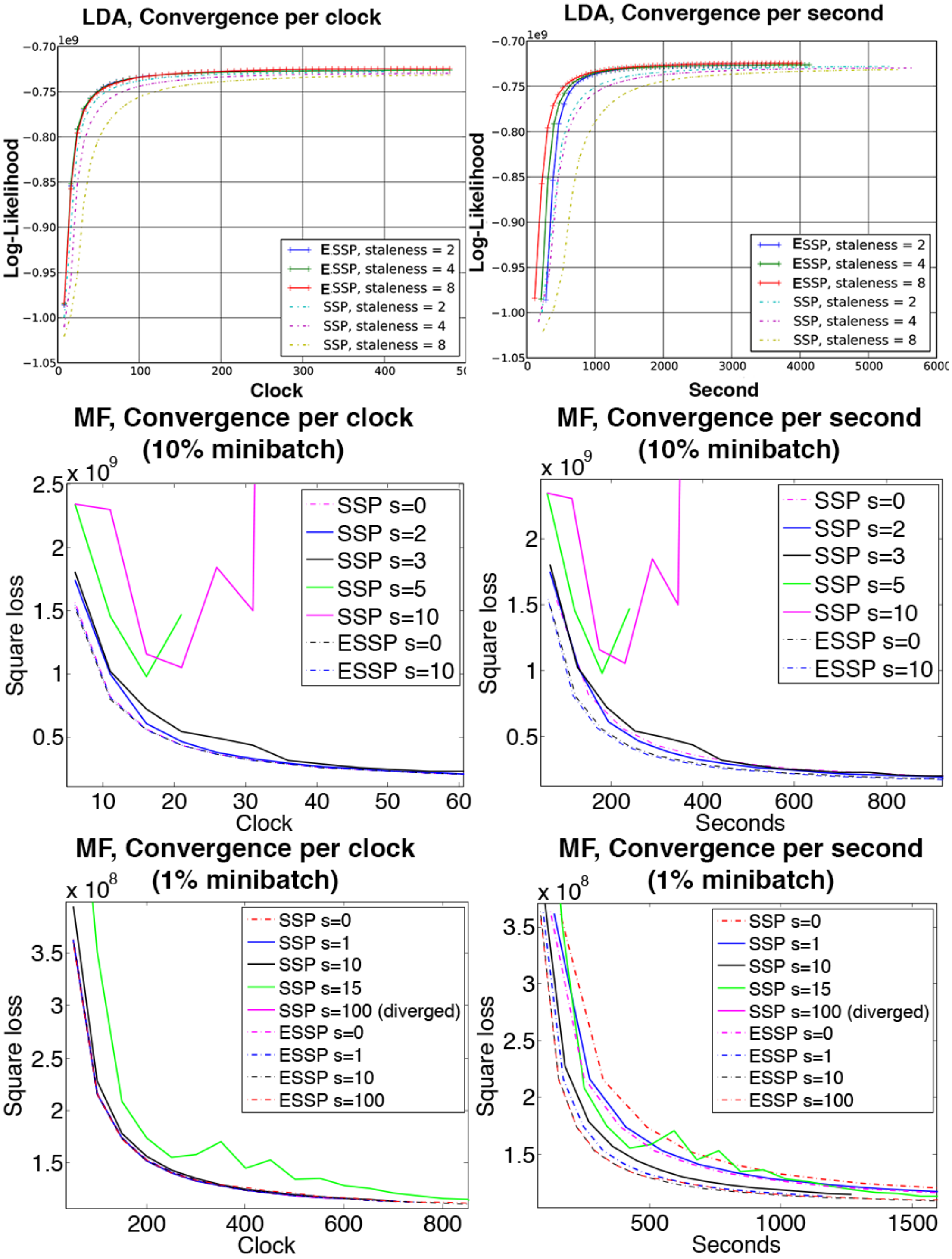


Figure 3.9: **ML Convergence.** The convergence speed per iteration and per second for LDA and MF. The y-axes are the training objectives. In the case of LDA the training objective is log-likelihood, for which higher is better. In the case of MF the training objective is square loss (the regularization loss is negligible compared with square loss), for which lower is better. In certain cases for MF the training objective diverges (i.e., fails to converge), such as SSP $s = 5$ with 10% minibatch. In those cases the convergence curves are truncated at the clock that divergence occurs.

Chapter 4

Model Parallel Learning with Staleness

In previous chapters we have explored data parallel algorithms in which the dataset is partitioned to concurrently executing workers who read and update the entire model parameters. They do not address the high dimensional problem that may incur substantial memory and network overheads in storing and transmitting the undivided model to distributed workers.

In this chapter, we consider an alternative approach to dividing a large ML problem in stale settings. Instead of dividing the data, we partition the model parameters so that each worker updates and maintain a subset of model coordinates. In this way, we only need to transmit $O(D)$ amount of data, where D is the size of dataset.[1] Our framework extends the proximal gradient descent into distributed model parallel setting, and covers many important problems such as Lasso, (ℓ_1-penalized) logistic regression, support vector machines, among others. Importantly, our results do not assume convexity on the model (both smooth loss and the non-smooth regularizer), and thus applies to non-convex problems such as Group Lasso. We support our theoretical analyses with large-scale experiments on a 100 node cluster with the Lasso application.

4.1 Introduction

Many machine learning and statistics problems fit into the general composite minimization framework:

$$\min_{\mathbf{x}\in\mathbb{R}^d} \ \tfrac{1}{n}\textstyle\sum_{i=1}^n f_i(\mathbf{x}) + g(\mathbf{x}), \tag{4.1}$$

where the first term is typically a smooth empirical risk over n training samples and the second term g is a nonsmooth regularizer that promotes structures. Popular examples under this framework include:

- Lasso: least squares loss $f_i(\mathbf{x}) = (y_i - \mathbf{a}_i^\top \mathbf{x})^2$ and ℓ_1 norm regularizer $g(\mathbf{x}) = \|\mathbf{x}\|_1$;

---

[1] For problems such as biology dataset, D (e.g., the number of patients) is usually much smaller than the model dimension (e.g., billions of gene interaction features).

- Logistic regression: logistic loss $f_i = \log(1 + \exp(-y_i \mathbf{a}_i^\top \mathbf{x}_i))$;
- Boosting: exponential loss $f_i(\mathbf{x}) = \exp(-y_i \mathbf{a}_i^\top \mathbf{x})$;
- Support vector machines: hinge loss $f_i(\mathbf{x}) = \max\{0, 1 - y_i \mathbf{a}_i^\top \mathbf{x}\}$ and (squared) ℓ_2 norm regularizer $g(\mathbf{x}) = \|\mathbf{x}\|_2^2$.

Over the years there is also a rising interest in using nonconvex losses (mainly for robustness to outliers) and nonconvex regularizers (mainly for smaller bias in feature selection and support estimation), see e.g. [45, 104, 129, 144, 152, 156, 157, 162].

Due to the apparent importance of the composite minimization framework and the rapidly growing size in both model dimension and sample volume, there is a strong need to develop a practical *parallel* system that can solve the problem in (4.1) efficiently and in a scale that is impossible for a single machine [4, 8, 19, 48, 65, 94, 100, 154]. Existing systems can roughly be divided into three categories: bulk synchronous [8, 140, 154], (totally) asynchronous [19, 100], and partially asynchronous (also called stale synchronous in this work) [4, 19, 48, 65, 94, 137]. The bulk synchronous parallel mechanism (BSP) forces synchronization barriers so that the worker machines can stay on the same page to ensure correctness. However, in a real deployed parallel system BSP usually suffers from the straggler problem, that is, the performance of the whole system is bottlenecked by the *slowest* worker machine. On the other hand, asynchronous systems achieve much greater throughputs, although at the expense of potentially losing the correctness of the algorithm. The stale synchronous parallel (SSP) mechanism is a compromise between the previous two mechanisms: it allows the worker machines to operate asynchronously, as long as they are not too far apart. SSP is particularly suitable for machine learning applications, where iterative algorithms robust to small errors are usually used to find an appropriate model. This view is also practiced by many recent works building on the SSP mechanism [4, 40, 48, 65, 94, 96, 120].

Existing parallel systems can also be divided into data parallel and model parallel. In the former case, one usually distributes the computation involving each component function f_i in (4.1) into different worker machines. This is suitable when $n \gg d$, i.e. large data volume but moderate model size. A popular algorithm for this case is the stochastic gradient algorithm and its proximal versions [4, 48, 65, 94], under the SSP mechanism. In contrast, model parallel refers to the regime where $d \gg n$, i.e. large model size but moderate data volume. This is the case for many computational biology and health care problems, where collecting many samples can be very expensive but for each sample we can relatively cheaply take a large number of measurements (features). As a result we need to partition the model $\mathbf{x}$ into different (disjoint) blocks and distribute them among many worker machines. The proximal gradient [25, 53] or its accelerated version [14] is again a natural candidate algorithm due to its nice ability of handling nonsmooth regularizers. However, such proximal gradient algorithm has not been investigated under SSP mechanism for model parallelism, although some other types of asynchronous algorithms are studied before (see Section 4.7). The main goal of this work is to fill in this important gap.

More specifically, we make the following contributions: 1). We propose msPG, an extension of the proximal gradient algorithm to the new model parallel and stale synchronous setting. 2). We provide a rigorous analysis of the convergence properties of msPG. Under a very general condition that allows both *nonsmooth* and *nonconvex* functions we prove in Theorem 4.1 that any limit point of the sequence generated by msPG is a critical point. Then, inspired by the recent Kurdyka-Łojasiewicz (KŁ) inequality [10, 12, 23, 25, 80], we further prove in Theorem 4.2 that the whole sequence of msPG in fact converges to a critical point, under mild technical assumptions that we verify for many familiar examples. Lastly, relating msPG to recent works on inexact proximal gradient (on a single machine), we provide, under the new model parallel and SSP setting, a simple proof of the usual sublinear $O(1/t)$ rate of convergence (assuming convexity). We remark our technical contributions with comparison to related work after each main results. 3). Building on the recent parameter server framework [65, 94], we give an economical implementation of msPG that completely avoids storing local full models in each worker machine. The resulting implementation only requires storing the partitioned data (with size $O(nd_i)$ for d_i assigned parameters) and communicating a vector of length n in each iteration. 4). We corroborate our theoretical findings with controlled numerical experiments.

4.2 Preliminaries

We collect here some useful definitions that will be needed in our later analysis.

Since we consider a proper and closed[2] function $h : \mathbb{R}^d \to (-\infty, +\infty]$ that may *not* be smooth or convex, we need a generalized notion of "derivative".

Definition 4.1 (Subdifferential and critical point, [126])**.** *The Frechét subdifferential* $\hat{\partial} h$ *of* h *at* $\mathbf{x} \in \operatorname{dom} h$ *is the set of* $\mathbf{u}$ *such that*

$$\liminf_{\mathbf{z} \neq \mathbf{x}, \mathbf{z} \to \mathbf{x}} \tfrac{h(\mathbf{z}) - h(\mathbf{x}) - \mathbf{u}^\top (\mathbf{z} - \mathbf{x})}{\|\mathbf{z} - \mathbf{x}\|} \geq 0, \tag{4.2}$$

while the (limiting) subdifferential ∂h *at* $\mathbf{x} \in \operatorname{dom} h$ *is the graphical closure of* $\hat{\partial} h$*:*

$$\{\mathbf{u} : \exists \mathbf{x}^k \to \mathbf{x}, h(\mathbf{x}^k) \to h(\mathbf{x}), \mathbf{u}^k \in \hat{\partial} h(\mathbf{x}^k) \to \mathbf{u}\}. \tag{4.3}$$

The critical points of h *are* $\operatorname{crit} h := \{\mathbf{x} : \mathbf{0} \in \partial h(\mathbf{x})\}$.

Pleasantly, when h is continuously differentiable or convex, the subdifferential ∂h and critical points $\operatorname{crit} h$ coincide with the usual notions.

Definition 4.2 (Distance and projection)**.** *The distance function to a closed set* $\Omega \subseteq \mathbb{R}^d$ *is defined as:*

$$\operatorname{dist}_\Omega(\mathbf{x}) := \min_{\mathbf{y} \in \Omega} \|\mathbf{y} - \mathbf{x}\|, \tag{4.4}$$

and the metric projection onto Ω *is:*

$$\operatorname{proj}_\Omega(\mathbf{x}) := \operatorname{argmin}_{\mathbf{y} \in \Omega} \|\mathbf{y} - \mathbf{x}\|. \tag{4.5}$$

---

[2] An extended real-valued function h is proper if its domain $\operatorname{dom} h := \{\mathbf{x} : h(\mathbf{x}) < \infty\}$ is nonempty; it is closed iff its sublevel sets $\{\mathbf{x} : h(\mathbf{x}) \leq \alpha\}$ is closed for all $\alpha \in \mathbb{R}$.

Note that $\operatorname{proj}_\Omega$ is always a singleton iff Ω is convex.

Definition 4.3 (Proximal map, e.g. [126])**.** *A natural generalization of the metric projection using a closed and proper function h is (with parameter $\eta > 0$):*

$$\operatorname{prox}_h^\eta(\mathbf{x}) := \operatorname{argmin}_{\mathbf{z}} h(\mathbf{z}) + \tfrac{1}{2\eta}\|\mathbf{z} - \mathbf{x}\|^2, \tag{4.6}$$

where $\|\cdot\|$ is the usual Euclidean norm.

If h decreases slower than a quadratic function (in particular, when h is bounded below), its proximal map is well-defined for all (small) η. For convex h, the proximal map is always a singleton while for nonconvex h, the proximal map can be set-valued. In the latter case we also abuse the notation $\operatorname{prox}_h^\eta(\mathbf{x})$ for an arbitrary element from that set. The proximal map is the key component of the popular proximal gradient algorithms [14, 25, 53].

Definition 4.4 (KŁ function, [24, 80])**.** *A function h is called KŁ if for all $\bar{\mathbf{x}} \in \operatorname{dom} \partial h$ there exist $\lambda > 0$ and a neighborhood X of $\bar{\mathbf{x}}$ such that for all $x \in X \cap [\mathbf{x} : h(\bar{\mathbf{x}}) < h(\mathbf{x}) < h(\bar{\mathbf{x}}) + \lambda]$ the following inequality holds*

$$\varphi'\left(h(\mathbf{x}) - h(\bar{\mathbf{x}})\right) \cdot \operatorname{dist}_{\partial h(\mathbf{x})}(\mathbf{0}) \geq 1, \tag{4.7}$$

for some desingularizing function $\varphi : [0, \lambda) \to \mathbb{R}_+, 0 \mapsto 0$, is continuous, concave, and has continuous and positive derivative φ' on $(0, \lambda)$.

Remark 4.1. *For many functions one can have $\varphi(s) := s^{1-\theta}$, $\theta \in (0, 1]$. The KŁ inequality (Eq. (4.7)) then becomes $\operatorname{dist}_{\partial h(\mathbf{x})}(\mathbf{0}) \geq (1 - \theta)^{-1}|h(\mathbf{x}) - h(\bar{\mathbf{x}})|^\theta$. For strongly convex function h that is differentiable, we can have $\theta = \frac{1}{2}$ and recovers the usual $\operatorname{dist}_{\partial h(\mathbf{x})}(\mathbf{0})^2 = ||\nabla h(\mathbf{x})||^2 \geq 2|h(\mathbf{x}) - h(\bar{\mathbf{x}})|$, where $\bar{\mathbf{x}}$ is the global minimizer.*

The KŁ functions ensures that one can reparameterize the range of the function such that the resulting function has a kink in the critical points $\operatorname{crit} h$ and is steep around $\operatorname{crit} h$. This characterization of functions precludes pathological cases such as indefinite oscillation generating descent paths of infinite length in descent algorithms. The KL inequality (4.7) is also an important tool to bound the trajectory length of a dynamical system (see [24, 80] and the references therein for some historic developments). It has recently been used to analyze discrete-time algorithms in [3] and proximal algorithms in [10, 11, 25]. Quite conveniently, most practical functions, in particular, "definable" functions and convex functions under certain growth condition, are KŁ.

For a more detailed discussion of KŁ functions, including many familiar examples, see [25, Section 5] and [11, Section 4].

4.3 Problem Formulation

We consider the composite minimization problem:

$$\min_{\mathbf{x} \in \mathbb{R}^d} F(\mathbf{x}), \text{ where } F(\mathbf{x}) = f(\mathbf{x}) + g(\mathbf{x}). \tag{P}$$

Usually f is a smooth loss function and g is a regularizer that promotes structure. We consider the **model parallel** scenario, that is, we decompose the d model parameters into p disjoint groups, and designate one worker machine for each group. Formally, consider the decomposition $\mathbb{R}^d = \mathbb{R}^{d_1} \times \mathbb{R}^{d_2} \times \cdots \times \mathbb{R}^{d_p}$, $\mathbf{x} = (x_1, x_2, \ldots, x_p)$, and let $\nabla_i f : \mathbb{R}^d \to \mathbb{R}^{d_i}$ be the partial gradient of f on the i-th factor space (machine). Clearly, $x_i, \nabla_i f(\mathbf{x}) \in \mathbb{R}^{d_i}$ and $\sum_{i=1}^p d_i = d$. The i-th machine is responsible for the i-th factor $x_i \in \mathbb{R}^{d_i}$, however, we also allow machine i to keep a local copy $\mathbf{x}^i \in \mathbb{R}^d$ of the *full* model parameter. This is for the convenience of evaluating the partial gradient $\nabla_i f : \mathbb{R}^d \to \mathbb{R}^{d_i}$, and we will discuss in Section 4.5 how to implement this in an economical way. Note that unlike the **data parallel** setting, we do *not* consider explicitly distributing the computation of the gradient $\nabla_i f$.

We extend the proximal gradient algorithm [25, 53] to solve the composite problem (P) under the new model parallel setting, and we require the following standard assumptions for our convergence analysis.

Assumption 4.1. *Regarding the functions f, g in* (P)*:*

1. *They are bounded from below;*
2. *The function f is differentiable and the gradients ∇f, $\nabla_i f$ are Lipschitz continuous with constant L_f and L_i, respectively. Set $L = \sum_{i=1}^p L_i$;*
3. *The function g is closed, and separable, i.e., $g(\mathbf{x}) = \sum_{i=1}^p g_i(x_i)$.*

The first assumption simply allows us to have a finite minimum value and is usually satisfied in practice. The second assumption (smoothness) is critical in two aspects: (1) It allows us to upper bound f by its quadratic expansion at the current iterate—a standard step in the convergence proof of gradient type algorithms:

$$\forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^d, \; f(\mathbf{x}) \le f(\mathbf{y}) + \langle \mathbf{x} - \mathbf{y}, \nabla f(\mathbf{y}) \rangle + \tfrac{L}{2} \|\mathbf{x} - \mathbf{y}\|^2. \tag{4.8}$$

(2) It allows us to bound the inconsistencies in different machines due to asynchronous updates. The separable assumption is what makes model parallelism interesting and feasible. We remark that both the second assumption (smoothness) and the third assumption (separability) can be relaxed using techniques in [15] and [152], respectively. For brevity we do not pursue these extensions here. Note that we do *not* assume convexity on either f or g, and g need not even be continuous.

The separability assumption above on g implies that

$$\operatorname{prox}_g^\eta(\mathbf{x}) = \big(\operatorname{prox}_{g_1}^\eta(x_1), \; \ldots \;, \; \operatorname{prox}_{g_p}^\eta(x_p)\big). \tag{4.9}$$

Let us introduce the update operator (on machine i):

$$U_i(\mathbf{x}^i) = U_i(\mathbf{x}^i, x_i) := \operatorname{prox}_{g_i}^\eta(x_i - \eta \nabla_i f(\mathbf{x}^i)) - x_i, \tag{4.10}$$

i.e. machine i computes the i-th part of the gradient using its local model $\mathbf{x}^i$, updates its parameter x_i in charge using step size η, and finally applies the proximal map of the component function g_i. In a real large scale parallel system, the communication delay among machines and the unexpected shut down of machines are practical issues that bottlenecks the performance of the system, and hence a more relaxed synchronization protocol than full synchronization is needed. Consider a global clock shared by all machines and denote T_i the set of active clocks when

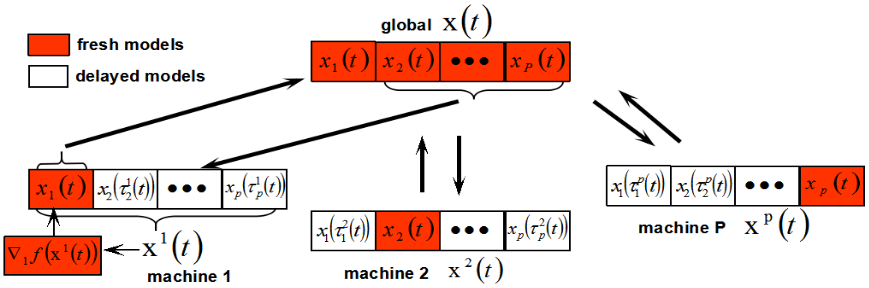

fresh models
delayed models
global $\mathrm{x}(t)$
$x_1(t)$ $x_2(t)$ ••• $x_p(t)$
$x_1(t)$ $x_2(\tau_2^1(t))$ ••• $x_p(\tau_p^1(t))$
$\nabla_1 f(\mathrm{x}^1(t))$
$\mathrm{x}^1(t)$
machine 1
$x_1(\tau_1^2(t))$ $x_2(t)$ ••• $x_p(\tau_p^2(t))$
machine 2 $\mathrm{x}^2(t)$
$x_1(\tau_1^p(t))$ $x_2(\tau_2^p(t))$ ••• $x_p(t)$
machine P $\mathrm{x}^{\mathrm{P}}(t)$

Figure 4.1: The algorithm msPG under model parallelism and stale synchronism. Machine i keeps a local model $\mathbf{x}^i(t)$ that contains stale parameters of other machines (due to communication delay and network latency). These local models are used to compute the partial gradient $\nabla_i f(\mathbf{x}^i)$ which is then used to update the parameters $x_i(t)$ in each machine. See Section 4.5 for an economical implementation of msPG.

machine i computes an update, and $\mathbb{I}_{\{t \in T_i\}}$ as the indicator function of the event $t \in T_i$. Formally, the t-th iteration on machine i can be written as:

$$\text{msPG} \begin{cases} \quad \forall i, \ x_i(t+1) = x_i(t) + \mathbb{I}_{\{t \in T_i\}} U_i(\mathbf{x}^i(t)), \\ \text{(local)} \quad \mathbf{x}^i(t) = \big(x_1(\tau_1^i(t)), \ \ldots \ , \ x_p(\tau_p^i(t))\big), \\ \text{(global)} \quad \mathbf{x}(t) = \big(x_1(t), \ \ldots \ , \ x_p(t)\big), \end{cases}$$

That is, machine i only performs its update operator at its active clocks. The local full model $\mathbf{x}^i(t)$ assembles all components from other machines, and is possibly a delayed version of the global model $\mathbf{x}(t)$, which assembles the most up-to-date component in each machine. Note that the global model is introduced for our analysis, and is not accessible in a real implementation. More specifically, $\tau_j^i(t) \leq t$ models the communication delay among machines: when machine i conducts its t-th update it only has access to $x_j(\tau_j^i(t))$, a delayed version of the component $x_j(t)$ on the j-th machine. We will refer to the above updates as msPG (for **m**odel parallel, **s**tale synchronous, **P**roximal **G**radient). Figure 4.1 illustrates the main idea of msPG.

In a practical distributed system, communication among machines is much slower than local computations, and the performance of a *synchronous* system is often bottlenecked at the *slowest* machine, due to the need of synchronization in every step. The delays $\tau_j^i(t)$ and active clocks T_i that we introduced in msPG aim to address such issues.

To establish convergence for msPG we obviously need some control over the delay $\tau_j^i(t)$ and the active clocks T_i, for otherwise some machines may not make progress at all. For our convergence analyses, we need the following assumptions:

Assumption 4.2. *The delay and skip frequency satisfy:*

1. $\forall i, \forall j, \forall t, \ 0 \leq t - \tau_j^i(t) \leq s$*;*
2. $\forall i, \forall t, \ \tau_i^i(t) = t$*;*
3. $\forall i, \forall t, \ T_i \cap \{t, t+1, \cdots, t+s\} \neq \emptyset$*.*

Intuitively, the first assumption guarantees that the information machine i gathered from other machines at the t-th iteration are not too obsolete (bounded by at most s clocks apart). The second assumption ($\tau_i^i(t) \equiv t$) is natural since the i-th worker machine is maintaining x_i hence would always have the latest copy. The third assumption requires each machine to update at least once in every $s+1$ iterations. We remark that these assumptions are very natural and have been widely adopted in previous works [4, 19, 48, 65, 92, 96, 137]. They are also in some sense unavoidable: one can construct instances such that msPG do not converge if these assumptions are violated. Clearly, when $s = 0$ (no delay) our framework reduces to the bulk synchronous proximal gradient algorithm.

4.4 Convergence Analysis

In this section, we conduct detailed analysis of the model parallel stale synchronous proximal gradient algorithm msPG. Our first result is as follows:

Theorem 4.1 (Asymptotic consistency)**.** *Let Assumption 4.1 and 4.2 hold, and apply msPG to problem* (P). *If the step size $\eta < (L_f + 2Ls)^{-1}$, then the global model and local models satisfy:*

1. $\sum_{t=0}^{\infty} \|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2 < \infty$;
2. $\lim\limits_{t\to\infty} \|\mathbf{x}(t+1) - \mathbf{x}(t)\| = 0$, $\lim\limits_{t\to\infty} \|\mathbf{x}(t) - \mathbf{x}^i(t)\| = 0$;
3. *The limit points* $\omega(\{\mathbf{x}(t)\}) = \omega(\{\mathbf{x}^i(t)\}) \subseteq \operatorname{crit} F$.

Remark 4.2. *Our bound on the step size η is natural: If $s = 0$, i.e., there is no asynchronism then we recover the standard step size rule $\eta < 1/L_f$ (we can increase η by another factor of 2, had convexity on g been assumed). As staleness s increases, we need a smaller step size to "damp" the system to still ensure convergence. The factor $L := \sum_{i=0}^{p} L_i$ is a measurement of the degree of "dependency" among worker machines.*

The proof is non-trivial and can be found in Section C.1. The first assertion of the above theorem states that the global sequence $\mathbf{x}(t)$ has square summable successive differences, while the second assertion implies that both the successive difference of the global sequence and the inconsistency between the local sequences and the global sequence diminish as the number of iterations grows. These two conclusions provide a preliminary stability guarantee for msPG. The third assertion further justifies msPG by showing that, without convexity assumption on either f of g, any limit point it produces is necessarily a critical point. Of course, when F is convex, any critical point is globally optimal.

The closest result to Theorem 4.1 we are aware of is [19, Proposition 7.5.3], where essentially the same conclusion was reached but under the much more restrictive assumption that g is an indicator function of a product *convex* set. In contrast, our result allows g to be a convex function such as the ℓ_1 norm that is widely used to promote sparsity. Furthermore, we allow g to be any closed separable function (convex or not), covering the many recent nonconvex regularization functions in machine learning and statistics (see e.g. [45, 104, 156, 157]). We also note that the proof of Theorem 4.1 (for nonconvex g) involves significantly new ideas beyond those of [19].

We note that the existence of limit points can be guaranteed, for instance, if $\{\mathbf{x}(t)\}$ is bounded or the sublevel set $\{\mathbf{x} \mid F(\mathbf{x}) \leq \alpha\}$ is bounded for all $\alpha \in \mathbb{R}$. However, we have yet to prove that the sequence $\{\mathbf{x}(t)\}$ generated by msPG does converge to one of the critical points. In fact, in the model parallel setting with delays and skips, it is already a nontrivial task to argue that the objective values $\{F(\mathbf{x}(t))\}$ do not diverge to infinity. This is in sharp contrast with the bulk synchronous setting where it is trivial to guarantee the objective values to decrease (by using a sufficiently small step size). This is where we need some further assumptions.

Assumption 4.3 (Sufficient Decrease)**.** *There exists $\alpha > 0$ such that the global model $\mathbf{x}(t)$ generated by* msPG *(for problem* (P)*) satisfies: for all large t,*

$$F(\mathbf{x}(t+1)) \le F(\mathbf{x}(t)) - \alpha\|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2. \tag{4.11}$$

The sufficient decrease assumption is automatically satisfied in many descent algorithms, e.g., the proximal gradient algorithm. However, in the stale synchronous parallel setting, it is highly nontrivial to satisfy the sufficient decrease assumption because of the complication due to communication delays and update skips. Note also that none of the worker machines actually has access to the global sequence $\mathbf{x}(t)$, so even verifying the sufficient decrease property is not trivial. To simplify the presentation, we first analyze the performance of msPG using the KŁ inequality and taking the sufficient decrease property for granted, and later we we will give some verifiable conditions to justify this simplification.

Assumption 4.4. *F is a KŁ function.*

As we mentioned in the end of Section 4.2, most practical functions (including all functions in this thesis) are KŁ. Hence, this is a very mild assumption. Armed with these additional assumptions, we can strengthen the convergence properties in Theorem 4.1 for msPG:

Theorem 4.2 (Finite Length)**.** *Let Assumption 4.1, 4.2, 4.3 and 4.4 hold, and apply* msPG *to problem* (P)*. If the step size $\eta < (L_f + 2Ls)^{-1}$ and $\{\mathbf{x}(t)\}$ is bounded, then*

$$\textstyle\sum_{t=0}^{\infty} \|\mathbf{x}(t+1) - \mathbf{x}(t)\| < \infty, \tag{4.12}$$

$$\forall i = 1, \ldots, p, \ \textstyle\sum_{t=0}^{\infty} \|\mathbf{x}^i(t+1) - \mathbf{x}^i(t)\| < \infty. \tag{4.13}$$

Furthermore, $\{\mathbf{x}(t)\}$ and $\{\mathbf{x}^i(t)\}, i = 1, ..., p$, converge to the same critical point of F.

The proof is in Section C.2. Compared with the first assertion in Theorem 4.1, we now have the successive differences to be absolutely summable (instead of square summable). The former property is usually called finite length in dynamical systems. It is a significantly stronger property as it immediately implies that the whole sequence is Cauchy hence convergent, whereas we cannot get the same conclusion from the square summable property in Theorem 4.1.[3] We note that local maxima are excluded from being the limit in Theorem 4.2, thanks to Assumption 4.3. Also, the boundedness assumption on $\{\mathbf{x}(t)\}$ is easy to satisfy, for instance, when F has bounded sublevel sets. We refer to [11, Remark 3.3] for more conditions that guarantee the boundedness. Needless to say, if F is convex, then the whole sequence in Theorem 4.2 converges to a global minimizer.

We now provide some justifications on Assumption 4.3. For simplicity we assume all worker machines perform updates in each time step t:

---

[3] A simple example would be the sequence $x(t) = \sum_{k=1}^{t} \frac{1}{k}$, whose successive difference is square summable but clearly $x(t)$ does not converge. Consequently, $x(t)$ is not absolutely summable.

Assumption 4.5. $\forall i = 1, \cdots, p, \forall t, t \in T_i$.

Note that Assumption 4.5 is commonly adopted in the analysis of many recent parallel systems [4, 40, 48, 65, 94, 96, 120].

We will replace the sufficient decrease property in Assumption 4.3 with the following key property that turns out to be easier to verify:

Assumption 4.6 (Proximal Lipschitz)**.** *The update operators $U_i, i = 1, ..., p$ are eventually Lipschitz continuous, i.e., for all large t and small learning rate $\eta > 0$:*

$$\|U_i(\mathbf{x}^i(t+1)) - U_i(\mathbf{x}^i(t))\| \leq C_i \eta \|\mathbf{x}^i(t+1) - \mathbf{x}^i(t)\|, \tag{4.14}$$

where $C_i \geq L_i, i = 1, ..., p$, are positive constants.

The proximal Lipschitz assumption is motivated by the special case where $g \equiv 0$ and hence $\Delta_\eta(\mathbf{x}) = -\eta \nabla f(\mathbf{x})$ is η-Lipschitz, thanks to Assumption 4.1.2. As we have seen in previous sections, Lipschitz continuity plays a crucial role in our proof where a major difficulty is to control the inconsistencies among different worker machines due to communication delays. Similarly here, the proximal Lipschitz property, as we show next, allows us to remove the sufficient decrease property in Assumption 4.3—the seemingly strong assumption that we needed in proving our main result Theorem 4.2.

Equipped with this assumption, we can now justify Assumption 4.3. (Proof is in Section C.3.) In the sequel, we denote $C = \sum_{i=1}^{p} C_i \geq L$.

Lemma 4.1. *Assume $\forall t, i, t \in T_i$. Let the step size $\eta < \frac{\rho-1}{4C\rho} \frac{\sqrt{\rho}-1}{\sqrt{\rho}^{s+1}-1}$ for any $\rho > 1$ and all $U_i, i = 1, ..., p$ be proximal-Lipschitz continuous, then the sequences $\{\mathbf{x}(t)\}$ and $\{\mathbf{x}^i(t)\}, i = 1, ..., p$, have finite length.*

Hence it is sufficient to further characterize Assumption 4.6, which turns out to be a mild condition. Instead of giving a very technical justification, we give here some popular examples where Assumption 4.6 holds (proof in Section C.4). Some of these will also be tested in our experiments.

Example 4.1. *Assume $\forall t, i, t \in T_i$, then Assumption 4.6 holds for the following cases (modulo a technical condition on the 1-norm):*

- *$g \equiv 0$ (no regularization), $g = \|\cdot\|_0$ (nonconvex 0-norm), $g = \|\cdot\|_1$ (1-norm), $g = \|\cdot\|^2$ (squared 2-norm);*
- *elastic net $g = \|\cdot\|_1 + \|\cdot\|^2$ and its nonconvex variation $g = \|\cdot\|_0 + \|\cdot\|^2$;*
- *non-overlapping group norms $g = \|\cdot\|_{0,2}$ and $g = \|\cdot\|_{0,2} + \|\cdot\|^2$.*

Further for this non-skip case ($\forall t, i, t \in T_i$), msPG can be cast as an inexact proximal gradient algorithm (IPGA), which, together with the finite length property in Theorem 4.2, provide new insights on the nature of staleness in real parallel systems. It also allows us to easily obtain the usual $O(1/t)$ rate of convergence of the objective value.

Let us introduce an error term $\mathbf{e}(t) = \big(e_1(t), \ \ldots \ , e_p(t)\big)$, with which we can rewrite the global sequence of msPG as:

$$\forall i, \ x_i(t+1) = \operatorname{prox}_{g_i}^{\eta} \left(x_i(t) - \eta \nabla_i f(\mathbf{x}(t)) + e_i(t)\right), \tag{4.15}$$

where $e_i(t) = \eta[\nabla_i f(\mathbf{x}(t)) - \nabla_i f(\mathbf{x}^i(t))]$. This alternative representation of msPG falls under the IPGA in [127], where $\mathbf{e}(t)$ is the (gradient) error at iteration t. Note, however, that the error $\mathbf{e}(t)$ is not caused by computation but by communication delay and network latency, which only presents itself in a real stale synchronous parallel system. For convex functions F, [127] showed that the convergence rate of the objective value of IPGA can be controlled by the summability of the error magnitude $\|\mathbf{e}(t)\|$. Interestingly, our next result proves that the finite length property in Theorem 4.2 immediately implies the summability of the errors $\|\mathbf{e}\|$, even for nonconvex functions f and g. Moreover, for convex F, this also leads to the usual $O(1/t)$ rate of convergence in terms of the objective value.

Theorem 4.3 (Global rate of convergence)**.** *If the finite length property in Theorem 4.2 holds, then*

1. $\sum_{t=0}^{\infty} \|\mathbf{e}(t)\| < \infty$*;*
2. $F(\frac{1}{t}\sum_{k=1}^{t} \mathbf{x}(k)) - \inf F \leq O\left(t^{-1}\right)$.

The proof is in Section C.5. Intuitively, if the error $\mathbf{e}(t)$ decreases (slightly) faster than $O(1/t)$, then the rate of convergence of msPG is not affected even under the model parallel and stale synchronous setting (provided F is convex). To the best of our knowledge, this is the first *deterministic* rate of convergence result in the model parallel and stale synchronous setting.

4.5 Economical Implementation

In this section, we show how to economically implement msPG for the widely used linear models:

$$\min_{\mathbf{x}\in\mathbb{R}^d} f(A\mathbf{x}) + g(\mathbf{x}), \tag{4.16}$$

where $A \in \mathbb{R}^{n\times d}$. Typically $f : \mathbb{R}^n \to \mathbb{R}$ is the likelihood function and $g : \mathbb{R}^d \to \mathbb{R}$ is the regularizer (we absorb the regularization constant into g). Each row of A corresponds to a sample point and we have suppressed the labels in classification or responses in regression. Support vector machines, Lasso, logistic regression, boosting, etc. all fit under this framework. Our interest here is when the model dimension d is much higher than the number of samples n (d can be up to hundreds of millions and n can be up to millions). This is the usual setup in many computational biology and health care problems.

A naive implementation of msPG might be inefficient in terms of both network communication and parameter storage. First, Each machine needs to communicate with every other machine, to exchange the latest block of parameters. If using a peer-to-peer network topology, the resulting connections will be too dense and crowded when the system holds hundreds of machines. We resolve this issue by adopting the parameter server system advocated in previous works [65, 94], that is, we dedicate a specific server (which can span a set of machines if needed) to store the key parameters (will be specified later) and let each worker machine to communicate only with the server. There is a second advantage for this master-slave network topology, as we shall see momentarily.

Second, each machine needs to keep a local copy of the full model (i.e. $\mathbf{x}^i(t)$ in msPG), which can incur a very expensive storage cost when the dimension is high. This is where the linear model structure in (4.16) comes into help. Note that the local models $\mathbf{x}^i(t)$ are kept solely for the convenience of evaluating the partial gradient $\nabla_i f : \mathbb{R}^d \to \mathbb{R}^{d_i}$. For some problems such as the Lasso, a seemingly easy workaround is to pre-compute the Hessian $H = A^\top A$ and distribute the corresponding row blocks of H to each worker machine. This scheme, however, is problematic in the high dimensional setting: the pre-computation of the Hessian can be very costly, and each row block of H has a very large size ($d_i \times d$).

Instead, we use the column partition scheme [e.g. 27, 124], namely, we partition the matrix A into p *column* blocks $A = [A_1, \ldots, A_p]$ and distribute the block $A_i \in \mathbb{R}^{n \times d_i}$ to machine i. Now the local update computed by machine i at the t-th iteration can be rewritten as

$$U_i(\mathbf{x}^i(t)) = \operatorname{prox}_{g_i}^{\eta}\left(x_i(t) - \eta A_i^\top f'(A\mathbf{x}^i(t))\right) - x_i(t) \tag{4.17}$$

Since machine i is in charge of updating the i-th block $x_i(t)$ of the global model, to compute the local update (4.17) it is sufficient to have the matrix-vector product $A\mathbf{x}^i(t)$. For simplicity we initialize $\forall i,\ \mathbf{x}^i(0) \equiv \mathbf{0}$, then we have the following cumulative form:

$$A\mathbf{x}^i(t) = \sum_{j=1}^{p} A_j[\mathbf{x}^i(t)]_j = \sum_{j=1}^{p} \sum_{k=0}^{\tau_j^i(t)} \underbrace{A_j \mathbb{I}_{\{k \in T_j\}} U_j(\mathbf{x}^j(k))}_{\Delta_j(k)},$$

where recall that when machine i conducts its t-th iteration it only has access to a delayed copy $x_j(\tau_j^i(t))$ of the parameters in machine j. Since this matrix-vector product is needed by every machine to conduct their local updates in (4.17), we aggregate $\Delta_j(t) \in \mathbb{R}^n$ on the parameter server whenever it is generated and sent by the worker machines. In details, the worker machines first pull this aggregated matrix-vector product (denoted as $\blacktriangle$) from the server to conduct the local computation (4.17) in an economical way (by replacing $A\mathbf{x}^i(t)$ in (4.17) with $\blacktriangle$). Then machine i performs the simple update:

$$x_i(t+1) = x_i(t) + U_i(\mathbf{x}^i(t)). \tag{4.18}$$

Note that machine i does not maintain or update other blocks of parameters $x_j(t), j \neq i$. Lastly, machine i computes and sends the vector $\Delta_i(t) = A_i U_i(\mathbf{x}^i(t)) \in \mathbb{R}^n$ to the server, and the server immediately performs the aggregation:

$$\blacktriangle \leftarrow \blacktriangle + \Delta_i(t). \tag{4.19}$$

We summarize the above economical implementation in Algorithm 1, where $\blacktriangle$ denotes the aggregation of individual matrix-vector products Δ on the server. The storage cost for each worker machine is $O(nd_i)$ (for storing A_i). Each iteration requires two matrix-vector products that cost $O(nd_i)$ in the dense case, and the communication of a length n vector between the server and the worker machines.

Algorithm 5 Economic Implementation of msPG

```
For the server:
    while recieves update $\Delta_i$ from machine $i$ do
        $\blacktriangle \leftarrow \blacktriangle + \Delta_i$
    end while
    while machine $i$ sends a pull request do
        send $\blacktriangle$ to machine $i$
    end while
For machine $i$ at active clock $t \in T_i$:
    pull $\blacktriangle$ from the server
    $U_i \leftarrow \operatorname{prox}_{g_i}^{\eta}\big(x_i - \eta A_i^\top f'(\blacktriangle)\big) - x_i$
    send $\Delta_i = A_i U_i$ to the server
    update $x_i \leftarrow x_i + U_i$
```

4.6 Experiments

4.6.1 Group Lasso

We first test the convergence properties of msPG via a non-convex Lasso problem with the group regularizer $\|\cdot\|_{0,2}$, which takes the form

$$\min_{\mathbf{x}} \ \tfrac{1}{2}\|A\mathbf{x} - \mathbf{b}\|^2 + \lambda \textstyle\sum_{i=1}^{20} \mathbb{I}(\|x_i\| = 0), \tag{4.20}$$

Here $A \in \mathbb{R}^{1000\times 2000}, \mathbf{b} \in \mathbb{R}^{1000}$, and $\mathbf{x} \in \mathbb{R}^{2000}$ is divided into 20 equal groups of features. Matrix A is generated from $\mathcal{N}(0,1)$ with normalized columns. We set $\mathbf{b} = A\tilde{\mathbf{x}} + \varepsilon$, where ε is generated from $\mathcal{N}(0, 10^{-2})$ and $\tilde{\mathbf{x}}$ is a normalized vector with 8 non-zero groups of features generated from $\mathcal{N}(0,1)$. For the non-zero groups of $\tilde{\mathbf{x}}$, we set the corresponding $\gamma_i = 10^{-4}$, and for the remaining groups we set $\gamma_i = 10^{-2}$.

We implement msPG on four cores with each core assigned five group of features. Each core stores the corresponding column blocks of A.

We use 4 machines (cores) with each handling five groups of coordinates, and consider staleness $s = 0, 10, 20, 30$, respectively. To better demonstrate the effect of staleness, we let machines only communicate when exceeding the maximum staleness. This can be viewed as the worst case communication scheme and a larger s brings more staleness into the system. We set the learning rate to have the form $\eta(\alpha s) = 1/(L_f + 2L\alpha s), \alpha > 0$, that is, a linear dependency on staleness s as suggested by Theorem 4.1. Then we run Algorithm 5 with different staleness and use $\eta(10)$ and $\eta^*(\alpha s)$, where $\eta^*(\alpha s)$ is the largest step size we tuned for each s that achieves a stable convergence. We track the global model $\mathbf{x}(t)$ and plot the results in Figure 4.2, Fig. 4.3, and Fig. 4.4. Note that with the large step size $\eta(0)$ all instances (with nonzero staleness) diverge hence are not presented. With $\eta(10)$ (Figure 4.2), the staleness does not substantially affect the convergence in terms of the objective value. We note that the objective curves converge to

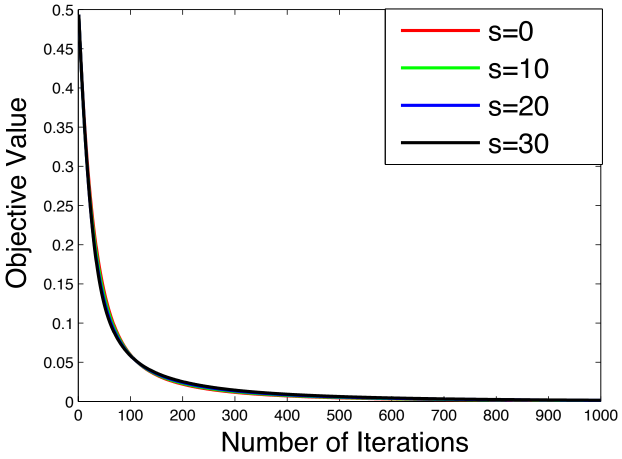


Figure 4.2: Convergence over clock for group lasso under msPG using learning rate $\eta(10)$ for staleness $s = 0, 10, 20, 30$.

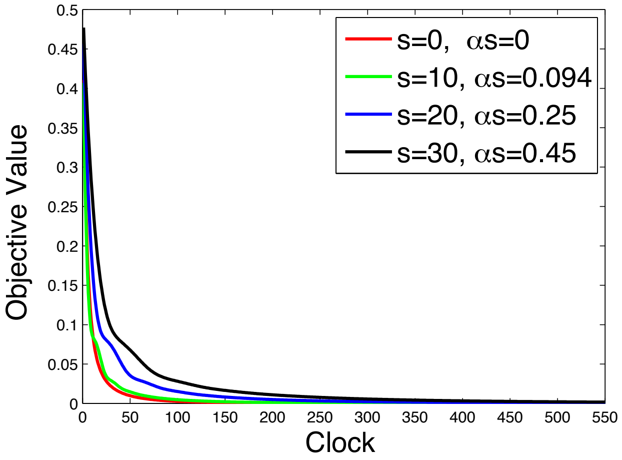


Figure 4.3: Convergence of parameter over clock for group lasso under msPG using learning rate $\eta^*(\alpha s)$ for staleness $s = 0, 10, 20, 30$.

slightly different minimal values due to the non-convexity of problem (4.20). With $\eta^*(\alpha s)$ (Figure 4.3), it can be observed that adding a slight penalty αs on the learning rate suffices to achieve a stable convergence, and the penalty grows as s increases, which is intuitive since a larger staleness requires a smaller step size to offset the inconsistency. In particular, for $s = 10$ the best convergence is comparable to the bulk synchronized case $s = 0$. (Figure 4.4) further shows the asymptotic convergence behavior of the global model $\mathbf{x}(t)$ under the step size $\eta^*(\alpha s)$. It is clear that a linear convergence is eventually attained, which confirms the finite length property in Theorem 4.2.

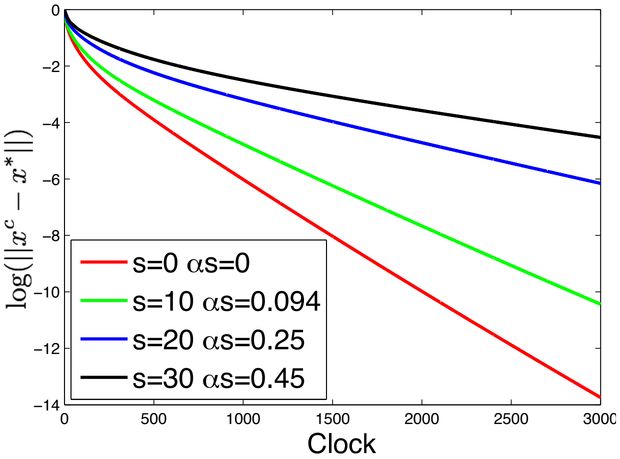


Figure 4.4: Convergence over clock for group lasso under msPG using learning rate $\eta^*(\alpha s)$ for staleness $s = 0, 10, 20, 30$. The linear convergence on the logrithmic scale is consistent with the finite length property in Theorem 4.2.

4.6.2 Large-scale Lasso

Next, we verify the time and communication efficiency of msPG via an l_1 norm Lasso problem with very high dimensions, taking the form

$$\min_{\mathbf{x}} \ \tfrac{1}{2}\|A\mathbf{x} - \mathbf{b}\|_2^2 + \lambda\|\mathbf{x}\|_1. \tag{4.21}$$

We generate $A \in \mathbb{R}^{n\times d}$ of size $n = 10^6$ and $d = 10^8$ as follows:

Data Generation. We generate the data column-wise. Starting from first column, we randomly pick 10^4 samples to have non-zero in column 1 and sample each value from Uniform$(-1, 1)$. We normalize it such that the ℓ_2-norm of the column is 1. We denote these values as $\mathbf{v}_1 \in \mathbb{R}^n$. To generate column i, with probability 0.5 we randomly pick a new set of samples to have non-zero values at column i (otherwise we use the same samples from column $i - 1$). This simulates the correlations between each column. Once the samples are chosen, we assign values from Unif$(-1, 1)$. $\mathbf{v}_i$ is again normalized. We generate ground truth regressor $\beta \in \mathbb{R}^d$ from $\mathcal{N}(0, 1)$ with 1% non-zero entries, and obtain the regressed value from $\mathbf{b} = A\beta$ where A, the design matrix, is the conctenation of $v_1, ..., v_{10^8}$.

We implement Algorithm 5 on Petuum [40, 65] — a stale synchronous parallel system which eagerly updates the local parameter caches via stale synchronous communications. The system contains 100 computing nodes and each is equipped with 16 AMD Opteron processors and 16GB RAM linked by 1Gbps ethernet. We fix the learning rate $\eta = 10^{-3}$ and consider maximum staleness $s = 0, 1, 3, 5, 7$, respectively. Figure 4.5 shows that per-iteration progress is virtually indistinguishable among various staleness settings, which is consistent with the previous experiment with group lasso. Figure 4.6 shows that system throughput is significantly higher when we introduce staleness. This is due to lower synchronization overheads, which offsets any potential loss due to staleness in progress per iteration. We also track the distributions of staleness during the experiments, where we record in ▲ the clocks of the freshest updates that accumulate

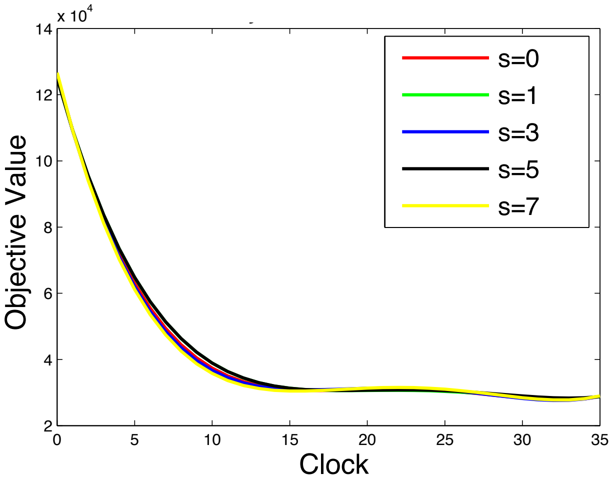


Figure 4.5: Convergence over clock for large-scale lasso under msPG for staleness $s = 0, 1, 3, 5, 7$.

from all the machines. Then whenever a machine pulls ▲ from the server, it compares its local clock with these clocks and records the clock differences. Figure 4.7 shows the distributions of staleness under different maximal staleness settings. Observe that bulk synchronous ($s = 0$) peaks at staleness 0 by design, and the distribution concentrates in small staleness area due to the eager communication mechanism of Petuum. It can be seen that a small amount of staleness is sufficient to relax the communication bottlenecks without affecting the iterative convergence rate much.

4.7 Additional Related Work

The stale synchronous parallel system dates back to [18, 19, 137, 138], where it is also referred to as partially asynchronous system. These work consider using stale synchronous systems to solving different kinds of optimization problems with allowing machines to skip updates during the process. Asymptotic convergence of partially asynchronous gradient descent algorithm for solving unconstrained smooth optimization is established in [19], with its stochastic version being analyzed in [138]. Asymptotic convergence of partially asynchronous gradient projection algorithm for solving smooth optimization with convex constraint is established in [19], and a "B-step" linear convergence is further established in [137] with an error bound condition. Linear convergence of partially asynchronous algorithm for finding the fixed point of maximum norm contraction mappings is established in [18, 47].

Another series of work focus on SSP systems where machines are not allowed to skip updates [92, 94]. In their settings, The system imposes an upper bound on the maximum clock difference between machines. Asymptotic convergence is established for proximal gradient algorithm for data parallelism [93] and for block coordinate descent [92] with a smooth objective and convex regularizer. Other works consider stochastic algorithms on stale synchronous system.

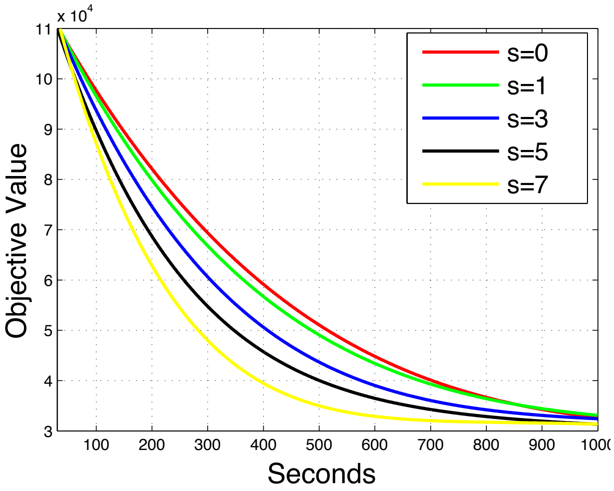


Figure 4.6: Convergence over wall clock time for large-scale lasso under msPG for staleness $s = 0, 1, 3, 5, 7$.

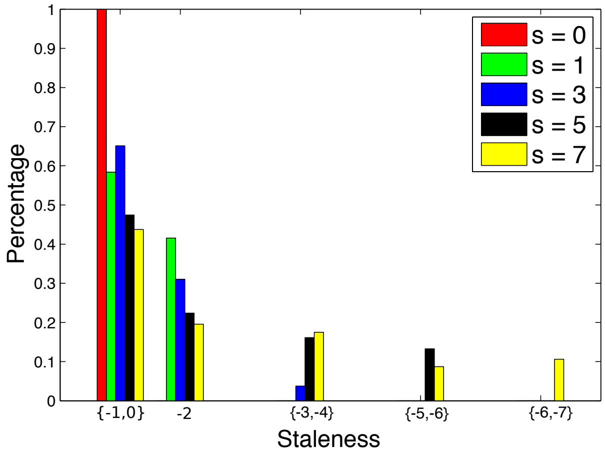


Figure 4.7: Empirical staleness distribution under staleness $s = 0, 1, 3, 5, 7$. X-axis is parameter age - local clock, *i.e.*, the absolute value of clock differential, similar to Fig. 3.7. Y-axis is the distribution of the clock differentials observed in parameter reads. $s = 0$ reduces to Bulk Synchronous Parallel where the empirical staleness is concentrated at clock differential -1.

[65] proposes an SSP system for stochastic gradient descent, and establishes $O(1/\sqrt{k})$ regret bound under bounded diameter and bounded sub-gradient assumption. [48, 120] consider a delayed stochastic gradient descent algorithm. Linear convergence to a neighborhood of optimum is established with strong convexity assumption in [48] and with additional bounded gradient assumption in [120]. [4] proposes a distributed delayed dual averaging and mirror descent algorithm, and establishes $O(1/\sqrt{k})$ regret bound under standard stochastic assumptions.

Chapter 5

Staleness in Parallel Frank-Wolfe Algorithms

The classical Frank-Wolfe (FW) algorithm [50] has witnessed a surge of interest recently, due to the simple subproblem structure that can scale to large dataset [5, 34, 68, 69]. For example, FW is the state-of-the-art method for structural SVMs [82], whose dual problem yields a large number of variables that renders classical projected gradient descent intractable. Although FW is known for more than half a century, there was not parallelizable. In this chapter, we consider a class of problems that exhibit block-separable structure optimized by FW, and characterize the convergence behaviors by staleness and problem-dependent quantities.

5.1 Introduction

The FW algorithm iteratively minimizes a smooth function f (typically convex) over a compact convex set $\mathcal{M} \subset \mathbb{R}^m$. Unlike methods based on projection, FW uses just a *linear oracle* that solves $\min_{x \in \mathcal{M}} \langle x, g \rangle$, which can be much simpler and faster than projection.

This feature underlies the great popularity of FW, which has by now witnessed several extensions such as regularized FW [28, 60, 160], linearly convergent special cases [54, 81], stochastic versions [62, 83, 115], and a randomized block-coordinate FW [82].

Despite this progress, parallel and distributed FW variants are barely known. We fill this gap and develop new asynchronous FW algorithms, for the particular setting where the constraint set $\mathcal{M}$ is *block-separable*; thus, we solve

$$\min_x \; f(x) \text{ s.t. } x = [x_{(1)}, ..., x_{(n)}] \in \prod_{i=1}^{n} \mathcal{M}_i, \tag{5.1}$$

where $\mathcal{M}_i \subset \mathbb{R}^{m_i}$ $(1 \leq i \leq n)$ is a compact convex set and $x_{(i)}$ are coordinate partitions of x. This setting for FW was considered in [82], who introduced the Block-Coordinate Frank-Wolfe (BCFW) method.

Such problems arise in many applications, notably, structural SVMs [82], routing [88], group fused lasso [7, 21], trace-norm based tensor completion [97], reduced rank nonparametric regression [49], and structured submodular minimization [70], among others.

A standard approach to solve (5.1) is via block-coordinate (gradient) descent (BCD), which forms a local quadratic model for a block of variables, and then solves a *projection* subproblem [16, 113, 125]. However, for many problems, including the ones noted above, projection can be expensive (e.g., projecting onto the trace norm ball, onto base polytopes 52), and in some cases even computationally intractable [35].

Frank-Wolfe methods excel in such scenarios as they rely only on linear oracles that solve $\min_{s\in\mathcal{M}}\langle s, \nabla f(\cdot)\rangle$. For $\mathcal{M} = \prod_i \mathcal{M}_i$, this breaks into the n independent problems

$$\min_{s_{(i)}\in\mathcal{M}_i} \quad \langle s_{(i)}, \nabla_{(i)} f(x)\rangle, \quad 1 \le i \le n, \tag{5.2}$$

where $\nabla_{(i)}$ denotes the gradient w.r.t. coordinates $x_{(i)}$. It is obvious that these n subproblems can be solved in parallel (an idea dating back to at least as early as 88). However, having to update all the coordinates at each iteration is expensive, hampering the use of FW on big-data problems.

This drawback is partially ameliorated by BCFW [82], which randomly selects a block $\mathcal{M}_i$ at each iteration and performs FW updates. But these updates are *strictly* sequential, and do not take advantage of modern multicore architectures or of distributed clusters.

Contributions. Our main contributions are the following:

- Asynchronous Parallel block-coordinate Frank-Wolfe algorithms (AP-BCFW) for both shared-memory and distributed settings. Moreover, AP-BCFW depends only (mildly) on the *expected* delay, therefore is robust to stragglers and faulty worker threads.
- An analysis of the primal and primal-dual convergence of AP-BCFW and its variants for any minibatch size and potentially unbounded maximum delay. When the maximum delay is actually bounded, we show stronger results using results from load-balancing on max-load bounds.
- Insightful deterministic conditions under which minibatching *provably* improves the convergence rate for a class of problems (sometimes by orders of magnitude).
- Experiments that demonstrate on real data how our algorithm solves a structural SVM problem several times faster than the state-of-the-art.

In short, our results contribute towards making FW more attractive for big-data applications. To add perspective, we compare our methods to closely related works below; we refer the reader to [51, 69, 82, 159] for additional notes and references.

Notation. We briefly summarize our notation now. The vector $x \in \mathbb{R}^m$ denotes the parameter vector, possibly split into n coordinate blocks. For block $i = 1, ..., n$, $E_i \in \mathbb{R}^{m\times m_i}$ is the projection matrix which projects $x \in \mathbb{R}^m$ down to $x_{(i)} \in \mathbb{R}^{m_i}$; thus $x_{(i)} = E_i x$. The adjoint operator E_i^* maps $\mathbb{R}^{m_i} \to \mathbb{R}^m$, thus $x_{[i]} = E_i^* x_{(i)}$ is x with zeros in all dimensions except $x_{(i)}$ (note the subscript $x_{[i]}$). We denote the size of a minibatch by τ, and the number of parallel workers (threads) by T. Unless otherwise stated, k denotes the iteration/epoch counter and γ denotes a stepsize.

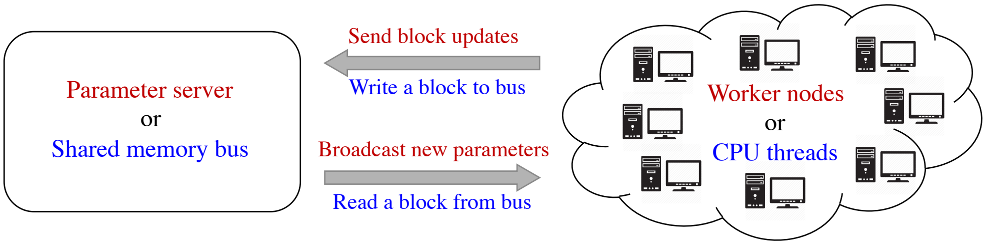


Figure 5.1: Illustration of the AP-BCFW in the distributed (in red) and share-memory settings (in blue). The "cloud" of all worker nodes (or CPU threads) is abstracted into an oracle that keeps feeding the server (or writing to the memory bus) with updates from solving possibly approximate (and/or delayed) solutions to (5.2) on iid uniform random blocks.

Finally, C_f^τ (and other such constants) denotes some curvature measure associated with function f and minibatch size τ. Such constants are important in our analysis, and will be described in greater detail in the main text.

5.2 Algorithm

In this section, we present an asynchronous parallel block-coordinate Frank-Wolfe algorithm (AP-BCFW) to solve (5.1). Our algorithm is designed to run fully asynchronously on either a shared-memory multicore architecture or on a distributed system.

For the shared-memory model, the computational work is divided amongst worker threads, each of which has access to a pool of coordinates that it may work on, as well as to the shared parameters. This setup matches the system assumptions in [98, 114, 125], and most modern multicore machines permit such an arrangement.

On a distributed system, the parameter server [39, 94] broadcasts the most recent parameter vector periodically to each worker and workers keep sending updates to the parameter vector after solving the subroutines corresponding to a randomly chosen parameter. In either setting, we do not wait for slower workers or synchronize the parameters at any point of the algorithm, therefore many updates sent from the workers could be calculated based on a delayed parameter.

For convenience, we treat the pool of all workers as a single "cloud" oracle $\mathcal{O}$ that keeps sending updates of form $\{i, s_{(i)}\}$ to the server, where i selects a block and $s_{(i)}$ is an approximate solution to (5.2) at the current parameter. Moreover, we assume that

A1. The sequence of i from $\mathcal{O}$ is sampled i.i.d. uniformly from $\{1, 2, ..., n\}$.

Assumption A1 is critical as it ensures Step 2 in the algorithm is an unbiased approximation of the batch FW. This assumption allows the workers to be arbitrarily heterogeneous as long as they each sample blocks i.i.d. uniformly and the time for each worker to produce $s_{(i)}$ does not

---

[1] We bound the probability of collisions in Appendix D.4.2.

Algorithm 6 AP-BCFW: Asynchronous Parallel Block-Coordinate Frank-Wolfe (distributed)

```
Input: An initial feasible $x^{(0)}$, mini-batch size $\tau$, a "Cloud" oracle $\mathcal{O}$ satisfying Assumptions A1, A2.
0. Broadcast $x^{(0)}$ to all workers in $\mathcal{O}$.
for $k$ = 1,2,... ($k$ is the iteration number) do
  1. Keep receiving $(i, s_{(i)})$ from $\mathcal{O}$ until we have $\tau$ disjoint blocks (overwrite if collision[1]). Denote the index set by $S$.
  2. Update $x^{(k)} = x^{(k-1)} + \gamma_k \sum_{i \in S}(s_{[i]} - x^{(k-1)}_{[i]})$ with $\gamma_k = \frac{2n\tau}{\tau^2 k + 2n}$ or via line-search.
  3. Broadcast $x^{(k)}$ (or just $x^{(k)} - x^{(k-1)}$) to $\mathcal{O}$.
  4. Break if converged.
end for
Output: $x^{(k)}$.
```

depend on the block index i. Admittedly, this could be troublesome for some applications with heterogeneous blocks, we describe ways to enforce A1 in Appendix D.4.1.

An advantage of this oracle abstraction is its potential applicability well beyond the per-worker i.i.d. scheme. In practice, each worker might only have access to a small subset of $[n]$ and might be doing sequential sampling with periodic reshuffling. At the aggregate level, however, the oracle assumptions might still be reasonable approximations, especially if the number of workers T is large.

Both distributed and shared-memory settings can be captured under this oracle as is illustrated in Figure 5.1. Pseudocode of our scheme is given in Algorithm 6.

5.3 Analysis

The three key questions pertaining to Algorithm 6 are:

- *Does it converge?*
- *How fast? How much faster than* BCFW *($\tau = 1$)?*
- *How do delayed updates affect the convergence?*

We answer the first two questions in Sections 5.3.1 and 5.3.2. Specifically, we show that AP-BCFW converges at a $O(1/k)$ rate. Our analysis reveals that the speedup of AP-BCFW over BCFW via parallelization is problem dependent. Intuitively, we show that speedups due to mini-batching ($\tau > 1$) depend on the average "coupling" of the objective function f across different coordinate blocks. For example, if f has a block symmetric diagonally dominant Hessian, then AP-BCFW converges $\tau/2$ times faster. We address the third question in Section 5.3.3, where we establish convergence results that depend only mildly on the "expected" delay κ. The bound is proportional to κ when we allow the delay to grow unboundedly, and proportional to $\sqrt{\kappa}$ when the delay is bounded by a small $\kappa_{\max}$.

5.3.1 Main convergence results

We begin by defining a few quantities needed for our analysis. The first key quantity—also key to the analysis of several other FW methods—is the notion of **curvature**. Since AP-BCFW updates a subset of coordinate blocks at a time, we define *set curvature* for an index set $S \subseteq [n]$ as

$$C_f^{(S)} := \sup_{\substack{x\in\mathcal{M}, s_{(S)}\in\mathcal{M}^{(S)},\\ \gamma\in[0,1],\\ y=x+\gamma(s_{[S]}-x_{[S]})}} \frac{2}{\gamma^2}\big(f(y)-f(x)- \langle y_{(S)}-x_{(S)}, \nabla_{(S)} f(x)\rangle\big). \tag{5.3}$$

For index sets of size τ, we define the *expected set curvature* over a uniform choice of subsets as

$$C_f^{\tau} := \mathbb{E}_{S:|S|=\tau}[C_f^{(S)}] = \binom{n}{\tau}^{-1} \sum\nolimits_{S\subset[n],|S|=\tau} C_f^{(S)}. \tag{5.4}$$

These curvature definitions are closely related to the global curvature constant C_f of [69] and the coordinate curvature $C_f^{(i)}$ and product curvature $C_f^{\otimes}$ of [82]. Lemma 5.1 makes this relation more precise.

Lemma 5.1 (Curvature relations)**.** *Suppose $S \subseteq [n]$ with cardinality $|S| = \tau$ and $i \in S$. Then,*

i) $C_f^{(i)} \leq C_f^{(S)} \leq C_f$;

ii) $\frac{1}{n} C_f^{\otimes} = C_f^1 \leq C_f^{\tau} \leq C_f^n = C_f$.

How the expected set curvature C_f^{τ} scales with τ is critical to bounding the speedup we can expect over BCFW; we provide a detailed analysis of this speedup in Section 5.3.2.

The next key object is an **approximate linear minimizer**. As in [69, 82], we also allow the core computational subroutine that solves (5.2) to yield an approximate minimizer $s_{(i)}$. Formally, we assume:

A2. There is a constant $\delta \geq 0$, such that for every $k \geq 1$, the chosen minibatch $S \subset [n]$ of size τ and the corresponding blocks $s_{(S)} := (s_{(i)})_{i\in S}$ from $\mathcal{O}$ obey

$$\mathbb{E}\left[\langle s_{(S)}, \nabla_{(S)} f^{(k)}\rangle - \min_{s'\in\mathcal{M}^{(S)}} \langle s', \nabla_{(S)} f^{(k)}\rangle\right] \leq \frac{\delta\gamma_k C_f^{\tau}}{2}. \tag{5.5}$$

where the expectation is taken over the random sequence of minibatch indices and corresponding updates from $\mathcal{O}$ in the entire history up to step k.

Assumption A2 is strictly weaker than what is required in [69, 82], as we only need the approximation to hold *in expectation*. With these definitions in hand, we are ready to state our convergence result.

Theorem 5.1 (Primal Convergence)**.** *Say we use a "Cloud" oracle $\mathcal{O}$ that generates a sequence of updates satisfying A1 and A2. Then, for each $k \geq 0$, the iterations in Algorithm 6 and its line search variant obey*

$$\mathbb{E}[f(x^{(k)})] - f(x^*) \leq \tfrac{2nC}{\tau^2 k + 2n},$$

where $C = nC_f^{\tau}(1+\delta) + f(x^{(0)}) - f(x^*)$.

At a first glance, the $n^2 C_f^\tau$ term in the numerator might seem bizarre, but as we will see in the next section, C_f^τ can be as small as $O(\frac{\tau}{n^2})$. This is the scale of the constant one should keep in mind to compare the rate to other methods, e.g., coordinate descent. Also note that so far this convergence result does not explicitly work for delayed updates, which we will analyze in Section 5.3.3 separately via the approximation parameter δ from (5.5).

For FW methods, one can also easily obtain a convergence guarantee in an appropriate primal-dual sense. To this end, we introduce our version of the **surrogate duality gap** [69]; we define this as

$$\begin{aligned} g(x) &:= \max_{s \in \mathcal{M}} \langle x - s, \nabla f(x) \rangle \qquad (5.6) \\ &= \sum_{i=1}^{n} \max_{s_{(i)} \in \mathcal{M}^{(i)}} \langle x_{(i)} - s_{(i)}, \nabla_{(i)} f(x) \rangle = \sum_{i=1}^{n} g^{(i)}(x). \end{aligned}$$

To see why (5.6) is actually a duality gap, note that since f is convex, the linearization $f(x) + \langle s - x, \nabla f(x) \rangle$ is always smaller than the function evaluated at any s, so that

$$g(x) \geq \langle x - x^*, \nabla f(x) \rangle \geq f(x) - f(x^*).$$

This duality gap is obtained for "free" in batch FW, but not in BCFW or AP-BCFW. Here, we only have an unbiased estimate $\frac{n}{|S|} \sum_{i \in S} g^{(i)}(x)$. For large τ, this estimate is close to $g(x)$ with high probability (McDiarmid's Inequality), and can still be useful as a stopping criterion.

Theorem 5.2 (Primal-Dual Convergence)**.** *Assume $\mathcal{O}$ satisfies A1 and A2. Define the expected surrogate duality gap $g_k := \mathbb{E} g(x^{(k)})$ and weighted average $\bar{g}_k := \frac{2}{K(K+1)} \sum_{k=1}^{K} k g_k$ for the sequence of parameters $x^{(k)}$ in Algorithm 6. Then for every $K \geq 1$, there exists $k^* \in [K]$ such that*

$$g_{k^*} \leq \bar{g}_K \leq \frac{6nC}{\tau^2 (K+1)},$$

with the same C in Theorem 5.1.

Relation with FW and BCFW: The above convergence guarantees can be thought of as an interpolation between BCFW and batch FW. If we take $\tau = 1$, they give exactly the convergence guarantee for BCFW [82, Theorem 2], and if we take $\tau = n$, we can drop $f(x^{(0)}) - f(x^*)$ from C (with a small modification in the analysis) and recover the classic batch guarantee as in [69].

Dependence on initialization: Unlike classic FW, the convergence rate of our method depends on the initialization. When $h_0 := f(x^{(0)}) - f(x^*) \geq n C_f^\tau$ and $\tau^2 < n$, the convergence is slower by a factor of $\frac{n}{\tau^2}$. The same concern was also raised in [82] with $\tau = 1$. We can actually remove the $f(x^{(0)}) - f(x^*)$ from C as long as we know that $h_0 \leq n C_f^\tau$. By Lemma 5.1, the expected set curvature C_f^τ increases with τ, so the fast convergence region becomes larger when we increase τ. In addition, if we pick $\tau^2 > n$, the rate of convergence is not affected by initialization anymore.

Speedup: The reader may have noticed the $n^2 C_f^\tau$ term in the numerator. This is undesirable as n can be large (for instance, in structural SVM n is the total number of data points). The saving grace in BCFW is that when $\tau = 1$, C_f^τ is as small as $O(n^{-2})$ (see 82, Lemmas A1 and A2), and it is easy to check that the dependence on n is the same even for $\tau > 1$. What really matters is how much speedup one can achieve over BCFW, and this relies critically on how C_f^τ depends on τ. Analyzing this dependence is our main focus in the next section.

5.3.2 Effect of parallelism / mini-batching

To understand when mini-batching is meaningful and to quantify its speedup, below we take a more careful look at the expected set curvature C_f^τ. In particular, we analyze and present a set of insightful conditions that govern its dependence on τ. The key idea is to quantify how strongly different coordinate blocks interact with each other.

To begin, assume that there exists a positive semidefinite matrix H such that for any $x, y \in \mathcal{M}$

$$f(y) \leq f(x) + \langle y - x, \nabla f(x) \rangle + \frac{1}{2}(y-x)^T H (y-x). \tag{5.7}$$

The matrix H may be viewed as a generalization of the gradient's Lipschitz constant (a scalar) to a matrix. For quadratic functions $f(x) = \frac{1}{2}x^T Q x + c^T x$, we can take $H = Q$. For twice differentiable functions, we can choose $H \in \{K \mid K \succeq \nabla^2 f(x), \ \ \forall x \in \mathcal{M}\}$.

Since $x = [x_1, \ldots, x_n]$ (we write x_i instead of $x_{(i)}$ for brevity), we separate H into $n \times n$ blocks; so H_{ij} represents the block corresponding to x_i and x_j such that we can take the product $x_i^T H_{ij} x_j$. Now, we define a *boundedness* parameter B_i for every i, and an *incoherence condition* with parameter μ_{ij} for every block coordinate pair $\mathcal{M}_i, \mathcal{M}_j$ such that

$$\begin{aligned} B_i &= \sup_{x_i \in \mathcal{M}_i} x_i^T H_{ii} x_i, & \mu_{ij} &= \sup_{x_i \in \mathcal{M}_i, x_j \in \mathcal{M}_j} x_i^T H_{ij} x_j, \\ B &= \mathbb{E}_{i \sim \mathrm{Unif}([n])} B_i, & \mu &= \mathbb{E}_{(i,j) \sim \mathrm{Unif}(\{(\mathrm{i,j}) \in [\mathrm{n}]^2, \mathrm{i} \neq \mathrm{j}\})} \mu_{ij}. \end{aligned}$$

Using these quantities, we obtain the following bound on the expected set-curvature.

Theorem 5.3. $C_f^\tau \leq 4(\tau B + \tau(\tau - 1)\mu)$ *for any* $\tau \in [n]$.

It is clear that when the incoherence term μ is large, the expected set curvature C_f^τ is proportional to τ^2, and when μ is close to 0, then C_f^τ is proportional to τ. In other words, when the interaction between coordinates block is small, one gains from parallelizing BCFW. This is analogous to the situation in parallel coordinate descent [98, 125] and we will compare the rate of convergence explicitly with them in Appendix D.4.5.

Corollary 1. *Consider a matrix M with B_i on the diagonal and μ_{ij} on the off-diagonal. If M is* symmetric diagonally dominant *(SDD), i.e., the sum of absolute off-diagonal entries in each row is no greater than the diagonal entry, then C_f^τ is proportional to τ.*

The above result depends on the parameters B and μ. In Appendix D.4.4, we provide two concrete examples (multi-class classification with structural SVM and graph fused lasso) where we can express B and μ as problem-dependent quantities and provide explicit upper bounds of C_f^τ. In both examples, we show that choosing larger τ yields faster convergence (at least up to some point).

5.3.3 Convergence with delayed updates

Often due to the delays in communication, some updates pushed back by workers are calculated based on delayed parameters that were broadcast earlier. Dropping these updates or enforcing synchronization will create a huge system overhead especially when the size of the minibatch is small. Ideally, we want to just accept the delayed updates as if they were correct, and broadcast new parameters to workers without locking the updates. The question is, does this idea work?

In this section, we model delays from updates to be i.i.d. from an unknown distribution that can depend on k, but not on blocks. Under these assumptions, we show that the effect of delayed updates can be treated as an approximate oracle that satisfies A2 in (5.5) with some specific constant δ that depends on the expected delay κ and the maximum delay parameter $\kappa_{\max}$ (when it exists). This allows us to invoke results in Section 5.3.1 to establish convergence for delayed updates. The results also depend on the following diameter and gradient Lipschitz constant for a norm $\|\cdot\|$

$$\begin{aligned}
D^{(S)}_{\|\cdot\|} &= \sup_{x,y\in\mathcal{M}^{(S)}} \|x-y\|,\\
L^{(S)}_{\|\cdot\|} &= \sup_{\substack{x,y\in\mathcal{M},y=x+s\\ \|s\|\le\gamma,\\ s\in\mathrm{span}(\mathcal{M}^{(S)})}} \frac{1}{\gamma^2}(f(y)-f(x)-\langle y-x,\nabla f(x)\rangle),\\
D^{\tau}_{\|\cdot\|} &= \max_{S\subset[n]\Big||S|=\tau} D^{(S)}_{\|\cdot\|},\ \text{and}\ L^{\tau}_{\|\cdot\|} = \max_{S\subset[n]||S|=\tau} L^{(S)}_{\|\cdot\|}.
\end{aligned}$$

Theorem 5.4 (Delayed Updates as Approximate Oracle)**.** *For each norm $\|\cdot\|$ of choice, let $D^{\tau}_{\|\cdot\|}$ and $L^{\tau}_{\|\cdot\|}$ be defined above. Let the a random variable of delay be $\varkappa$ and let $\kappa := \mathbb{E}\varkappa$ be the expected delay from any worker. Moreover, assume that the algorithm drops any updates with delay greater than $k/2$ at iteration k. Then for the version of the algorithm without line-search, the delayed oracle will produce $s\in\mathcal{M}^{(S)}$ such that* (5.5) *holds with*

$$\delta = 4\kappa\tau L^{1}_{\|\cdot\|}D^{1}_{\|\cdot\|}D^{\tau}_{\|\cdot\|}/(C^{\tau}_{f}). \tag{5.8}$$

Furthermore, if we assume that there is a $\kappa_{\max}$ such that $\mathbb{P}(\varkappa\le\kappa_{\max})=1$ for all k, then (5.5) *holds with $\delta = c_{n,\tau\kappa_{\max}}\frac{4\tau L^{1}_{\|\cdot\|}D^{1}_{\|\cdot\|}\mathbb{E}D^{\varkappa\tau}_{\|\cdot\|}}{C^{\tau}_{f}}$ where*

$$c_{n,\tau\kappa_{\max}} = \begin{cases} \frac{3\log n}{\log(n/(\tau\kappa_{\max}))} & \textit{if}\ \kappa_{\max}\tau < n/\log n,\\ O(\log n) & \textit{if}\ \kappa_{\max}\tau = O(n\log n),\\ \frac{(1+o(1))\tau\kappa_{\max}}{n} & \textit{if}\ \kappa_{\max}\tau \gg n\log n. \end{cases} \tag{5.9}$$

The results above imply that AP-BCFW (without line-search) converges in both primal optimality and in duality gap according to Theorems 5.1 and 5.2 with the same $O(1/k)$ rate. Comparing to versions that solve (5.2) exactly, the delayed version has an additional additive factor in the numerator of form

$$4n\kappa\tau L^{1}_{\|\cdot\|}D^{1}_{\|\cdot\|}D^{\tau}_{\|\cdot\|} \qquad \text{or} \qquad O\left(\tau L^{1}_{\|\cdot\|}D^{1}_{\|\cdot\|}\mathbb{E}D^{\varkappa\tau}_{\|\cdot\|}\log n\right)$$

with the additional assumption that $\kappa_{\max} = O(n \log n/\tau)$.

Note that (5.8) depends on the expected delay rather than the maximum delay, and as $k \to \infty$ we allow the maximum delay to grow unboundedly. This allows the system to automatically deal with heavy-tailed delay distributions and sporadic stragglers. When we do have a small bounded delay, we produce stronger bounds (5.9) with a multiplier that is either a constant (when $\tau\kappa_{\max} = O(n^{1-\epsilon})$ for any $\epsilon > 0$), proportional to $\log n$ (when $\tau\kappa \leq n$) or proportional to $\frac{\tau\kappa_{\max}}{n}$ (when $\tau\kappa$ is large). The whole expression often has sublinear dependence on the expected delay κ. For instance, we prove in the appendix the following:

Lemma 5.2. *When $\|\cdot\|$ is Euclidean norm*

$$\mathbb{E} D^{\varkappa\tau}_{\|\cdot\|} \leq D^{\lceil \mathbb{E}\varkappa\tau \rceil}_{\|\cdot\|} \leq \sqrt{\lceil \mathbb{E}\varkappa \rceil} D^{\tau}_{\|\cdot\|}.$$

The bound is proportional to $\sqrt{\kappa}$ when $\kappa = \Omega(1)$. This is strictly better than [114] which has quadratic dependence on $\kappa_{\max}$ and [98] which has exponential dependence on $\kappa_{\max}$. Our mild $\kappa_{\max}$ dependence for the cases $\tau\kappa_{\max} > n$ suggests that the (5.9) remains proportional to $\sqrt{\kappa}$ even when we allow the maximum delay parameter to be as large as n/τ or larger without significantly affecting the convergence. Note that this allows some workers to be delayed for several data passes.

Observe that when $\tau = 1$, where the results reduces to a lock-free variant for BCFW, δ becomes proportional to $L^1_{\|\cdot\|}[D^1_{\|\cdot\|}]^2/C^1_f$. This is always greater than 1 (see e.g., 69, Appendix D) but due to the flexibility of choosing the norm, this quantity corresponding to the most favorable norm is typically a small constant. For example, when f is a quadratic function, we show that $C^1_f = L^1_{\|\cdot\|}[D^1_{\|\cdot\|}]^2$ (see Appendix D.4.3). When $\tau > 1$, $\tau L^1_{\|\cdot\|} D^1_{\|\cdot\|} D^{\tau}_{\|\cdot\|}/C^{\tau}_f$ is often $O(\sqrt{\tau})$ for an appropriately chosen norm. Therefore, (5.8) and (5.9) are roughly in the order of $O(\kappa\sqrt{\tau})$ and $O(\sqrt{\kappa\tau})$ respectively.[2]

Lastly, we remark that κ and τ are not independent. When we increase τ, we update the parameters less frequently and κ gets smaller. In a real distributed system, with constant throughput in terms of the number of oracles that are solved per second from all workers. If the average delay is a fixed number in clock time specified by communication time. Then $\tau\kappa$ is roughly a constant regardless how τ is chosen.

5.4 Experiments

In this section, we experimentally demonstrate performance gains from the three key features of our algorithm: minibatches of data, parallel workers, and asynchrony.

---

[2] For details, see our discussion in Appendix D.4.3

5.4.1 Minibatches of Data

We conduct simulations to study the effect of mini-batch size τ, where larger τ implies greater degrees of parallelism as each worker can solve one or more subproblems in a mini-batch. In our simulation, we re-use the structural SVM setup from [82] for a sequence labeling task on a subset of the OCR dataset [136] $(n = 6251, d = 4082)$. The dual problem has block-separable probability simplex constraint therefore allowing us to run AP-BCFW, and each subproblem can be solved efficiently using the Viterbi algorithm (more details are included in Appendix D.3). The speedup on this dataset is shown in Figure 5.2(a). For this dataset, we use $\lambda = 1$ with weighted averaging and line-search throughout (no delay is allowed). We measure the speedup for a particular $\tau > 1$ in terms of the number of iterations (Algorithm 6) required to converge relative to $\tau = 1$, which corresponds to BCFW. Figure 5.2(a) shows that AP-BCFW achieves linear speedup for mini-batch size up to $\tau \approx 50$. Further speedup is sensitive to the convergence criteria.

In our simulation for Group Fused Lasso, we generate a piecewise constant dataset of size $(n = 100, d = 10$, in Eq. D.2) with Gaussian noise. We use $\lambda = 0.01$ and a primal suboptimality threshold as our convergence criterion. At each iteration, we solve τ subproblems (i.e. the mini-batch size). Fig. 5.2(b) shows the speed-up over $\tau = 1$ (BCFW). Similar to the structural SVM, the speedup is almost perfect for small τ $(\tau \leq 55)$ but tapers off for large τ to varying degrees depending on the convergence thresholds.

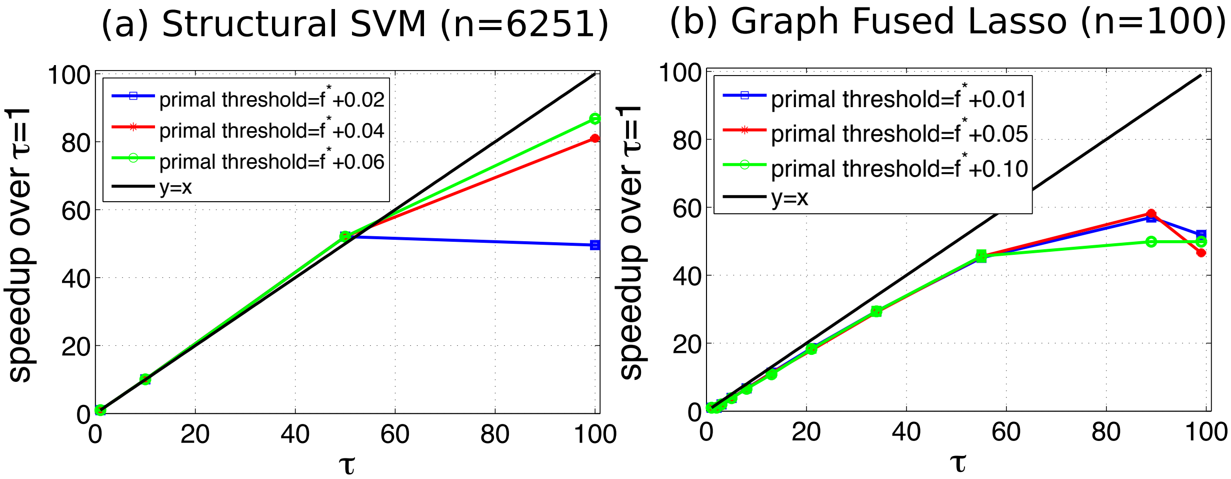


Figure 5.2: Performance improvement with τ for (a) Structual SVM on the OCR dataset [82, 136] and (b) Group Fused Lasso on a synthetic dataset. f^* denotes primal optimum (the "primal" problem is actually referring to the dual problem in both cases). The performance metric here is the number of iterations to achieve ϵ-suboptimality.

5.4.2 Shared Memory Parallel Workers

We implement AP-BCFW for the structural SVM in a multicore shared-memory system using the full OCR dataset ($n = 6877$). All shared-memory experiments were implemented in C++ and conducted on a 16-core machine with Intel(R) Xeon(R) CPU E5-2450 2.10GHz processors and 128G RAM. We first fix the number of workers at $T = 8$ and vary the mini-batch size τ. Fig. 5.3(a) shows the absolute convergence (i.e. the convergence per second). We note that AP-BCFW outperforms single-threaded BCFW under all investigated τ, showing the efficacy of parallelization. The optimal τ for a given number of workers (T) depends on both the dataset (how "coupled" are the coordinates) and also system implementations (how costly is the synchronization).

Since speedup for a given T depends on τ, we search for the optimal τ across multiples of T to find the best speedup for each T. Fig. 5.3(b) shows faster convergence of AP-BCFW over BCFW ($T = 1$) when $T > 1$ workers are available. It is important to note that the x-axis is *wall-clock time* rather than the number of epochs.

Fig. 5.3(c) shows the speedup with varying T. AP-BCFW achieves near-linear speed up for smaller T. The speed-up curve tapers off for larger T for two reasons: (1) Large T incurs higher system overheads, and thus needs larger τ to utilize CPU efficiently; (2) Larger τ incurs errors as shown in Fig. 5.2(a). If the subproblems were more time-consuming to solve, the affect of system overhead would be reduced. We simulate harder subproblems by simply solving them m ~Uniform$(5, 15)$ times instead of just once. The speedup is nearly perfect as shown in Fig. 5.3(d). Again, we observe that a more generous convergence threshold produces higher speedup, suggesting that resource scheduling could be useful (e.g., allocate more CPUs initially and fewer as algorithm converges).

We repeated the experiment on a larger synthetic dataset with $n = 103155, d = 4082$ created from the above mentioned OCR data as follows: for each of the 6877 words, generate 15 words with noisy images for characters, where the noise is introduced by flipping the bits of the images with probability 0.05 independently. The speedup with parallelization, shown in Fig. 5.4, essentially follows the same pattern as it did in Fig. 5.3(b)(c) for original data.

5.4.3 Performance gain with asynchronous updates

We compare AP-BCFW[3] with a synchronous version of the algorithm (SP-BCFW) where the server assigns τ/T subproblems to each worker, then waits for and accumulates the solutions before proceeding to the next iteration. We simulate workers of varying slow-downs in our shared-memory setup by assigning a *return probability* $p_i \in (0, 1]$ to each worker w_i. After solving each subproblem, worker w_i reports the solution to the server with probability p_i. Thus, a worker with $p_i = 0.8$ will drop 20% of the updates on average corresponding to 20% slow-down.

---

[3] The version that has no delayed updates, but allows workers to asynchronously solve subproblems within each mini-batch.

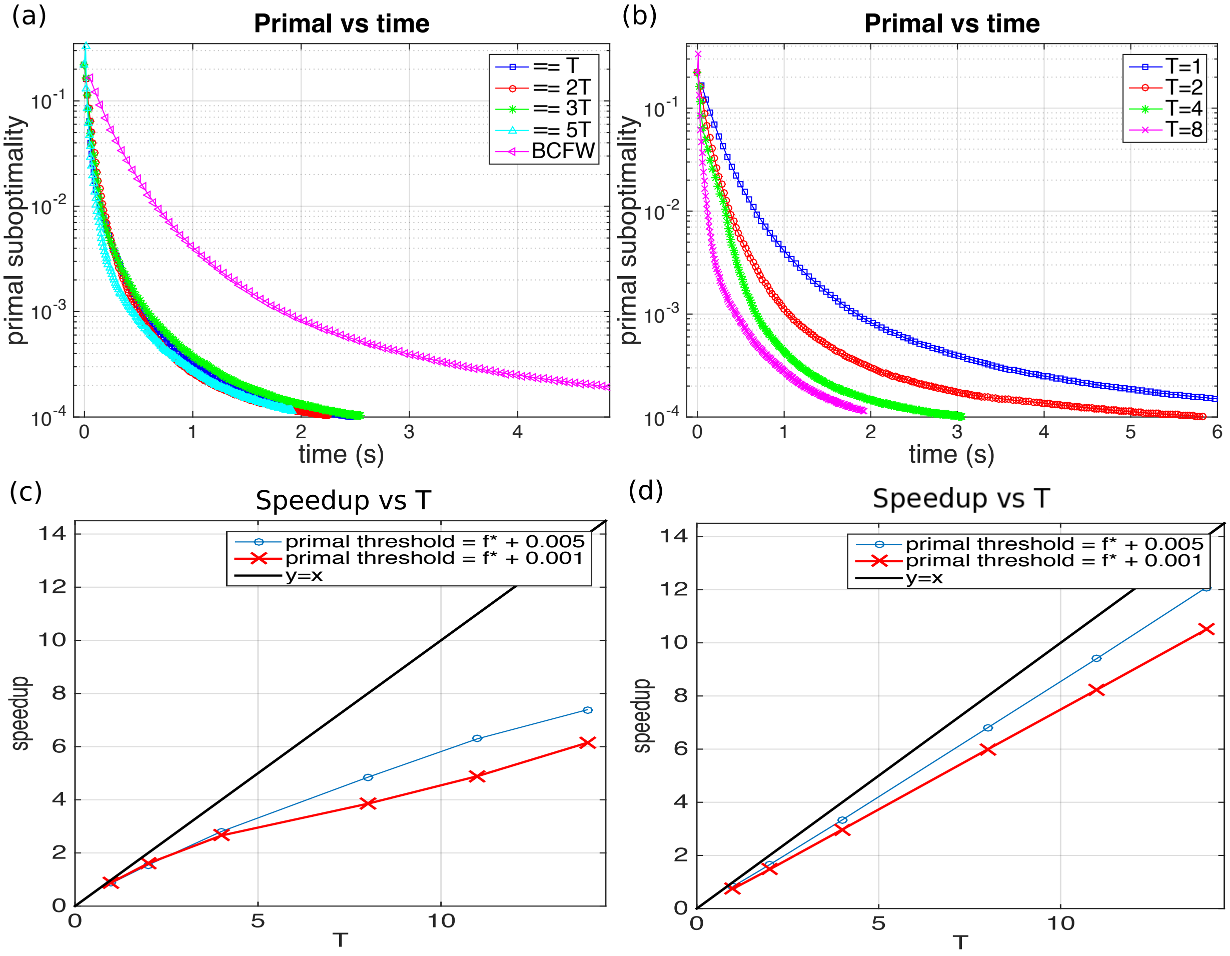


Figure 5.3: **(a)** Primal suboptimality vs wall-clock time using 8 workers ($T = 8$) and various mini-batch sizes τ. **(b)** Primal suboptimality vs wall-clock time for varying T with best τ chosen for each T separately. **(c)** Speedup via parallelization with the best τ chosen among multiples of T ($T, 2T, ...$) for each T. **(d)** The same with longer subproblems.

We use $T = 14$ workers for the experiments in this section. We first simulate the scenario with just one straggler with return probability $p \in (0, 1]$ while the other workers run at full speed ($p = 1$). Fig. 5.5(a) shows that the average time per effective datapass (over 20 passes and 5 runs) of AP-BCFW stays almost unchanged with slowdown factor $1/p$ of the straggler, whereas it increases linearly for SP-BCFW. This is because AP-BCFW relies on the average available worker processing power, while SP-BCFW is only as fast as the slowest worker.

Next, we simulate a heterogeneous environment where the workers have varying speeds. While varying a parameter $\theta \in [0, 1]$, we set $p_i = \theta + i/T$ for $i = 1, \ldots, T$. Fig. 5.5(b) shows that AP-BCFW slows down only by a factor of 1.4 compared to the no-straggler case. Assuming that the server and worker each take about half the (wall-clock) time on average per epoch, we would expect the run time to increase by 50% if the average worker speed halves, which is the case if $\theta = 0$ (i.e., $\frac{1}{\theta} \to \infty$). Thus, a factor of 1.4 is reasonable. The performance of SP-BCFW is almost

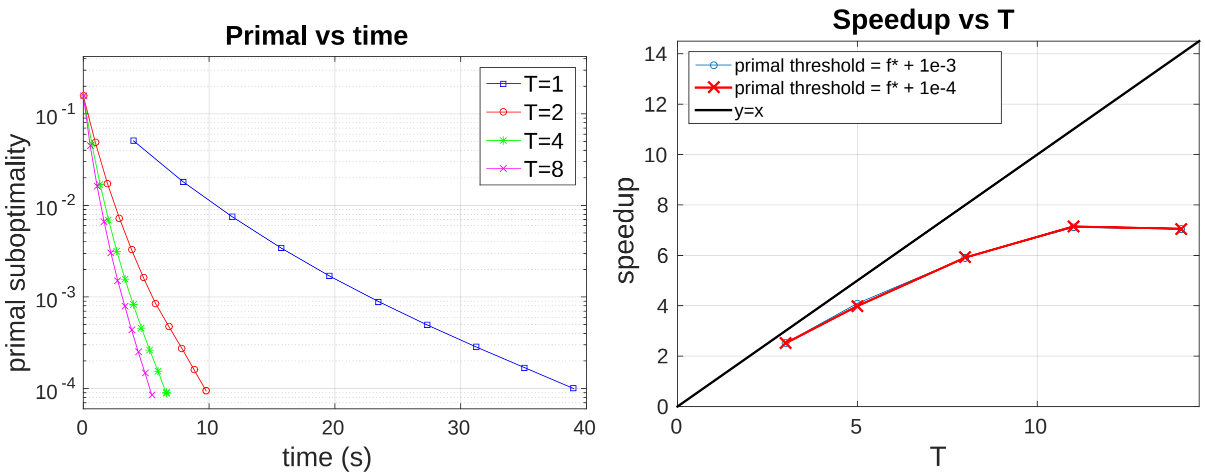


Figure 5.4: Speedup with parallelization on a synthetic OCR dataset. Left plot shows the decay of primal suboptimality and the right one shows the speedup.

identical to that in the previous experiment as its speed is determined by the slowest worker. Our experiments show that AP-BCFW is robust to stragglers and system heterogeneity.

5.4.4 Convergence under unbounded heavy-tailed delay

In this section, we illustrate the mild effect of delay on convergence by randomly drawing an independent delay variable for each worker. For simplicity, we use $\tau = 1$ (BCFW) on the group fused lasso problem from Section 5.4.1. We sample $\varkappa$ using either a Poisson distribution or a heavy-tailed Pareto distribution (round to the nearest integer). The Pareto distribution is chosen with shape parameter $\alpha = 2$ and scale parameter $x_m = \kappa/2$ such that $\mathbb{E}\varkappa = \kappa$ and $\operatorname{Var}\varkappa = \infty$. During the experiment, at iteration k, any updates that were based on a delay greater than $k/2$ are dropped (as our theory stipulates). The results are shown in Figure 5.6. Observe that for both cases, the impact of delay is rather mild. With expected delays up to 20, the algorithm only takes fewer than twice as many iterations to converge.

5.5 Additional Related Work

BCFW and Structural SVM. Our algorithm AP-BCFW extends and generalizes BCFW to parallel computation. Our analysis follows the structure in [82], but uses different stepsizes that must be carefully chosen. Our results contain BCFW as a special case. [82] primarily focus on more explicit (and stronger) guarantee for BCFW on structural SVM, while we mainly focus on a more general class of problems; the particular subroutine needed by structural SVM requires special treatment though (see Appendix D.3).

Parallelization of sequential algorithms. The idea of parallelizing sequential optimization

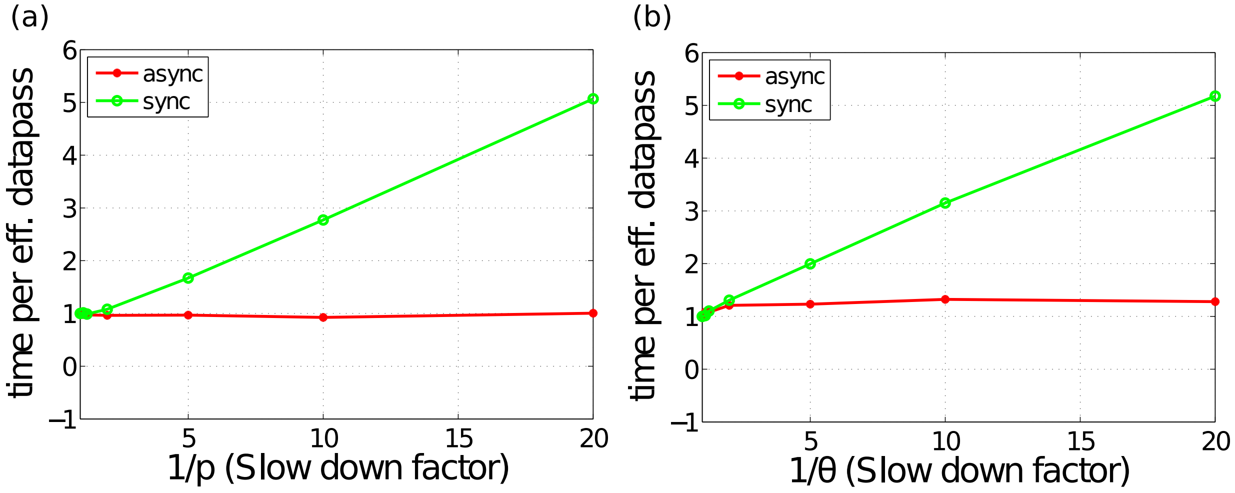


Figure 5.5: Average time per data pass in asynchronous and synchronous modes for two cases: one worker is slow with return probability p (left); workers have return probabilities (p_is) uniformly in $[\theta, 1]$ (right). Times normalized separately for AP-BCFW, SP-BCFW w.r.t. to where workers run at full speed.

algorithms is not new. It dates back to [139] for stochastic gradient methods; more recently [91, 98, 125] study parallelization of BCD. The conditions under which these parallel BCD methods succeed, e.g., expected separable overapproximation (ESO), and coordinate Lipschitz conditions, bear a close resemblance to our conditions in Section 5.3.2, but are not the same due to differences in how solutions are updated and what subproblems arise. In particular, our conditions are *affine invariant*. We provide detailed comparisons to parallel coordinate descent in Appendix D.4.5.

Asynchronous algorithms. Asynchronous algorithms that allow delayed parameter updates have been proposed earlier for stochastic gradient descent [114] and parallel BCD [98]. We propose the first asynchronous algorithm for Frank-Wolfe. Our asynchronous scheme not only permits delayed minibatch updates, but also allows the updates for coordinate blocks *within each* minibatch to have different delays. Therefore, each update may not be a solution of (5.2) for any single x. Moreover, we obtain strictly better dependence on the delay parameter than predecessors (e.g., an *exponential improvement* over [98]) possibly due to a sharper analysis.

Other related work. While preparing our manuscript, we discovered the preprint [17] which also studies distributed Frank-Wolfe. We note that [17] focuses on Lasso type problems and communication costs, and hence, is not directly comparable to our results.

5.6 Conclusion

In this chapter, we propose an asynchronous parallel generalization of the block-coordinate Frank-Wolfe method [82], analyze its convergence and provide intuitive conditions under which

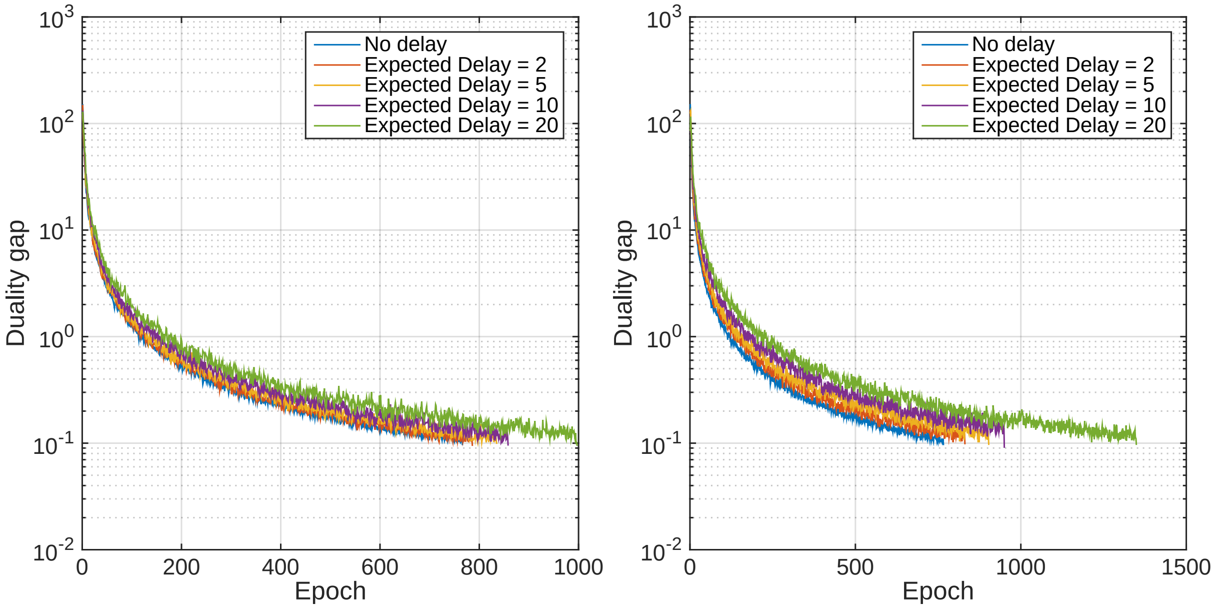


Figure 5.6: Illustrations of the convergence BCFW with delayed updates. On the left, we have the delay sampled from a Poisson distribution. The figure on the right is for delay sampled from a Pareto distribution. We run each problem until the duality gap reaches 0.1.

it has a provable speed-up over BCFW. We also show that the method is resilient to delayed updates in the distributed setting. The convergence bound depends only linearly on the expected delay and possibly sublinearly if the delay is bounded, yielding an exponential improvement over the dependence on the same parameter in parallel coordinate descent [98]. The asynchronous updates allow our method to be robust to stragglers and node failure as the speed of AP-BCFW depends on *average* worker speed instead of the slowest. We demonstrate the effectiveness of the algorithm in structural SVM and Group Fused Lasso with both controlled simulation and real-data experiments on a multi-core workstation. For the structural SVM, it leads to a speed-up over the state-of-the-art BCFW by an order of magnitude using 16 parallel processors. As a projection-free FW method, we expect our algorithm to be very competitive in large-scale constrained optimization problems, especially when projections are expensive. Future work includes analysis for the strongly convex setting, the non-convex setting and ultimately releasing a general purpose software package for practitioners to deploy in Big Data applications.

Chapter 6

Conclusion and Future Work

In this thesis, we approach learning with staleness from three inter-dependent directions:

- **Theoretical Analyses.** We extend the theoretical analyses of a number of classical ML algorithms. Our theoretical results paint the overall picture that many Machine Learning (ML) algorithms indeed work under limited staleness, under both data parallelism and model parallelism, consistent with the notion that iterative-convergent ML methods are error-tolerant [146]. In particular, (1) for stochastic gradient descent under data parallelism, we show in Chapter 3 that Stale Synchronous Parallel (SSP), a bounded staleness consistency model, achieves better convergence with lower empirical staleness. This captures the intuition that lower *run time* staleness should help convergence, even under the same maximum staleness bound. This dependency on runtime staleness was not revealed in previous analyses that only considers worst-case staleness. Additionally, our results shows that the variance is decreasing when close to optimum. (2) For proximal gradient descent, we show in Chapter 4 that it converges when parallelized over the model parameters (i.e., model parallelism). Our analyses apply to non-convex objective functions, for both the smooth part and the non-smooth regularizer term. This includes most practical ML problems in large-scale settings, such as regularized logistic regression, boosting, and deep learning models. (3) For Frank-Wolfe algorithms, we show in Chapter 5 that block-coordinate Frank-Wolfe algorithms can achieve speed-up, when block updates are executed in parallel following the model parallelism paradigm. Our analyses reveal problem-dependent quantities (i.e. the objective function and the dataset) that governs the acceleration over sequential execution. Furthermore, the parallel algorithm is only mildly dependent on expected staleness, instead of the worst-case delay.
- **Empirical Evaluation.** We conduct simulation and large-scale distributed experiments to study the empirical effect of staleness on ML algorithms under complex conditions. In Chapter 2 we perform simulation study on 5 models: Multi-class Logistic Regression, Deep Neural Networks, Matrix Factorization, Variational Auto-encoder, and Latent Dirichlet Allocation (LDA), solved by 10 algorithms, including stochastic gradient descent (SGD) and variants, variational inference, and Gibbs sampling. Each algorithm's sensitiv-

ity to staleness is highly variable. For example, among the SGD variants, methods using advanced step size tuning are generally much more susceptible to staleness. Perhaps surprisingly, limited staleness can sometimes accelerate convergence for SGD. The sensitivity to staleness is subject to the specific model, with more slowdown for more complex models (e.g., deeper DNNs). Gradient coherence–which is easy to evaluate at runtime–offers a possibly explanation for the impact of staleness. Collapsed Gibbs sampling for LDA exhibit highly non-linear response to staleness, in which staleness below a certain threshold makes virtually no impact, but precipitates rapid convergence degradation above certain threshold. And while some algorithms are more robust to staleness, no ML method is immune to the negative impact of staleness. In Chapter 4 and Chapter 5, our experiments corroborate with the theoretical analyses of proximal gradient descent and block-coordinate Frank-Wolfe algorithms, achieving convergence and speed up. We consider Group Lasso, structural SVM, and Lasso. In particular, the speed-up can be sensitive to the convergence criteria, where higher requirements on model quality can result in more limited speed-up.

- **Staleness-Minimizing Systems.** Using the insights gleaned from our empirical and theoretical analyses, we design staleness-minimizing Parameter Server systems in Chapter 3 that optimizes the synchronization mechanisms to effectively control the runtime staleness. Bösen, our Parameter Server, supports Eager Stale Synchronous Parallel (ESSP) synchronization mechanism, which is an optimized implementation of the existing Stale Synchronous Parallel (SSP), a bounded staleness consistency model. By utilizing the additional network capacity to communicate early before reaching the maximum staleness bound, Bösen effectively reduces the runtime staleness, and the profiling of runtime staleness distribution shows that most parameter reads concentrate on low staleness regime, in contrast to a lazy communication under SSP that incurs substantial high staleness parameter reads. Moreover, ESSP implementation also significantly reduces the network wait time experienced by ML applications, because most communication occurs before reaching the maximum staleness bounds and thus does not block computation. We show that our system stabilizes diverging optimization paths such as experienced by SGD on Matrix Factorization, and substantially accelerates convergence due to the lowered staleness for Gibbs sampling on LDA and SGD on MF. By using ESSP, the users are relieved of the burden to tune the staleness level, as the runtime staleness is no longer sensitive to the user-specified maximum staleness. Our experiments demonstrate similar convergence paths regardless of the staleness parameter.

Our results support the conclusion that staleness is a fundamental governing parameter for most, if not all, ML algorithms, with diverse manifested effects among the considered algorithms. Furthermore, we demonstrate that we can design systems that minimizes the effects of staleness by optimizing the synchronization mechanism.

6.1 Future Work

There are still a number of interesting open problems at the intersection of staleness, ML algorithms, and systems:

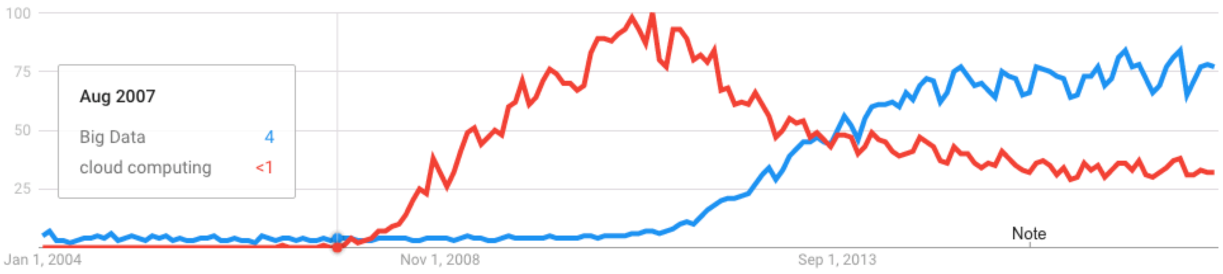
- Currently it is difficult to know a priori which models and algorithms are robust to staleness, or even how staleness would affect convergence under different configurations of models, algorithms, and network conditions. Is it possible to device an indicator that can be efficiently evaluated empirically for each problem so that the system can auto-adjust the staleness level accordingly, or even dynamically during the algorithm runtime? This would eliminate the need to have trial runs with varying staleness that may or may not lead to desired convergence. Our initial results around gradient coherence is promising, but further research is needed to generalize this concept of coherence to non-gradient methods.
- How do we theoretically characterize the convergence behaviors for sampling methods under staleness? Our results from Gibbs sampling on LDA suggest that there is a threshold such that below the threshold the staleness has virtually no effect. However, when staleness is above the threshold, the convergence fails completely. It may be interesting to understand such behavior from posterior contraction perspective [135].
- Currently we are treating the staleness of all parameters uniformly. How can we prioritize more "important" parameters to communicate over others? [145] uses absolute magnitude and relative magnitude of change as the heuristics to estimate parameter importance, which works well for LDA but not for MF. Is there a more principled way to prioritize parameters to synchronize?
- There are a new class of algorithms and statistical methods that require minimal synchronization [72, 112, 128]. Those methods generally requires only single or very few synchronization steps, in contrast to the algorithms considered in this thesis. Since communication strictly enhances the capacity of the system, it would be interesting to consider if we can blend in the two approaches. For example, is it possible that, when the staleness is high, the algorithm would become to be more similar to the embarrassingly parallel approach, but when the bandwidth is available, the algorithm can take advantage of those communication opportunity. This would be especially useful for performing ML on the edge where low power devices with extremely limited bandwidth are mixed with more powerful gateway computers.

Chapter 7

The Debate: Synchronous vs Non-Synchronous Training for Machine Learning

We end the thesis with an informal discussion examining the broader arguments against or in support of non-synchronous ML execution.

Distributed ML has garnered a substantial attention in the past 10 years. Google trend shows that “Big Data” continues to be a subject of interest into 2018, even as the interest in cloud computing subsides:



The problem of synchronous and non-synchronous training has broad implications. For one, ML computation is bound to grow substantially, with increasing industry adoption at larger scales. There will be a lot of opportunity for savings and consequently a surging demand for computational efficiency, much like what happened to the move from disk-based Hadoop computation to Spark’s in-memory computation, or the web stack moving from lower performing scripting languages (e.g. Python / Ruby on Rails) to Go and Node.js.

Another reason that non-synchronous vs synchronous is fundamental is that they result in very different system design, which can be non-trivial to alter afterwards. For example, non-synchronous systems generally have different abstraction than synchronous ones, such as restricting client-side updates to be commutative and associative, or requiring additional control on synchronization such as the staleness parameter [40, 65] or push and pull primitives [93]. Once the system gains adoption, change as fundamental as the execution model is very unlikely—it is highly nontrivial to integrate Spark, a synchronous system, with a non-synchronous abstraction. It requires significant extension to Spark's existing semantics [57]. The two synchronization paradigms also lead to divergent system optimization. For example, in non-synchronous system it is advantageous to communicate early which improves the quality of stale shared parameters [145]. In contrast, for synchronous systems, sending out updates before reaching the end of an iteration does not advance the execution whatsoever, but can actually adversely create larger network traffic leading to network congestion.

Because distributed ML involves both ML configurations like learning rate and momentum, as well as system runtime properties such as network throughput, non-synchronous training often needs extensive tuning to work. In fact, there are papers devoted to extensive parameter tuning in training algorithms to adapt to distributed settings [1, 59]. This reminds me of the development of deep learning, which for a long time is only available for those who master the "tricks" needed to get it to work. There's a reason why we have books with title *Efficient BackProp in Neural Networks: Tricks of the Trade* [89]. But over the years, due to the overwhelming amount of empirical results and years of tuning, the network architecture designs become more standardized, with much much more predictable performance. The need for "tricks" does not go away completely, but they have been largely simplified and standardized. This also does not mean that we understand everything about neural nets. In fact, as far as I know we do not yet have a good grasp of things like "internal covariate shift" that batch normalization claims to solve. But nonetheless, we got deep learning to work on a variety of supervised learning tasks, especially in image and speech, sometimes with jaw dropping performance.

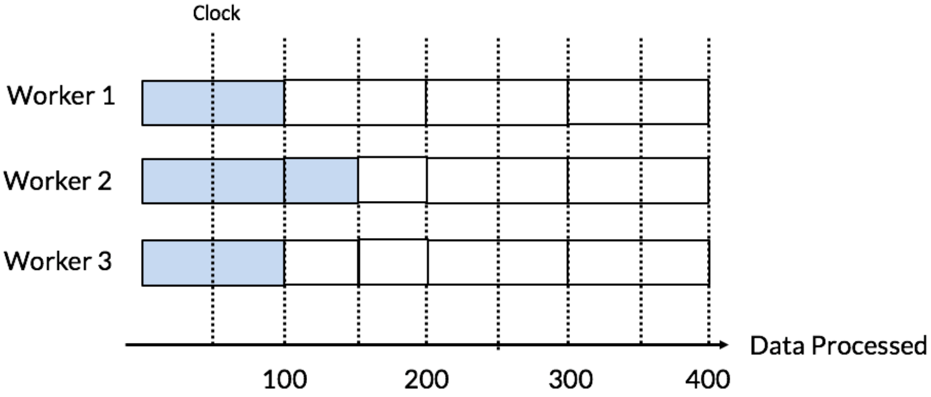
In contrast to deep learning, the progress in distributed ML world is not as satisfying. There is still fundamentally contradictory results on whether asynchrony helps or hurts the overall training speed. There are simply too many tuning knobs in a distributed execution. If ML models are already hard to reproduce, distributed ML, with myriad of less controllable hardware performance, system implementation, and interference from other concurrent workloads, makes it much more daunting to compare results across different papers. As a result, in the distributed ML world we are still very much like deep learning in 2012: we hear about high profile success stories in distributed ML, but applying distributed ML under a different circumstance can be hit or miss. There is a lack of systematic understanding of the empirical behaviors.

A goal of this discussion is to expose the factors that are often hidden or coupled together. By doing this, I hope to frame the synchronous vs non-synchronous debate more clearly. Along the way I will also touch on the many gaps in our understanding that future research may fill in.

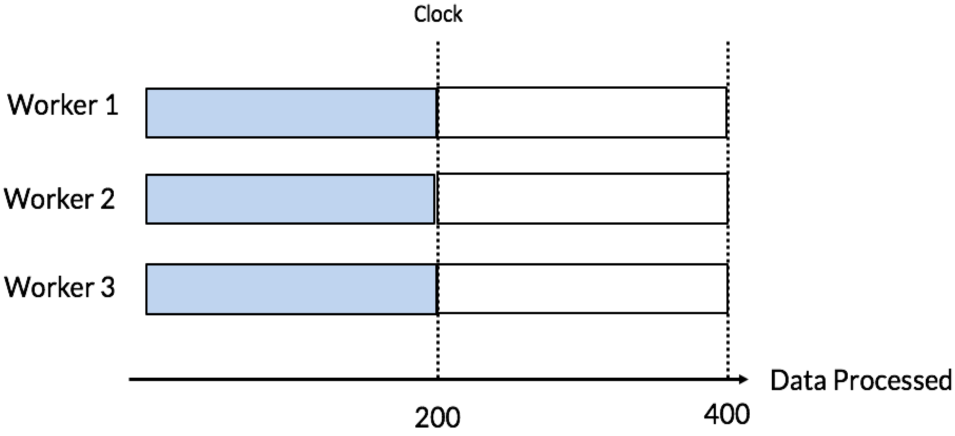
7.1 Async Isn’t Aways More Stale than Sync

A common misconception is that if a system is not synchronous, then the workers are more out of sync. In some sense this is true, because non-synchronous systems allow different model copies on the workers to be different, whereas in synchronous systems they are exactly the same. But this view overlooks a key tuning parameter, namely, the size of workload in each clock. For example, consider the following scenario: a non-synchronous system clocks at 4x the rate of a synchronous system, as measured by the number of data processed:

(a) Non-synchronous execution with high clock rate



(b) Synchronous execution with low clock rate



Now, if the non-synchronous system keeps all workers within 1 clock of each other, then which one is more out of sync? Assuming the updates are not scaled differently based on the clock rate, the workers in the non-synchronous system actually miss fewer updates than the synchronous system! To see this, simply observe that in the non-synchronous system there are always fewer pending updates not observed by the model when each data is processed. By the same reasoning, we can see that just because a system is fully asynchronous, it is not necessarily more stale than the synchronous system. Asynchronous execution simply means that the system makes best effort to synchronize, which, depending on the network and system implementation, can sometimes have very low staleness!

Exploiting Bounded Staleness [36] offers a pretty insightful comparison along this line between fully synchronous execution and stale synchronous execution. The result? Clock rate matters. The paper defines work per clock (WPC) as the number of unit workloads between each clock. For synchronous systems (slack = 0) lower clock rate (higher WPC) indeed converges much more slowly, as seen in Fig. 7.1. Notice that the x-axis is the work done, which is the same as the number of data passes (iterations). The paper then compares synchronous systems (A-BSP) with non-synchronous one (SSP):

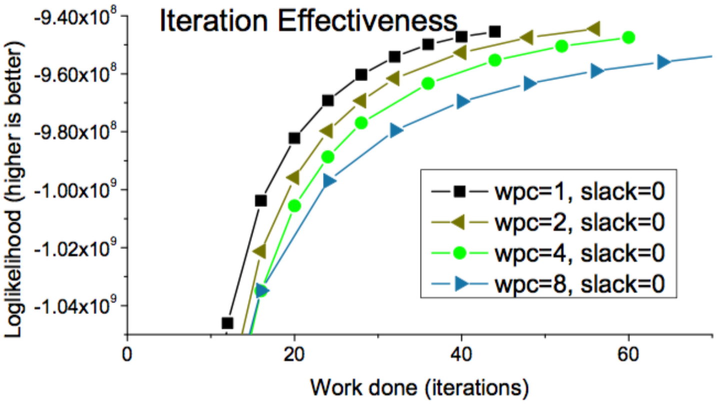


Figure 7.1: (Fig. 3 in [36]) Convergence over logical time (work done) under synchronous training of topic models.

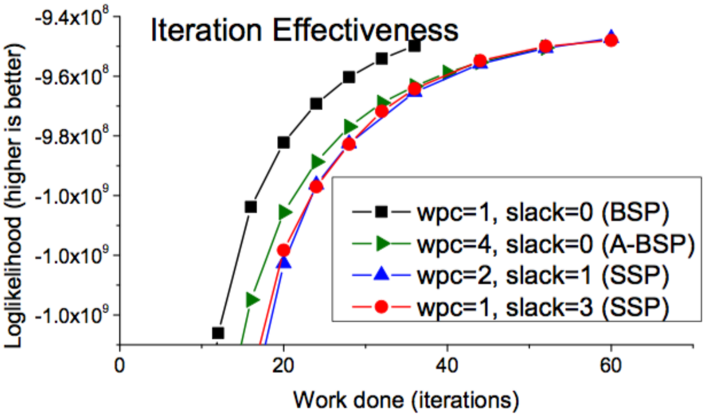


Figure 7.2: (Fig. 3 in [36]) Convergence over logical time (work done) under synchronous training (BSP, A-BSP) and non-synchronous (SSP) of topic models.

Fig. 7.2 shows a comparison between synchronous and non-synchronous training with different work per clock. The x-axis is the number of data passes (iterations). Notice that it takes roughly the same time to reach objective value -9.5×10^8 for the green curve (wpc=4, slack=0, synchronous), the blue, and the red curves (non-synchronous). In other words, by controlling the clock rate and limiting the degree of staleness, the algorithm makes similar level of progress under synchronous and non-synchronous settings per unit of computation.

The moral of the story is that distributed execution is complex and can sometimes behave in surprising ways. It is not enough to compare synchronous against non-synchronous execution. It's probably more helpful to consider the "effective staleness", which captures the actual staleness experienced during the run time, for both synchronous and non-synchronous training. If all ML parameters are kept fixed, effective staleness can be measured by:

> **Effective Staleness:** For a training instance d_i used in update u which is computed based on model x_t and is applied to model $x_{t'}$, the effective staleness for d_i is the number of data used in all the updates committed between t and t', including those in update u.

This definition can be applied to both synchronous and non-synchronous training. Note that it's important to keep the ML parameters fixed. For example, batch size in stochastic gradient descent algorithms can affect the relationship between the number of data processed and the effective staleness level, as discussed in Section 2.3.3.) Many theoretical works consider staleness similar to our definition, such as [105, 161], but for some reason this key quantity is rarely considered in empirical analyses. In fact, I'm not aware of any work besides [40, 161] that tracks any form of empirical staleness. This kind of metric is quite fundamental to understanding distributed ML systems. If you think about it, most distributed systems that support latency sensitive workloads would measure response latency [42, 84]. ML workloads clearly are latency sensitive in the sense that latency causes more stale models which slows convergence. So why shouldn't "ML latency" be properly characterized in distributed ML systems?

7.2 Computation-to-Communication Ratio

Given the lack of explicit empirical staleness in the literature, we sometimes have to resort to proxy measurements to give us a sense of the runtime staleness. One useful metric to consider is the computation-to-communication ratio. Loosely speaking, if that ratio is high, then we know that the system can process a lot of data per second (and generating updates to the model) for a given communication capacity (e.g., bandwidth), leading to higher staleness. Even though this ratio is seldom, if ever, explicitly measured, we can sometimes make reasonable guesses.

One prominent work championing asynchronous training for deep learning is Project Adam [32], which is a system optimized for CPU computation. (They divide the models so that they can fit in L3 cache, and they even build assembly kernels in order to access both row major and column major memory efficiently!) They train a standard model with 5 convolutional layers followed by 3 fully connected layers on the full ImageNet dataset which has 15 million images in 22000 categories. Using fully asynchronous execution, they train 570 billion "connections" per second in their largest experiment, which is run on 108 computers, each with 16 Intel Xeon CPU cores. Out of the 108 nodes, 88 are workers while the rest are parameter servers.

This may seem like an impressive processing power, but, on the same model and dataset, GeePS [38] achieves higher throughput than the whole 108-node cluster used in Project Adam with just 4 machines, each with a NVIDIA Tesla K20C GPU. Using 16 of those machines achieve more than 5x the total throughput of Project Adam.

It's not really a surprise that GPU cluster achieves higher throughput than CPU cluster. But assuming the network capacity is similar, using GPU cluster can seriously increase the computation-to-communication ratio.

We can dig deeper into the network capacity of these systems. GeePS nodes are connected via 40 Gbps Ethernet, with 12 Gbps measured throughput via iperf. On the other hand, each node in Project Adam has two 10 Gbps NIC and 1 Gbps interface, for a total 21 Gbps bandwidth. Adam might have slightly better network, but the two network setups seem to have pretty comparable throughput per node on paper. Moreover, both GeePS and Project Adam uses parameter server to mediate their communication, which incurs total model synchronization traffic linear in the number of worker nodes. However, Project Adam additionally performs model partition, so each worker actually only hosts 1/4 of a model, reducing the communication load. Both systems send the error vector instead of the full gradient matrix for the fully connected layers. (This massively reduces the size of communication from $O(nm)$ to $O(n+m)$ where n, m are the number of input and output dimensions, respectively, of fully connected layers.) All considered, I'd say Project Adam likely can communicate the model updates more quickly due to the model partition, although at the end of the day, it is difficult to know the effective staleness in an asynchronous system, because those systems generally forgo the control of synchronization to the system performance and arbitrary network conditions, without any measurement of the empirical staleness.

Given these contexts, we can examine their experiences with synchronous vs asynchronous training. GeePS states that "while synchronization delays can be largely eliminated, as expected, convergence is much slower with the more asynchronous models because of reduced training quality." Their result shows that much more data (2x to 3x more) need to be processed to reach the same accuracy. That completely eliminate any throughput gain with asynchrony. While 2x to 3x sounds like a lot, they are in line with our findings (see Fig. 2.5).

Project Adam, on the other hand, does not provide comparison between asyn and sync, but it shows that it converges to 29% top-1 accuracy in 10 days on 48 worker machines, compared with $\sim$24% top-1 accuracy achieved by GeePS on 16 GPU nodes. Overall, based on the throughput numbers discussed earlier, the training time seems to suggest that Project Adam does not need to process much more images to compensate for async execution. In fact, asynchronous with possibly smaller batch sizes might have helped reduce the number of images needed to be processed to reach the same test accuracy due to the momentum effect and generalization from small batch training, which we will discuss later.

So does staleness really hurt training by increasing the number of iterations to reach the same model quality? One way to understand this seemingly contradictory results from the two works is to consider the computation-to-communication ratio. GeePS with 16 GPU nodes has 5x computation power than the largest cluster setup in Project Adam, whereas Project Adam enjoys higher communication capacity relative to the size of the (partitioned) model. So overall Project Adam probably has a much lower computation-to-communication ratio. Therefore it is highly possible that Project Adam has a lower runtime staleness than GeePS even though GeePS is synchronous whereas Project Adam is async. The dichotomy of synchronous vs async simply does not capture the effective staleness that's the main driver of convergence slowdown in distributed training.

One important factor we have overlooked so far is the choice of ML algorithms. Adam uses stochastic gradient descent (SGD), while GeePS uses SGD with momentum. Staleness interacts with algorithms in strange ways, as we shall see shortly.

7.3 Staleness and Momentum

Staleness alters ML algorithms in ways that we do not yet fully understand. In general, we see that it slows down convergence as staleness introduces additional noise in during convergence. However, in certain cases moderate level of staleness actually helps, as observed in our earlier experiments (e.g. Fig. 2.2(a,b,f)).

This phenomenon can be explained by *implicit momentum*. A simple intuition is that by using the stale version of the model, which hasn't reflected the updates from other workers, the updates may attempt to correct what other unseen gradients have already corrected. To illustrate, consider this simple 1D function in Fig. 7.3. After step t, we are much closer to the optimum, and therefore the gradient at w_{t+1} is of the same direction but smaller in magnitude than that at w_t. However, if a concurrent worker does not see the update at t and still thinks that the current parameter is at w_t, then it may generate a large gradient that can be viewed as momentum. (Note that the "gradient" here is really the negative of gradient).

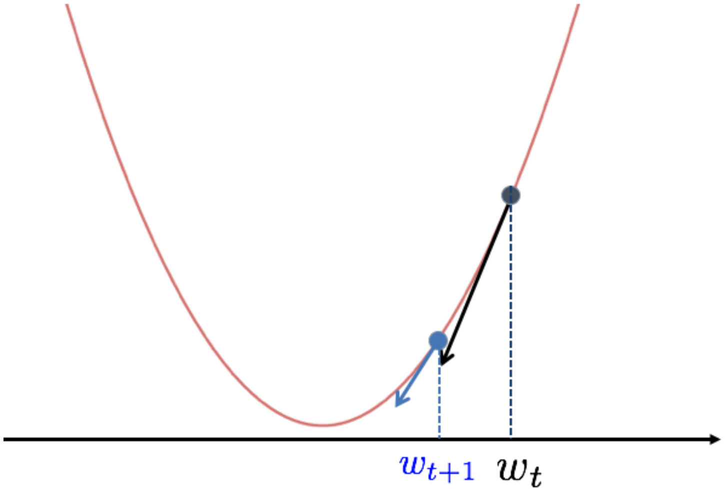


Figure 7.3: Illustration of gradient descent at two iterates.

Another way to understand implicit momentum is that because the stale models are a model from an earlier step, there is "memory" built in to any non-synchronous system. The memory is what enables most of modern SGD variants. For example, momentum SGD uses the following update:

$$w_{t+1} = w_t + \mu(w_t - w_{t-1}) - \eta_t g_t$$

where w_t is our model parameters at time step t and g_t is the gradient, and η_t is the learning rate while μ is the momentum parameter, often set to 0.9. As you can see, the momentum is computed by using the past parameter from history w_{t-1}. Other SGD variants such as Adam, RMSProps, FTRL all uses gradient history in some way.

The *Asynchrony Begets Momentum* paper [109] elegantly shows that under some assumption of the staleness distribution (namely, that the staleness is geometrically distributed), asynchronous execution basically corresponds to adding an implicit momentum term to the SGD algorithm.

They demonstrate the effect of implicit momentum empirically using a simple experiment: Since for each problem setting there is an optimal level of momentum, if asynchrony generates implicit momentum, then the optimal explicit momentum μ in the algorithm must be reduced. This is indeed what they find using CaffeNet on Cifar10 and ImageNet dataset (Fig. 7.4).

They further show, in *Omnivore* [59], that the system can tune the degree of asynchrony by measuring the optimal explicit momentum empirically. If the explicit momentum is zero, then the implicit momentum from asynchrony is likely higher than optimal, and thus they can reduce asynchrony to improve convergence.

Although the papers may lack many details of the experiments, such as the target test accuracy at convergence, or training time (especially in [109]), these works mark some of the rare cases when the degree of asynchrony (which is directly related to staleness) is more explicitly considered.

The momentum view of SGD under the staleness explains well why asynchrony can sometimes speed up the convergence for SGD algorithms. For example, in Project Adam which uses SGD without momentum, the limited asynchrony can be beneficial. On the other hand, for GeePS which uses SGD with momentum, it is possible that their momentum parameter at the default 0.9 is too high for the level of staleness, considering that the cluster consists of high throughput GPU nodes. It is actually quite fascinating to see how staleness, which is usually considered as a system parameter, can implicitly modify the ML algorithm parameters!

7.4 Staleness and the Convergence Dynamics

So far we have considered staleness as having an uniform impact throughout the convergence process. However, gradient methods have very different dynamics at the beginning of the training (when the parameter estimate is far away from the optima) vs toward the end (when the parameter estimate is close to the optima). These different dynamics interact with staleness and bring the already complex effects of staleness to the next level.

Let's start with the initial phase of optimization. When the parameter estimate is far away from the optima, for most functions this usually results in gradients with large magnitudes. For intuition, Fig. 7.3 can serve as a good (albeit simplistic) mental picture. Deep neural nets are

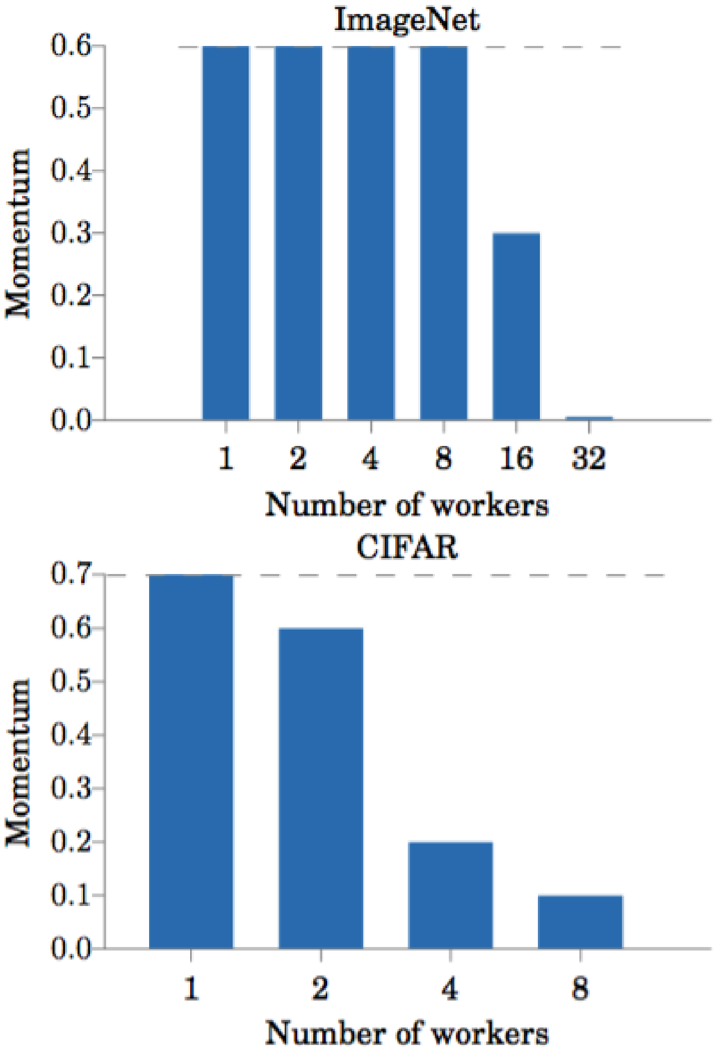


Figure 7.4: (Fig. 3 in [109]) Explicit momentum μ to achieve the best convergence result.

actually quite difficult to optimize. Back in the "old days" when good initialization schemes for deep neural nets weren't available or popularized, the initial gradient dynamics can be highly unpredictable and difficult to control [56]. A common failure mode during the initial phase of SGD were due to large gradients pushing parameters to really undesirable regimes where gradient dynamics stall or diverge. In the diverging case SGD generates even larger gradients that can severely overshoot (in the right direction). Non-synchronous execution, with the aforementioned implicit momentum or other forces that are not yet well understood, can bring back those nightmares.

Let's pause here and consider what staleness does during the initial phase of the training. If we have a function that is convex or "simple" like Fig. 7.5 (left), then slightly different verions of the model do not drastically change the direction and magnitude of the gradient. However, for more complex functions such as Fig. 7.5 (right), perturbation of the model due to staleness can potentially lead to very different gradients. Indeed, that is what we observe in Section 2.3.5 that gradient coherence along the convergence path is low for a more complex function.

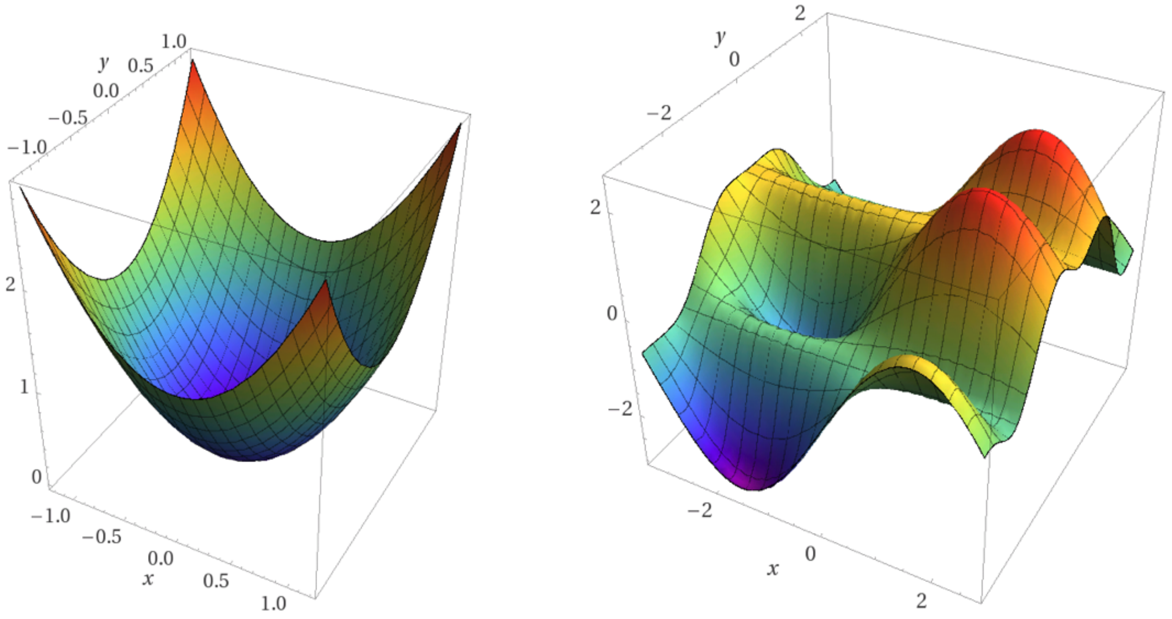


Figure 7.5

Given the challenges caused by staleness during the initial phase of convergence it's perhaps not surprising that there are quite a few warm-up strategies or tricks people use for distributed SGD. In *Revisiting distributed synchronous SGD* [31] the authors run RMSProp with momentum to optimize Inception architecture using 50–200 machines, each with a K40 GPU. Although they do not report any learning rate or momentum parameter, they empirically find that it is necessary to clip gradients by the global norm (so that the norm of the gradient does not exceed a predefined threshold) to stabilize convergence under the asynchronous setting. (They report that synchronous settings do not need gradient clipping.)

Examples of using tricks during the initial phase of asynchronous training abound. *Omnivore* [59] also trains the Inception network on ImageNet with momentum SGD under asynchronous setting. During the initial warm-up phase, they use only synchronous training until the network reaches 50% training accuracy before switching to asynchronous mode. The *Distbelief* system [43] also uses 10 hours synchronous SGD to warm start their acoustic neural net model before using asynchronous SGD ("Downpour SGD") and the distributed L-BFGS trainer ("Sandblaster"). Outside of neural nets, the *Parameter Server Consistency* paper [40] shows that matrix factorization optimized by SGD can diverge when staleness is high.

We note that these stability issues in the beginning phase of SGD training are not reported in Project Adam, possibly due to low staleness or lower learning rate (not reported). GeePS also does not report the need for warm start, though the parallelism is likely low due to well controlled staleness and possibly low number of machines (not reported in the paper).

All considered, the common practice of warm start in non-synchronous training strongly suggests that non-synchronous training is challenging during the initial phase of the SGD training.

As a side note, the optimization difficulty during the initial phase of SGD is not unique for non-synchronous training. Synchronous execution with large mini-batch sizes (which is often needed to keep many machines busy) can also be tricky during the warm-up phase. *Training ImageNet in 1 Hour* [1] trains ResNet50 on ImageNet dataset with momentum SGD under fully synchronous settings. One of their key findings is that careful learning rate schedule during the first 5 passes over the dataset is needed. Without that the final accuracy appears to be lower (though that may be corrected with more epochs of training before shrinking the learning rate by 10x, judging from Fig. 2 of [1]). Nonetheless, all of their experiments converge successfully, so it seems that synchronous training does not have as severe stability issues as the non-synchronous cases.

7.5 Non-synchronous Training Gets to an "Okay" Model Faster than Synchronous Training

Once the initial convergence dynamics is under control, non-synchronous training usually can get to regions close to the optima more quickly than synchronous ones, in terms of wall clock time. For example, in the *Revisiting distributed synchronous SGD* paper [31], we can examine the convergence curves with 50, 100, and 200 GPU workers (Fig. 7.6).

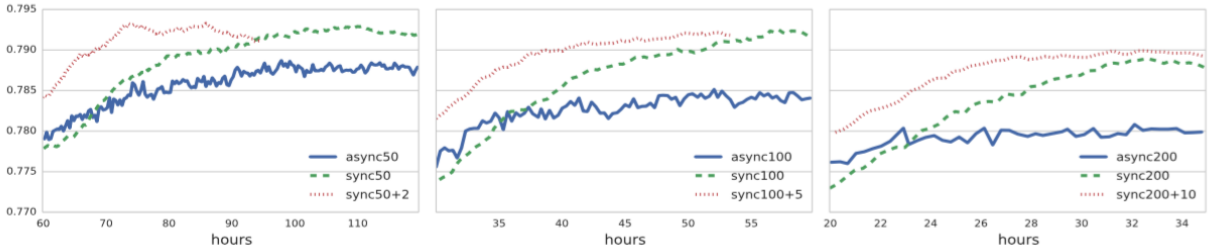


Figure 7.6: (Fig. 3 in [31]) Test accuracies with respect to training time for sync, async training on 50, 100, and 200 nodes.

The curves only capture the later part of the convergence when the model approaches optima. Notice how the blue curve (async) dominates green curves (sync) until the last part approaching high accuracy. (We ignore the red curves which use backup servers that speed up synchronous execution quite a lot in their experiment setup.)

The main reason for this is that non-synchronous training usually enjoys a much higher throughput than synchronous training, because non-synchronous training incurs lower synchronization overheads. Fig. 7.7 and Fig. 7.8 shows the breakdown between network waiting time and the compute time over varying staleness level, where higher staleness / maximal delays imply looser synchronization. We see very consistently that relaxing synchronization reduces communication wait time:

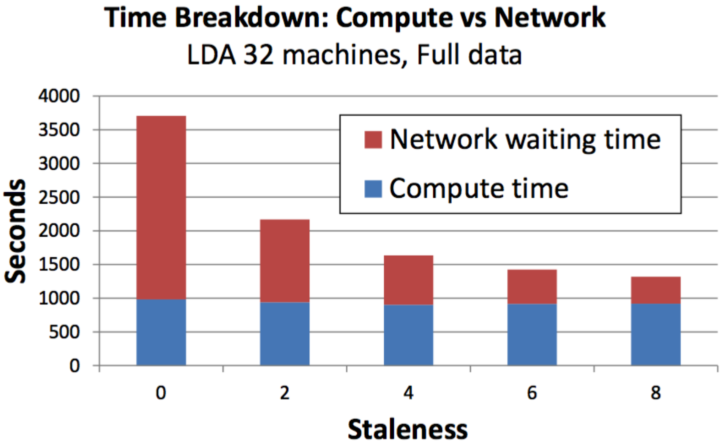


Figure 7.7: (Fig. 3 in [65]) The computation time vs network waiting time breakdown for topic model (Latent Dirichlet Allocation) optimized by collapsed Gibbs sampling under various staleness levels. The experiment runs on 32 VMs (each with 8 cores).

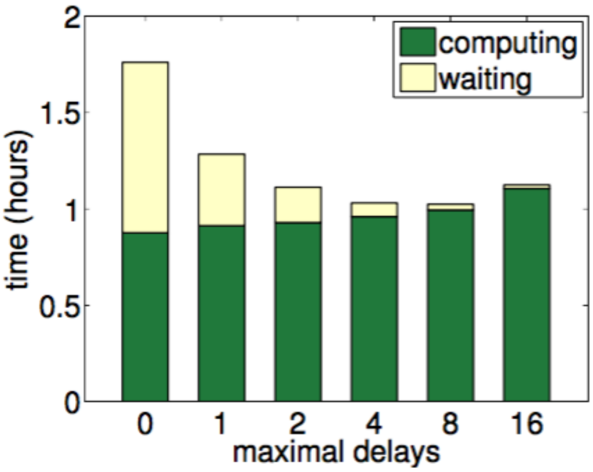


Figure 7.8: (Fig. 13 in [94]) The computation and network waiting time breakdown for sparse logistic regression optimized by block proximal gradient method under different maximal delays. The experiment runs on 1000 machines, each with 16 cores.

This is really the main reason for doing non-synchronous training—to reduce communication overheads. Fundamentally, synchronous training is bad for the network, because synchronous training utilizes network in bursty fashion, leaving network idle during the computation. By introducing non-synchronous execution, the communication can happen more or less independently from the computation (especially for a well design system like Bösen [145]), which is really what we want from the system perspective.

Before we move on, notice how the computation time increases, perhaps ever so slightly, as staleness increases, in both Fig. 7.7 and Fig. 7.8. That's because higher staleness increases the number of iterations needed to reach the same model quality. For LDA (Fig. 7.7) and logistic regression (Fig. 7.8) their response to staleness is very mild, unlike deep neural networks. Thus more staleness wins (up to some point)!

By the virtue of higher system throughput and implicit momentum, non-synchronous training is usually effective during the phase of finding the neighborhood of an optima. But to get to the highest model quality is subject to the intricate convergence dynamics at close to the optima, which we now consider.

7.6 Staleness' Effects on the Final Model Quality

Perhaps the most mysterious part of learning with staleness is how it affects the final model quality. There are a number of factors at play here. To begin with, let's consider the gradient dynamics. Generally speaking, if the gradient is high during the initial phase of training, when we are approaching the optima (or, in the case of non-convex problem, more likely, saddle points), the gradients "level off". (Imagine a river that slows down when it exits the mountains and enters expansive plains.) At close to the optima, the gradients become smaller, and more "diffused", or less coherent. To approach the optima as closely as possible, we need to fight against variance. SGD is unbiased, but it has higher variance than full batch gradient descent even in the sequential settings because we use only a small portion of the dataset to compute gradient estimates rather than the full dataset. Even in the sequential settings, there are many works on reducing the variance towards convergence [71, 121].

By adding staleness, we introduce another source of stochasticity and variance to the mix. Theoretically analyzing variance in non-synchronous settings is highly challenging, as evident in my own work [40]. But empirically we see that asynchrony indeed can slow down the convergence towards the end. For example, Fig. 7.6 shows that non-synchronous training results in lower accuracy. *Revisiting distributed synchronous SGD* [31] demonstrates that for both synchronous and asynchronous settings the accuracy degrades to some degree with the increasing number of workers / parallelism (Fig. 7.9).

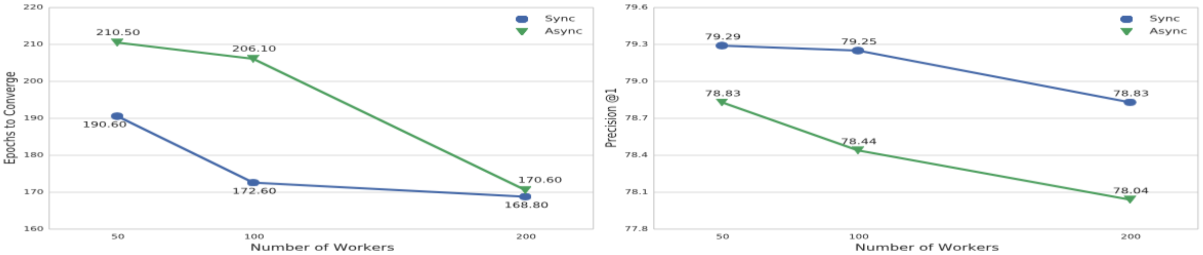


Figure 7.9: (Fig. 1 in [31]) The number of epochs (left) and test accuracy (right) of Inception model trained on ImageNet by varying numbers of workers (x-axis).

Beyond SGD, parallelism indeed makes convergence to high precision more challenging. For example, our work [143] applies parallel block coordinate Frank Wolfe methods to structural

SVM on an OCR dataset. The speed up is quite sensitive to the convergence criteria (Fig. 7.10). Speedup decreases when the convergence criteria becomes more stringent, limiting the degree of parallelism.

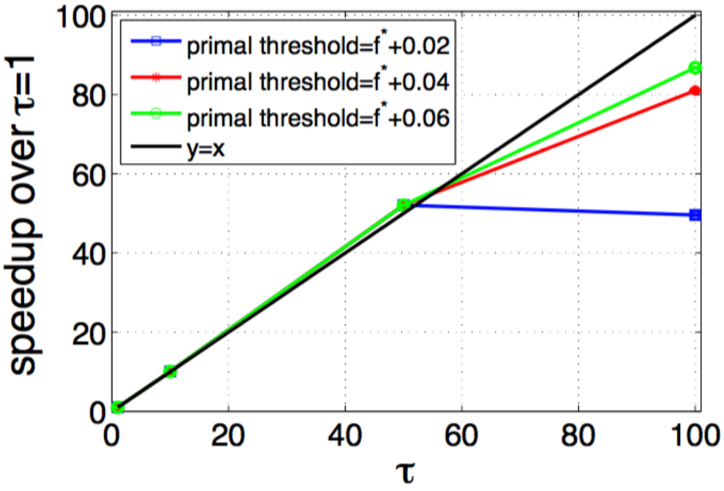


Figure 7.10: Fig. 5.2. Parallel block coordinate Frank Wolfe method applied to structural SVM on OCR dataset shows different speedup with different convergence threshold. When primal threshold is stringent (e.g., the blue curve), the algorithm does not scale beyond 50 parallel updates (x-axis).

It may seem that staleness is bad if the goal is to get a high quality model. However, the story isn't that simple. In some cases staleness actually improves the generalization ability of the resulting models.

Project Adam is a prominent example. It shows that async training not only attains the performance of models trained by synchronous updates, but can actually surpass it! They demonstrate it on MNIST dataset, stating that "we believe that our accuracy improvement arises from the asynchrony in Adam which adds a form of stochastic noise while training that helps the models generalize better when presented with unseen data. In addition, it is possible that the asynchrony helps the model escape from unstable local minima to potentially find a better local minimum." This statement presents a few puzzles.

The first part of the statement seems to refer to the regularizing effects of staleness. The basic idea is that because staleness introduces an additional source of variance, as we explained earlier, staleness prevents the optimization to achieve lower training loss and therefore prevents overfitting the training data.

To see how this can possibly help, we must point out that regularization has been a key factor in deep neural nets' success. For example, the famous dropout is an algorithmic procedure widely used to prevent overfitting, mostly by the massive fully connected layers [133, 141]. Similarly, batch normalization [67] also regularizes neural nets and helps generalization [41]. All these are in addition to the common L1, L2 regularization used to reign in neural nets.

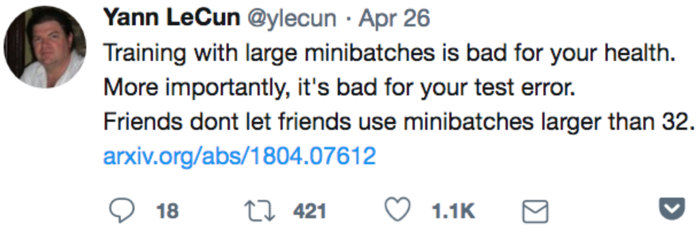
So can the variance introduced by staleness actually help improve generalization? Possibly, if we have just the right amount of staleness. All regularization needs tuning to perform well, and it is not hard to imagine that very high staleness can lead to models that are next to useless. It's very likely that Project Adam has just the right amount of staleness to get it to work well. Whether this implicit regularization is truly there and can be leveraged consistently for other models remains to be seen. Perhaps there are more literature out there supporting this point that I'm not aware of. But in any case, it would take more evidence to shed light on the regularizing effects of staleness.

Another thing is that it is not clear whether the better generalization performance in Project Adam is due to staleness' regularizing effect, or it finding a better minima, which leads us to the second part of Project Adam's statement.

The second part of Project Adam's statement is that asynchrony helps escape worse local minima. Again, to my knowledge the literature around this aspect of asynchronous training is quite sparse. However, this is related to a larger topic that is currently hotly debated: small vs large batch training.

The batch size debate stems from the need to increase parallelism efficiently. Larger batches allow more GPUs or machines to process in parallel, whereas smaller batch training usually causes higher overheads with parallel training.

Those on the side of small mini-batch argues that models trained by smaller mini-batches generalize better to test dataset than those trained by large batches (Fig. 7.11)



Yann LeCun @ylecun · Apr 26

Training with large minibatches is bad for your health.
More importantly, it's bad for your test error.
Friends dont let friends use minibatches larger than 32.
arxiv.org/abs/1804.07612

Figure 7.11: Twitter feed captured on May 1, 2018.

In this camp, *Generalization Gap and Sharp Minima* [73] offers a good characterization of the optima reached by large batches. It shows that both small and large batch optimization can reach similar training accuracy, implying that it is not the optimization challenge that causes generalization gap. However, the curvatures around the optima discovered by small batch and large batch methods are very different. Larger batches lead to the so-called "sharp minima", i.e., certain directions around the minima increase very quickly. In contrast, the small batch optimization can generally escape those minima and discover "flat minima" around which the functional surface is flat everywhere. This is because small batch has higher variance under the linear scaling of learning rate [66]. This is linked to the oft-observed phenomenon that small minibatch generalizes better [89, 103]. Fig. 7.12 gives an intuitive view of how sharp minima hurts the solution's generalization capability.

On the other side of the debate, *Training ImageNet in 1 Hour* [1] provides extensive experiments to demonstrate that with careful warm start and other tricks, large batch training can attain gen-

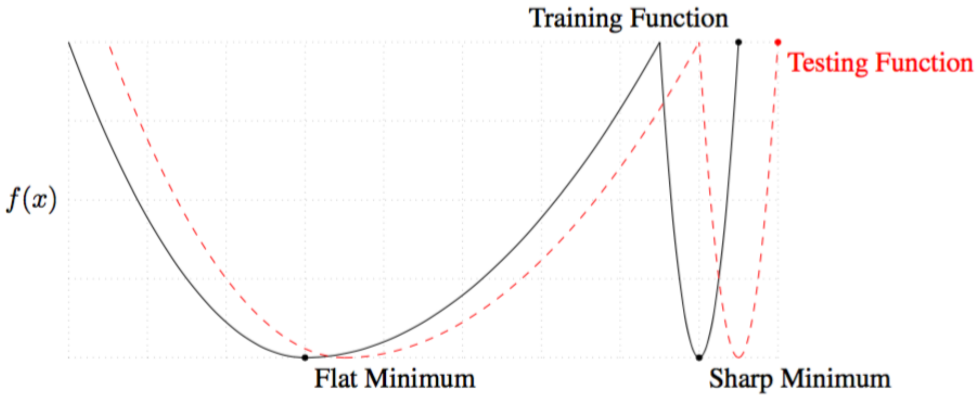
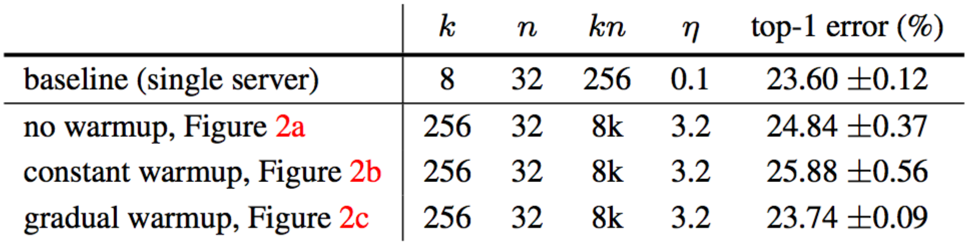


Figure 7.12: (Fig. 1 of [73])

eralization error very close to that of small batch training [1], as shown in Fig. 7.13, although ResNet-50 is the only model they train.



	k	n	kn	η	top-1 error (%)
baseline (single server)	8	32	256	0.1	23.60 ±0.12
no warmup, Figure 2a	256	32	8k	3.2	24.84 ±0.37
constant warmup, Figure 2b	256	32	8k	3.2	25.88 ±0.56
gradual warmup, Figure 2c	256	32	8k	3.2	23.74 ±0.09

Figure 7.13: (Table 1 of [1]) Validation error on ImageNet using ResNet-50. kn denotes mini-batch size, ranging from 256 (small batch) to large batch (8k). Notice that with gradual warm-up the error rate is very close to that of the small batch result.

Even though the jury is still out for the small vs large batch size debate, a key concept that emerges from those works is that stochastic noise in highly non-convex problem works in very different ways than in the traditional convex settings. In convex optimization stochastic noise is a nuisance that slows down convergence because we “cheap out” on computing the full gradient carefully. However, in the non-convex world, the objective function is fraught with many local optima and critical points which have similar objective values [33, 132], but with very different curvatures around them. Therefore stochastic noise can actually be helpful in escaping those sharp critical points.

This might be the reason that non-synchronous training leads to better generalization than synchronous ones in the case of Project Adam vs GeePS. If it were true that larger batch size in fact leads to worse generalization, non-synchronous training would be a viable way to keep the batch size small. Namely, by treating each worker’s update as independent (small) batches, we avoid the need to aggregate gradients cluster-wide to form batches of size linear in the number of workers. This immediately introduces two sources of stochasticity: one from the smaller batch size that exhibit higher variance than larger batches, and the other from asynchrony. Both may be proven useful under the right tuning and a proper understanding of the underlying staleness model of the system.

All these seem to suggest that noisy gradient (from small batches) and stable gradients (from large batches) really are solving two different problems, and might be able to work in concert. The small batch gradients help escape the poor critical points surrounded by sharp curvatures, while large batch gradients can help find the best solution once we reach the neighborhood of "flat surface" around a (good) critical point. That is indeed what *Generalization gap and sharp minima* [73] shows: large batch training can improve small batch results once the parameter estimates are in the good areas (Fig. 7.14).

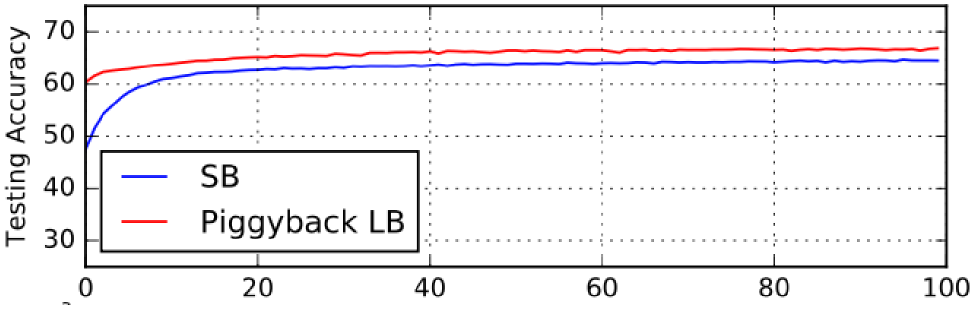


Figure 7.14: (Fig. 5 of [73]) Test accuracy of small batch (SB) method and large batch (LB) method that is warm started ("piggybacked") from the SB estimates at each of the 100 SB epochs (x-axis).

How does that impact the design of distributed training? Perhaps the level of asynchrony should not be static throughout the training, but we can allow more conservative and synchronous training at certain stages of the optimization (e.g., the beginning and the end), while keeping the middle part asynchronous? Is synchronous vs non-synchronous even the right division?

7.7 Looking to The Future

Despite all the challenges, it is clear that distributed large-scale training is here to stay. The model complexity and data sizes lead to computational demands that outpace the capacity of CPU and GPU alike, and building distributed systems is generally a cheaper and more accessible option than specialized supercomputers. The industry has already widely adopted distributed training, both synchronous and non-synchronous [1, 29, 105]. To go from here, I'd like to highlight a number of directions that can improve the way we understand and leverage non-synchronous training in distributed ML systems.

- **Treat staleness as a continuum, not a binary synchronous vs non-synchronous divide.** As ML computation grows, so will the need to make it more efficient. The quest for large scale computing will continue to demand staleness control, whether sync or non-sync. As discussed earlier, very large batch sizes, even if it's synchronous training, still introduces staleness that may adversely impact the trained models, such as their generalization capability. On the other hand, non-synchronous training has substantial system advantages, and opens up a whole world of possibilities on staleness control. And there are evidences to believe that non-synchronous training is indeed better on some models or under some convergence regime. ML training generally responds to parameter changes smoothly, such as batch size and learning rate. The evidence in this thesis and other works [59, 65] suggest that ML algorithms' responses to staleness are usually smooth, and thus we have a good reason to treat staleness as a continuum, not a binary decision. Creating a more general notion of staleness that applies to both synchronous and asynchronous training would be a good step in this direction.

- **Better understanding of staleness' effects.** The community needs to move beyond understanding model performance under some unknown staleness distribution based on whatever the underlying system and hardware we happen to use. Either we instrument the system to monitor the runtime staleness distribution explicitly, or we explicitly control the underlying staleness (via system design or simulation). Without decoupling algorithmic behavior from system properties, we will continue to have more folklore than scientific understanding. This emphasis on empirical evidence is paramount when the convergence is difficult to analyze theoretically or to get a tight bound under asynchrony. The distributed ML community can take a page from the deep learning development that has turned neural networks into an ML staple even though we do not have as much theoretical understanding as many of us would like.

- **Design systems to satisfy staleness level.** Most non-synchronous systems do not maintain any control on staleness nor even monitor it [43, 59]. Just like how key-value stores are designed to minimize various delay bound (e.g., 99% tail latency), distributed ML training should be designed to satisfy certain latency bounds, deterministically or probabilistically [13]. Together with better understanding of ML convergence under staleness, this will enable much more predictable ML training under staleness.

- **Better Programming Interface.** Spark's [154] initial success in industry was perhaps more due to its clean and interactive interface than its in-memory performance. Before Spark came along part of the Hadoop community already moved to Scalding [30], which is a vast improvement over Hadoop's verbose syntax. Compared with Scalding, Spark further consolidates the multiple tasks into a single driver program that makes it easy to program much more complex MapReduce stages. If the non-synchronous engines can offer an expressive and well-defined high level programming model / interface for users to express both synchronous and non-synchronous workloads, it would gain adoption more easily especially with the promise of performance gains (that are hopefully relatively predictable with the aforementioned efforts). Many ML researchers have already developed their algorithms on Spark, even though Spark incurs substantial overheads [108]. A good interface and implementation of non-synchronous system would go a long way in serving the large-scale ML research community.

For now, non-synchronous ML training on large-scale distributed system is still difficult to use for most ML practitioners. But perhaps in a few years' time the community will be able to get much better understanding of staleness' impact on convergence, and offer ML practitioners better systems that monitor and mitigate those impacts.

Appendices

Appendix A

Appendix for Chapter 2

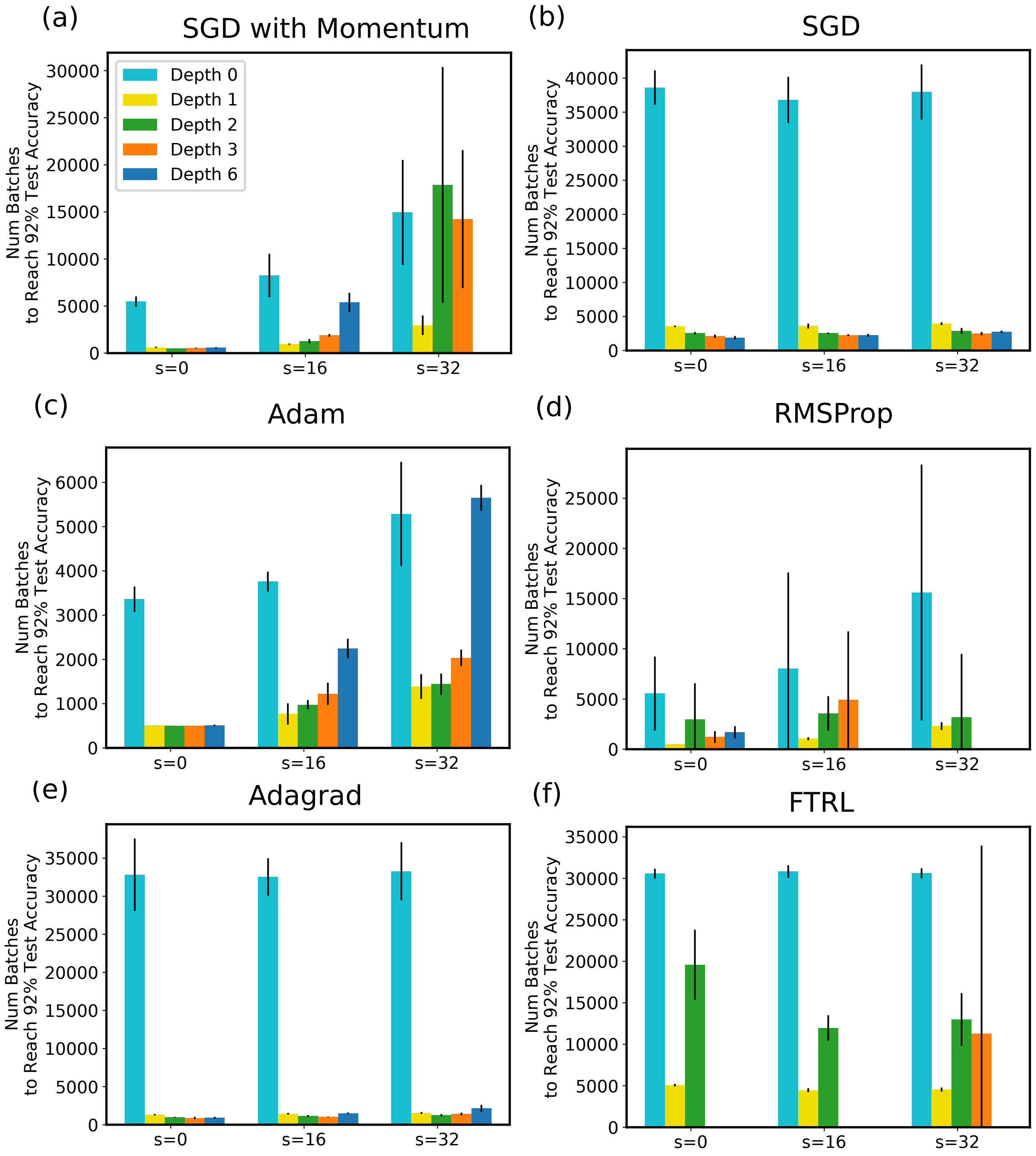


Figure A.1: The number of batches to reach 92% test accuracy using Deep Neural Networks with varying numbers of hidden layers using 1 worker. We consider several variants of SGD algorithms (a)-(e). Note that with depth 0 the model reduces to multi-class logistic regression (MLR), which is convex. MLR generally takes many more batches to converge because 92% test accuracy is close to the limit of the model performance, whereas deeper models can easily achieve 95-98% test accuracy. The error bars represent 1 standard deviation, computed from 5 randomized runs.

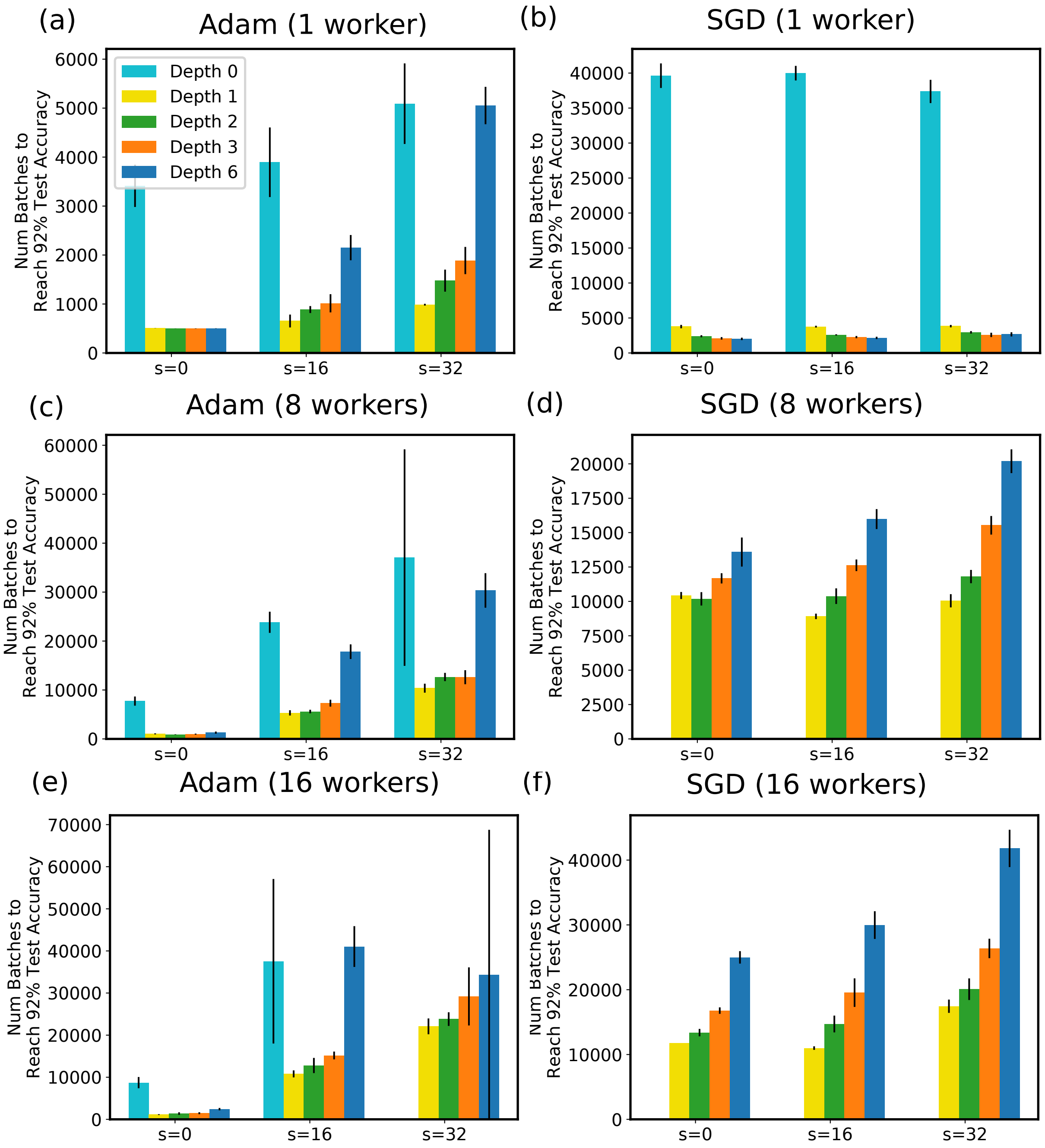


Figure A.2: The number of batches to reach 92% test accuracy for Adam and SGD on 1, 8, 16 workers with varying staleness. The error bars represent 1 standard deviation based on 5 randomized runs. Depth 0 under SGD with 8 and 16 workers do not converge within the experiment horizon (77824 batches) and is thus not shown.

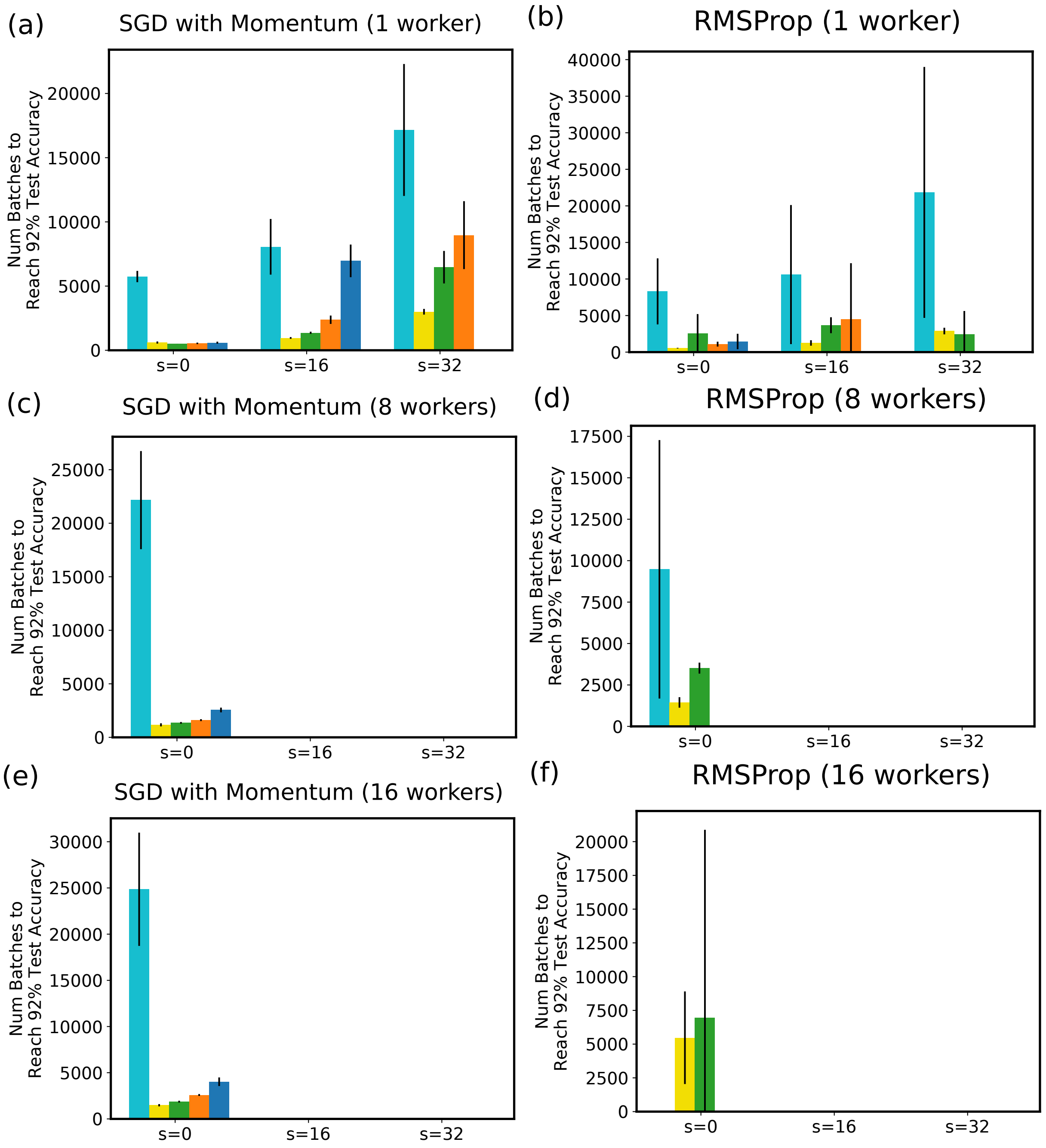


Figure A.3: The number of batches to reach 92% test accuracy for SGD with momentum and RMSProp on 1, 8, 16 workers with varying staleness. The error bars represent 1 standard deviation based on 5 randomized runs. Both optimizers did not reach 92% test accuracy for all runs with staleness 16 and 32 on worker 8 and 16 within the experiment horizon (77824 batches), and thus are not shown.

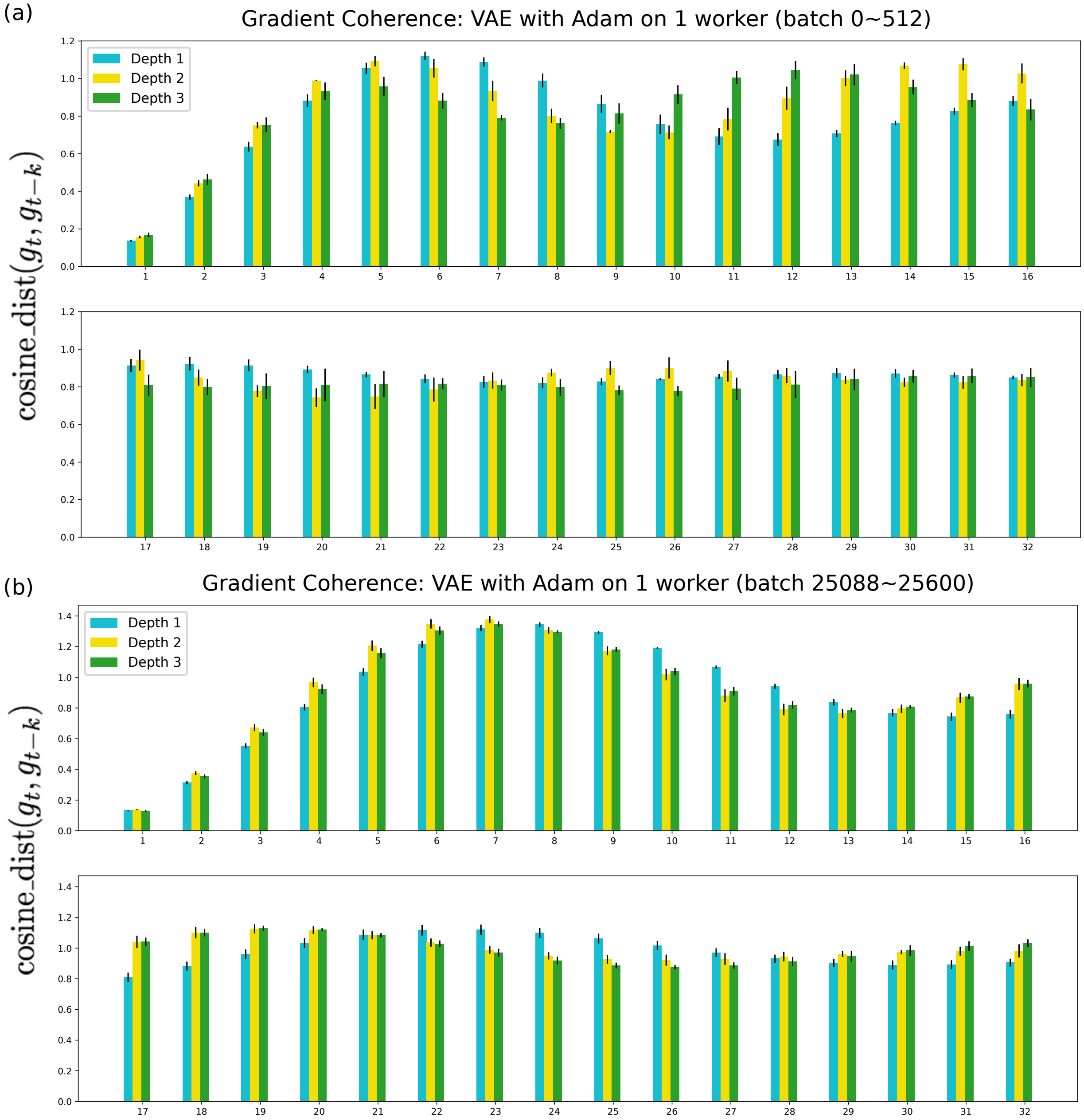


Figure A.4: Gradient coherence for VAEs with varying depths (depth 1∼3) optimized by Adam optimization using 1 worker with no staleness ($s = 0$). The x-axis is $k = 1, ..., 32$. Here we show cosine distance up to 32 batches back. (a) is a snapshot taken from the first 512 batches, while (b) is taken from 512 batches starting from batch 25088 after algorithms have converged. The error bars around the means represent 1 standard deviation computed from 5 randomized runs.

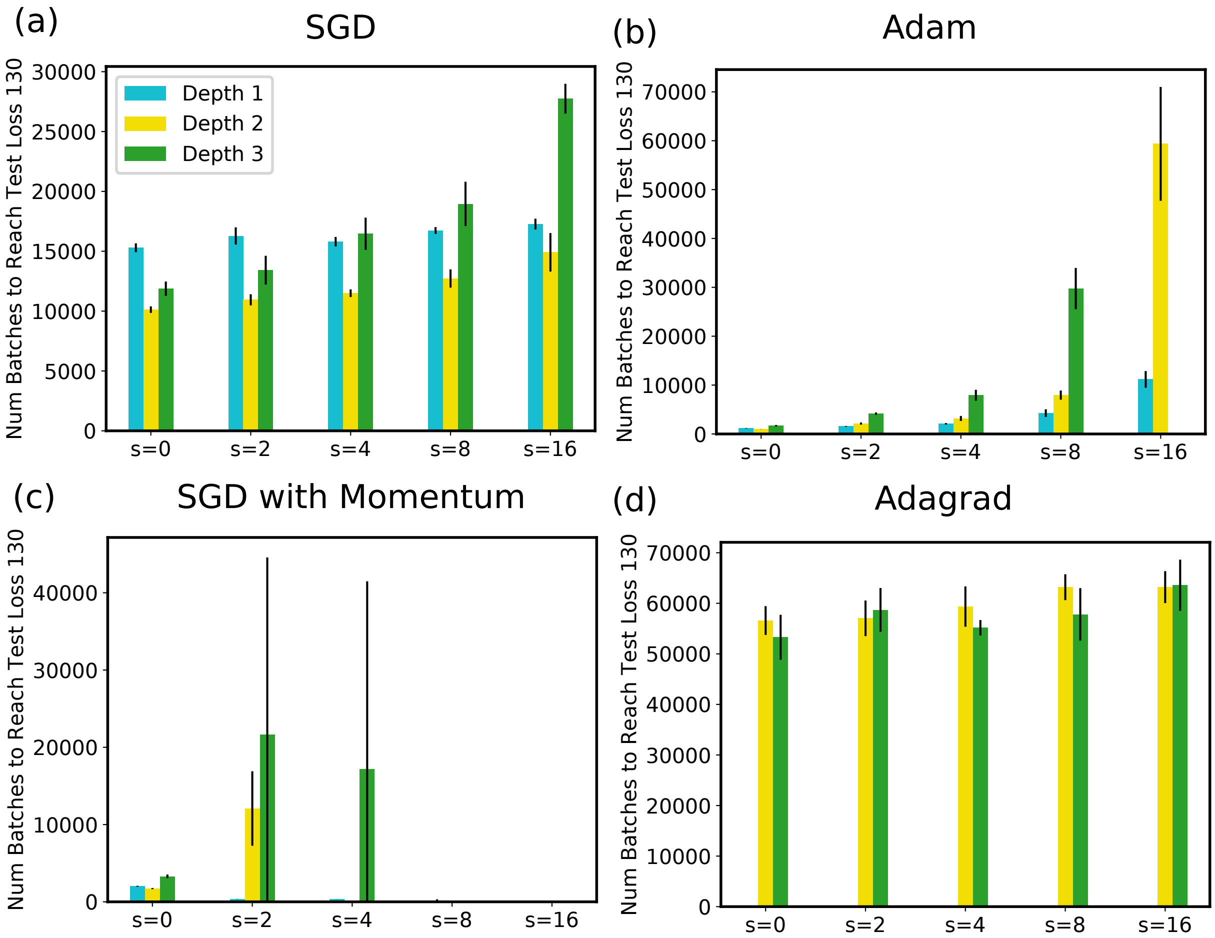


Figure A.5: The number of batches to reach test loss 130 by Variational Autoencoders (VAEs) on 1 worker, under staleness 0 to 16. We consider VAE with depth 1, 2, and 3 (the number of layers in the encoder and the decoder networks). Configurations that do not converge to the desired test loss are omitted, such as Adam optimization for VAEs with depth 3 and $s = 16$.

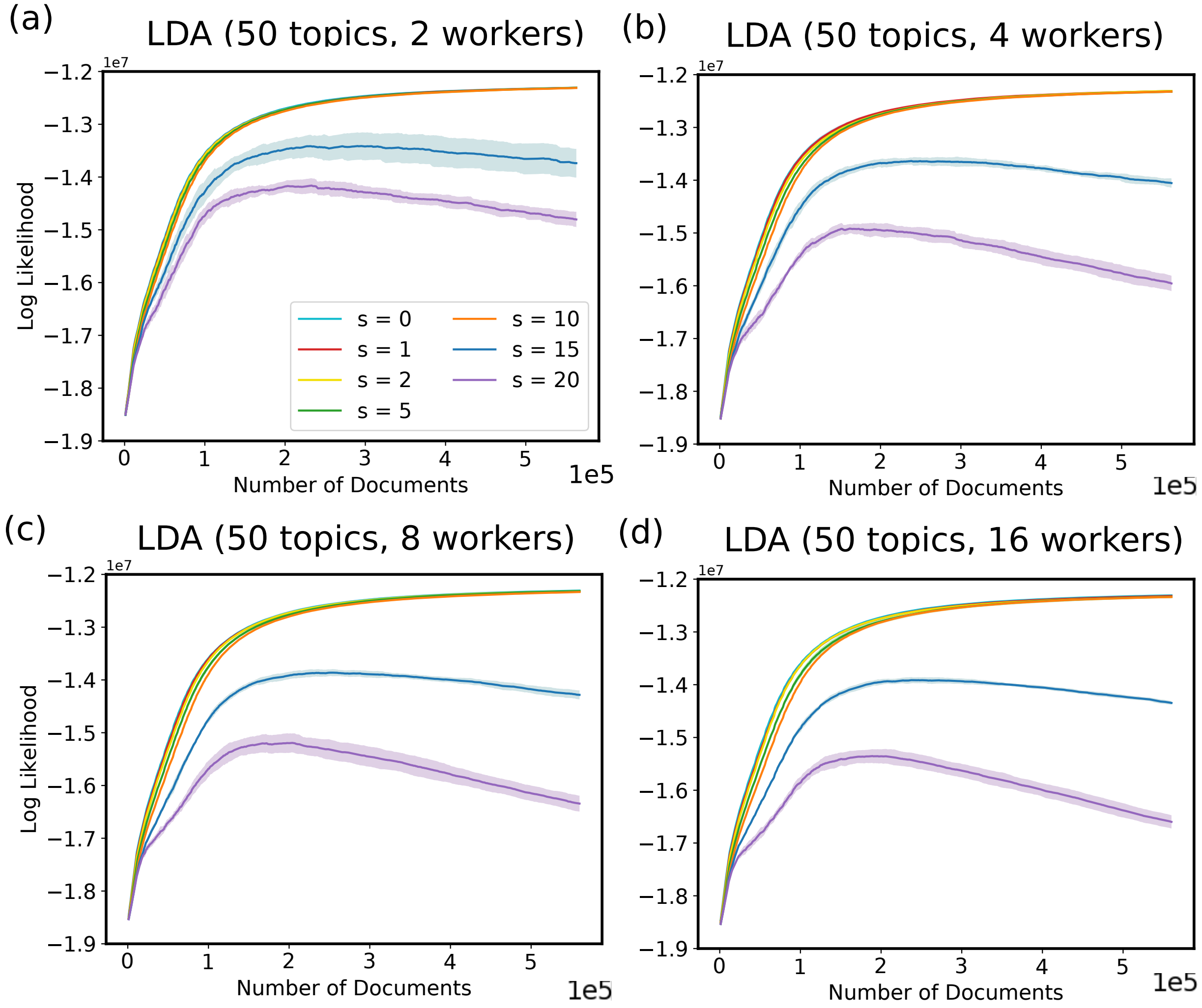


Figure A.6: Convergence of LDA log likelihood using 50 topics with respect to the number of documents processed by Gibbs sampling, with varying staleness and the number of workers. The shaded regions are 1 standard deviation around the means (curves) based on 5 randomized runs.

Appendix B

Appendix for Chapter 3

B.1 Proof of Theorem 3.1

Theorem 3.1 (SGD under VAP, convergence in expectation)**.** *Given convex function* $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$ *such that components* f_t *are also convex. We search for minimizer* $\boldsymbol{x}^*$ *via stochastic gradient descent on each component* ∇f_t *with step-size* $\breve{\eta}_t$ *close to* $\eta_t = \frac{\eta}{\sqrt{t}}$ *such that the update* $\hat{\boldsymbol{u}}_t = -\breve{\eta}_t \nabla f_t(\breve{\boldsymbol{x}}_t)$ *is computed on noisy view* $\breve{\boldsymbol{x}}_t$*. The VAP bound follows the decreasing* v_t *described above. Under suitable conditions (*f_t *are L-Lipschitz and bounded diameter* $D(x\|x') \leq F^2$*),*

$$R[X] := \sum_{t=1}^{T} f_t(\breve{\boldsymbol{x}}_t) - f(\boldsymbol{x}^*) = \mathcal{O}(\sqrt{T})$$

and thus $\frac{R[X]}{T} \to 0$ *as* $T \to \infty$*.*

Proof. We will use real-time sequence $\hat{\mathbf{x}}_t$ defined by

$$\hat{\mathbf{x}}_t := \mathbf{x}_0 + \sum_{t'=1}^{t} \hat{\mathbf{u}}_{t'}$$

$$\begin{aligned}
R[X] &= \sum_{t=1}^{T} f_t(\breve{\mathbf{x}}_t) - f(\mathbf{x}^*) \\
&\leq \sum_{t=1}^{T} \langle \nabla f_t(\breve{\mathbf{x}}_t), \breve{\mathbf{x}}_t - \mathbf{x}^* \rangle && (f_t \text{ are convex}) \\
&= \sum_{t=1}^{T} \langle \breve{\boldsymbol{g}}_t, \breve{\mathbf{x}}_t - \mathbf{x}^* \rangle
\end{aligned}$$

where $\breve{\boldsymbol{g}}_t := \nabla f_t(\breve{\mathbf{x}}_t)$. From Lemma A.1 below we have

$$R[X] \leq \sum_{t=1}^{T} \frac{1}{2}\breve{\eta}_t ||\breve{\boldsymbol{g}}_t||^2 + \frac{D(\mathbf{x}^*||\hat{\mathbf{x}}_t) - D(\mathbf{x}^*||\hat{\mathbf{x}}_{t+1})}{\breve{\eta}_t} + \langle \breve{\mathbf{x}}_t - \hat{\mathbf{x}}_t, \breve{\boldsymbol{g}}_t \rangle$$

We now bound each term:

$$\begin{aligned}
\sum_{t=1}^{T} \frac{1}{2}\breve{\eta}_t ||\breve{\boldsymbol{g}}_t||^2 &\leq \sum_{t=1}^{T} \frac{1}{2}\breve{\eta}_t L^2 && \text{(Lipschitz assumption)} \\
&= \sum_{t=r+1}^{T} \frac{1}{2}\frac{\eta}{\sqrt{t-r}} L^2 + const && \text{($r > 0$ is the finite clock drift in VAP)} \\
&= \frac{1}{2}\eta L^2 \sum_{t=r+1}^{T} \frac{1}{\sqrt{t-r}} + const \\
&\leq \frac{1}{2}\eta L^2 \int_{t=r+1}^{T} \frac{1}{\sqrt{t-r}} dt + const \\
&\leq \frac{1}{2}\eta L^2 (\sqrt{T-r} - 1) + const \\
&= \mathcal{O}(\sqrt{T})
\end{aligned}$$

where the clock drift comes from the fact that $\breve{\eta}_t$ is not exactly $\eta_t = \frac{\eta}{\sqrt{t}}$ in VAP.

$$\begin{aligned}
\sum_{t=1}^{T} \frac{D(\mathbf{x}^*||\hat{\mathbf{x}}_t) - D(\mathbf{x}^*||\hat{\mathbf{x}}_{t+1})}{\breve{\eta}_t} &= \frac{D(\mathbf{x}^*||\hat{\mathbf{x}}_1)}{\breve{\eta}_1} - \frac{D(\mathbf{x}^*||\hat{\mathbf{x}}_{T+1})}{\breve{\eta}_T} + \sum_{t=2}^{T} \left[D(\mathbf{x}^*||\hat{\mathbf{x}}_t) \left(\frac{1}{\breve{\eta}_t} - \frac{1}{\breve{\eta}_{t-1}} \right) \right] \\
&\leq \frac{F^2}{\eta} + 0 + \frac{F^2}{\eta} \sum_{t=2}^{T} \left[\sqrt{t-k} - \sqrt{t-r} \right] \quad \text{(clock drift)} \\
&\leq \frac{F^2}{\eta} + \frac{F^2}{\eta} \int_{t=\max(k,r)}^{T} \left(\sqrt{t-k} - \sqrt{t-r} \right) dt + const \\
&= \frac{F^2}{\eta} + \frac{F^2}{\eta} \left[(t-k)^{3/2} - (t-r)^{3/2} \right]_{max(k,r)}^{T} + const \\
&= \frac{F^2}{\eta} + \frac{F^2}{\eta} \left[(T-k)^{3/2} - (T-r)^{3/2} \right] + const \\
&= \frac{F^2}{\eta} + \frac{F^2}{\eta} \left[\left(T^{\frac{3}{2}} + \frac{3}{2}kT^{\frac{1}{2}} + \mathcal{O}(\sqrt{T}) \right) \right. \\
&\quad \left. - \left(T^{\frac{3}{2}} + \frac{3}{2}rT^{\frac{1}{2}} + \mathcal{O}(\sqrt{T}) \right) \right] + const \quad \text{(binomial expansion)} \\
&= \mathcal{O}(\sqrt{T})
\end{aligned}$$

$$
\begin{aligned}
\sum_{t=1}^{T} \langle \breve{\mathbf{x}}_t - \hat{\mathbf{x}}_t, \breve{\boldsymbol{g}}_t \rangle &\leq \sum_{t=1}^{T} ||\breve{\mathbf{x}}_t - \hat{\mathbf{x}}_t||_2 ||\breve{\boldsymbol{g}}_t||_2 \\
&\leq \sum_{t=1}^{T} \sqrt{d} v_t L \quad \text{(using eq.(2) from main text)} \\
&= \sqrt{d} L \sum_{t=1}^{T} \frac{v_0}{\sqrt{t}} \\
&= \sqrt{d} L v_0 \sqrt{T} = \mathcal{O}(\sqrt{T})
\end{aligned}
$$

Together, we have $R[X] \leq \mathcal{O}(\sqrt{T})$ as desired.

$\square$

Lemma A.1 For $\mathbf{x}^*, \breve{\mathbf{x}}_t \in X$, and $X = \mathbb{R}^d$,

$$
\langle \breve{\boldsymbol{g}}_t, \breve{\mathbf{x}}_t - \mathbf{x}^* \rangle = \frac{1}{2} \breve{\eta}_t ||\breve{\boldsymbol{g}}_t||^2 + \frac{D(\mathbf{x}^*||\hat{\mathbf{x}}_t) - D(\mathbf{x}^*||\hat{\mathbf{x}}_{t+1})}{\breve{\eta}_t} + \langle \breve{\mathbf{x}}_t - \hat{\mathbf{x}}_t, \breve{\boldsymbol{g}}_t \rangle
$$

where $D(x||x') := \frac{1}{2}||x - x'||^2$.

Proof.

$$
\begin{aligned}
D(\mathbf{x}^*||\hat{\mathbf{x}}_t) - D(\mathbf{x}^*||\hat{\mathbf{x}}_{t+1}) &= \frac{1}{2}||\mathbf{x}^* - \hat{\mathbf{x}}_t + \hat{\mathbf{x}}_t - \hat{\mathbf{x}}_{t+1}||^2 - \frac{1}{2}||\mathbf{x}^* - \hat{\mathbf{x}}_t||^2 \\
&= \frac{1}{2}||\mathbf{x}^* - \hat{\mathbf{x}}_t + \breve{\eta}_t \breve{\boldsymbol{g}}_t||^2 - \frac{1}{2}||\mathbf{x}^* - \hat{\mathbf{x}}_t||^2 \\
&= \frac{1}{2}\breve{\eta}_t ||\breve{\boldsymbol{g}}_t||^2 - \breve{\eta}_t \langle \hat{\mathbf{x}}_t - \mathbf{x}^*, \breve{\boldsymbol{g}}_t \rangle
\end{aligned}
$$

Divide both sides by $\breve{\eta}_t$ gets the desired answer.

$\square$

Lemma 3.1. $\bar{u}_t \leq \frac{\eta}{\sqrt{t}} L$ *and* $\gamma_t := ||\boldsymbol{\gamma}_t||_2 \leq P(2s+1)$.

Proof. $||\mathbf{u}_t||_2 = ||-\eta_t \nabla f_t||_2 \leq \frac{\eta}{\sqrt{t}} L$ since f is L-Lipschitz. Therefore $\bar{u}_t = \frac{1}{P(2s+1)} \sum_{t' \in \mathcal{W}_t} ||\mathbf{u}_{t'}||_2 \leq \frac{\eta}{\sqrt{t}} L$ since $|\mathcal{W}_t| \leq P(2s+1)$.

If $\bar{u}_t = 0$, then $\boldsymbol{\gamma}_t = \mathbf{0}$ and the lemma holds trivially. For $\bar{u}_t > 0$. $\boldsymbol{\gamma}_t = \frac{1}{\bar{u}_t}(\tilde{\mathbf{x}}_t - \mathbf{x}_t) = \frac{1}{\bar{u}_t} \sum_{t' \in \mathcal{S}_t} \mathbf{u}_{t'}$. Thus $||\boldsymbol{\gamma}_t||_2 = \frac{1}{\bar{u}_t} || \sum_{t' \in \mathcal{S}_t} \mathbf{u}_{t'} ||_2 \leq \frac{1}{\bar{u}_t} \sum_{t' \in \mathcal{S}_t} ||\mathbf{u}_{t'}||_2 \leq \frac{1}{\bar{u}_t} \sum_{t' \in \mathcal{W}_t} ||\mathbf{u}_{t'}||_2 = P(2s+1)$. $\square$

B.2 Proof of Theorem 3.4

Theorem 3.4 (SGD under SSP, convergence in probability)**.** *Given convex function* $f(\boldsymbol{x}) = \sum_{t=1}^{T} f_t(\boldsymbol{x})$ *such that components* f_t *are also convex. We search for minimizer* $\boldsymbol{x}^*$ *via gradient descent on each component* ∇f_t *under SSP with staleness parameter* s *and* P *workers. Let* $\boldsymbol{u}_t := -\eta_t \nabla_t f_t(\tilde{\boldsymbol{x}}_t)$ *with* $\eta_t = \frac{\eta}{\sqrt{t}}$*. Under suitable conditions (*f_t *are* L*-Lipschitz and bounded divergence* $D(x||x') \leq F^2$*), we have*

$$P\left[\frac{R\left[X\right]}{T} - \frac{1}{\sqrt{T}}\left(\eta L^2 + \frac{F^2}{\eta} + 2\eta L^2 \mu_\gamma\right) \geq \tau\right] \leq \exp\left\{\frac{-T\tau^2}{2\bar{\eta}_T \sigma_\gamma + \frac{2}{3}\eta L^2 (2s+1) P\tau}\right\}$$

where $R[X] := \sum_{t=1}^{T} f_t(\tilde{x}_t) - f(x^*)$*, and* $\bar{\eta}_T = \frac{\eta^2 L^4 (\ln T + 1)}{T} = o(T)$*.*

Proof. From lemma A.1, substitute $\breve{\mathbf{x}}_t$ with $\tilde{x}_t$ we have

$$\begin{aligned}
R\left[X\right] &\leq \sum_{t=1}^{T} \langle \tilde{g}_t, \tilde{x}_t - x^* \rangle \\
&= \sum_{t=1}^{T} \frac{1}{2}\eta_t \left\|\tilde{g}_t\right\|^2 + \frac{D\left(x^* \| x_t\right) - D\left(x^* \| x_{t+1}\right)}{\eta_t} + \langle \tilde{x}_t - x_t, \tilde{g}_t \rangle \\
&\leq \eta L^2 \sqrt{T} + \frac{F^2}{\eta}\sqrt{T} + \sum_{t=1}^{T} \langle \bar{u}_t \boldsymbol{\gamma}_t, \tilde{g}_t \rangle \\
&\leq \eta L^2 \sqrt{T} + \frac{F^2}{\eta}\sqrt{T} + \sum_{t=1}^{T} \frac{\eta}{\sqrt{t}} L^2 \gamma_t
\end{aligned}$$

Where the last step uses the fact

$$\begin{aligned}
\langle \bar{u}_t \boldsymbol{\gamma}_t, \tilde{g}_t \rangle &\leq \bar{u}_t ||\boldsymbol{\gamma}_t||_2 ||\tilde{g}_t||_2 \\
&\leq \gamma_t \frac{\eta}{\sqrt{t}} L^2 \qquad \text{(Lemma 4)}
\end{aligned}$$

Dividing T on both sides,

$$\frac{R\left[X\right]}{T} - \frac{\eta L^2}{\sqrt{T}} - \frac{F^2}{\eta\sqrt{T}} \leq \frac{\sum_{t=1}^{T} \frac{\eta}{\sqrt{t}} L^2 \gamma_t}{T} \tag{B.1}$$

Let $a_t := \frac{\eta}{\sqrt{t}} L^2 (\gamma_t - \mu_\gamma)$. Notice that a_t zero-mean, and $|a_t| \leq \eta L^2 \max_t(\gamma_t) \leq \eta L^2 (2s+1)P$. Also, $\frac{1}{T}\sum_{t=1}^{T} var(a_t) = \frac{1}{T}\sum_{t=1}^{T} \frac{\eta^2}{t} L^4 \sigma_\gamma < \frac{\eta^2 L^4 \sigma_\gamma}{T}(\ln T + 1) = \bar{\eta}_T \sigma_\gamma$ where $\bar{\eta}_T = \frac{\eta^2 L^4 (\ln T + 1)}{T}$.

Bernstein's inequality gives, for $\tau > 0$,

$$P\left(\frac{\sum_{t=1}^{T}\frac{\eta}{\sqrt{t}}L^2\gamma_t - \frac{\eta}{\sqrt{t}}L^2\mu_\gamma}{T} \geq \tau\right) \leq \exp\left\{\frac{-T\tau^2}{2\bar{\eta}_T\sigma_\gamma + \frac{2}{3}\eta L^2(2s+1)P\tau}\right\} \tag{B.2}$$

Note the following identity:

$$\sum_{i=a}^{b}\frac{1}{\sqrt{i}} \leq 2\sqrt{b-a+1} \tag{B.3}$$

Thus

$$\frac{1}{T}\sum_{t=1}^{T}\frac{\eta}{\sqrt{t}}L^2\mu_\gamma \leq \frac{2\eta L^2\mu_\gamma}{\sqrt{T}} \tag{B.4}$$

Plugging eq. B.1 and B.4 to eq. B.2, we have

$$P\left[\frac{R\left[X\right]}{T} - \frac{1}{\sqrt{T}}\left(\eta L^2 + \frac{F^2}{\eta} + 2\eta L^2\mu_\gamma\right) \geq \tau\right] \leq \exp\left\{\frac{-T\tau^2}{2\bar{\eta}_T\sigma_\gamma + \frac{2}{3}\eta L^2(2s+1)P\tau}\right\}$$

□

We need the following Lemma to prove Theorem 2 and 6.

Lemma A.2 Let $\mathbf{\Omega}^*$ be the hessian of the loss at optimum $\mathbf{x}^*$, then

$$\boldsymbol{g}_t := \nabla f(\tilde{\mathbf{x}}_t) = (\tilde{\mathbf{x}}_t - \mathbf{x}^*)\mathbf{\Omega}^* + \mathcal{O}(\rho_t^2)$$

for $\tilde{\mathbf{x}}_t$ close to the optimum such that $\mathcal{O}(\rho_t) = \mathcal{O}(||\tilde{\mathbf{x}}_t - \mathbf{x}^*||)$ is small. Here $\mathbf{\Omega}^* = \nabla^2 f(\mathbf{x})|_{\mathbf{x}=\mathbf{x}^*}$ is the Hessian at the optimum

Proof. Using Taylor's theorem and expanding around $\mathbf{x}^*$,

$$\begin{aligned} f(\tilde{\mathbf{x}}_t) &= f(\mathbf{x}^*) + (\tilde{\mathbf{x}}_t - \mathbf{x}^*)^T \left.\nabla f(\mathbf{x})\right|_{\mathbf{x}=\mathbf{x}^*} \\ &\quad + \frac{1}{2}(\tilde{\mathbf{x}}_t - \mathbf{x}^*)^T\mathbf{\Omega}^*(\tilde{\mathbf{x}}_t - \mathbf{x}^*) + \mathcal{O}(||\tilde{\mathbf{x}}_t - \mathbf{x}^*||^3) \\ &= f(\mathbf{x}^*) + \frac{1}{2}(\tilde{\mathbf{x}}_t - \mathbf{x}^*)^T\mathbf{\Omega}^*(\tilde{\mathbf{x}}_t - \mathbf{x}^*) + \mathcal{O}(||\tilde{\mathbf{x}}_t - \mathbf{x}^*||^3) \end{aligned}$$

where the last step uses $\nabla f(\mathbf{x}) = 0$ at $\mathbf{x}^*$. Taking gradient w.r.t. $\tilde{\mathbf{x}}_t$,

$$\begin{aligned} \nabla f(\tilde{\mathbf{x}}_t) &= (\tilde{\mathbf{x}}_t - \mathbf{x}^*)^T\mathbf{\Omega}^* + \mathcal{O}(||\tilde{\mathbf{x}}_t - \mathbf{x}^*||^2) \\ &= (\tilde{\mathbf{x}}_t - \mathbf{x}^*)^T\mathbf{\Omega}^* + \mathcal{O}(\rho_t^2) \end{aligned}$$

□

B.3 Proof of Theorem 3.5

Theorem 3.5 (SGD under SSP, decreasing variance)**.** *Given the setup in Theorem 3.4 and assumption 1-2. Further assume that* $f(\boldsymbol{x})$ *has bounded and invertible Hessian* Ω^* *at optimum* $\boldsymbol{x}^*$ *and* γ_t *is bounded. Let* $\mathrm{Var}_t := \mathbb{E}[\tilde{\boldsymbol{x}}_t^2] - \mathbb{E}[\tilde{\boldsymbol{x}}_t]^2$, $\boldsymbol{g}_t = \nabla f_t(\tilde{\boldsymbol{x}}_t)$, η_t *be the learning rate, then for* $\tilde{\boldsymbol{x}}_t$ *near the optima* $\boldsymbol{x}^*$ *such that* $\rho_t = ||\tilde{\boldsymbol{x}}_t - \boldsymbol{x}^*||$ *and* $\xi_t = ||\boldsymbol{g}_t|| - ||\boldsymbol{g}_{t+1}||$ *are small:*

$$\mathrm{Var}_{t+1} = \mathrm{Var}_t - 2\eta_t cov(\boldsymbol{x}_t, \mathbb{E}^{\Delta_t}[\boldsymbol{g}_t]) + \mathcal{O}(\eta_t\xi_t) \tag{3.6}$$

$$+ \mathcal{O}(\eta_t^2\rho_t^2) + \mathcal{O}^*_{\gamma_t} \tag{3.7}$$

where covariance $cov(\boldsymbol{a}, \boldsymbol{b}) := \mathbb{E}[\boldsymbol{a}^T\boldsymbol{b}] - \mathbb{E}[\boldsymbol{a}^T]\mathbb{E}[\boldsymbol{b}]$ *uses inner product.* $\mathcal{O}^*_{\gamma_t}$ *are high order (*$\geq$ *5th) terms involving* $\gamma_t = ||\gamma_t||_\infty$. Δ_t *is a random variable capturing the randomness of update* $\boldsymbol{u}_t$ *conditioned on* $\boldsymbol{x}_t$.

Proof. We write eq. 3 from the main text as $\tilde{\mathbf{x}}_t = \mathbf{x}_t + \boldsymbol{\delta}_t$ with $\boldsymbol{\delta}_t = \bar{u}_t\boldsymbol{\gamma}_t$. Conditioned on $\mathbf{x}_t$, we have

$$p(\tilde{\mathbf{x}}_t|\mathbf{x}_t)d\tilde{\mathbf{x}}_t = p(V_t(\boldsymbol{\delta}_t, \mathbf{x}_t))dV_t \tag{B.5}$$

where V_t is a random variable representing the state of $\boldsymbol{\delta}_t$ conditioned on $\mathbf{x}_t$. We can express $\mathbb{E}^{\tilde{\mathbf{x}}_t}[f(\tilde{\mathbf{x}}_t)]$ in terms of $\mathbb{E}^{\mathbf{x}_t}$ for any function $f()$ of $\tilde{\mathbf{x}}_t$:

$$\begin{aligned}
\mathbb{E}^{\tilde{\mathbf{x}}_t}[f(\tilde{\mathbf{x}}_t)] &= \int_{\tilde{\mathbf{x}}_t} f(\tilde{\mathbf{x}}_t)p(\tilde{\mathbf{x}}_t)d\tilde{\mathbf{x}}_t \\
&= \int_{\tilde{\mathbf{x}}_t}\int_{\mathbf{x}_t} f(\tilde{\mathbf{x}}_t)p(\tilde{\mathbf{x}}_t|\mathbf{x}_t)p(\mathbf{x}_t)d\mathbf{x}_t d\tilde{\mathbf{x}}_t \qquad \text{(using eq. B.5)} \\
&= \int_{\mathbf{x}_t}\int_{V_t} f(\tilde{\mathbf{x}}_t)p(V_t(\boldsymbol{\delta}_t, \mathbf{x}_t))dV_t d\mathbf{x}_t \\
&= \mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{V_t}[f(\tilde{\mathbf{x}}_t)]\right]
\end{aligned} \tag{B.6}$$

Similarly, we have

$$\mathbb{E}^{\tilde{\mathbf{x}}_{t+1}}[f(\tilde{\mathbf{x}}_{t+1})] = \mathbb{E}^{\mathbf{x}_{t+1}}\left[\mathbb{E}^{V_{t+1}}[f(\tilde{\mathbf{x}}_{t+1})]\right] \tag{B.7}$$

In the same vein, we introduce random variable Δ, conditioned on $\mathbf{x}_t$:

$$p(\mathbf{x}_{t+1}|\mathbf{x}_t)d\mathbf{x}_{t+1} = p(\Delta_t(\mathbf{u}_t, \mathbf{x}_t))d\Delta_t \tag{B.8}$$

since $\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{u}_t$ (eq. 2 in the main text). Here Δ is a random variable representing the state of $\mathbf{u}_t$ conditioned on $\mathbf{x}_t$. Analogous to eq. B.6, we have

$$\mathbb{E}^{\mathbf{x}_{t+1}}[f(\mathbf{x}_{t+1})] = \mathbb{E}^{\mathbf{x}_t}[\mathbb{E}^{\Delta_t}[f(\mathbf{x}_{t+1})]] \tag{B.9}$$

for some function $f()$ of $\mathbf{x}_{t+1}$. There are a few facts we will use throughout:

$$\mathbb{E}^{\mathbf{x}_t}\left[h(\mathbf{x}_t, \bar{u}_t)\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t]\right] = \mathbb{E}^{\mathbf{x}_t}[h(\mathbf{x}_t, \bar{u}_t)]\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t] \qquad \text{(since } \boldsymbol{\gamma}_t \perp \mathbf{x}_t, \bar{u}_t\text{)} \tag{B.10}$$

$$\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{x}_t^T g(\mathbf{u}_t)]\right] = \mathbb{E}^{\mathbf{x}_t}\left[\mathbf{x}_t^T\mathbb{E}^{\Delta_t}[g(\mathbf{u}_t)]\right] \qquad (\Delta_t \text{ conditioned on } \mathbf{x}_t) \tag{B.11}$$

$$\mathbb{E}^{\Delta_t}[\bar{u}_{t+1}] = \bar{u}_{t+1} \tag{B.12}$$

where $h(\mathbf{x}_t, \bar{u}_t)$ is some function of $\mathbf{x}_t$ and $\bar{u}_t$, and similarly for $g()$. Eq. B.12 follows from $\bar{u}_{t+1}$ being an average over the randomness represented by Δ_t. We can now expand Var_t:

$$\begin{aligned}
\mathrm{Var}_t &= \mathbb{E}^{\tilde{\mathbf{x}}_t}[\tilde{\mathbf{x}}_t^2] - (\mathbb{E}^{\tilde{\mathbf{x}}_t}[\tilde{\mathbf{x}}_t])^2 \\
&= \mathbb{E}^{\mathbf{x}_t}[\mathbb{E}^{V_t}[\tilde{\mathbf{x}}_t^2]] - (\mathbb{E}^{\mathbf{x}_t}[\mathbb{E}^{V_t}[\tilde{\mathbf{x}}_t]])^2 && \text{(using eq. B.6)} \\
&= \mathbb{E}^{\mathbf{x}_t}[\mathbb{E}^{V_t}[\mathbf{x}_t^2 + \boldsymbol{\delta}_t^2 + 2\mathbf{x}_t^T\boldsymbol{\delta}_t]] - (\mathbb{E}^{\mathbf{x}_t}[\mathbb{E}^{V_t}[\mathbf{x}_t + \boldsymbol{\delta}_t]])^2 && \text{(B.13)}
\end{aligned}$$

We expand each term:

$$\begin{aligned}
&\mathbb{E}^{\mathbf{x}_t}[\mathbb{E}^{V_t}[\mathbf{x}_t^2 + \boldsymbol{\delta}_t^2 + 2\mathbf{x}_t^T\boldsymbol{\delta}_t]] \\
&= \mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^2 + \mathbb{E}^{V_t}[\boldsymbol{\delta}_t^2] + 2\mathbf{x}_t^T\mathbb{E}^{V_t}[\boldsymbol{\delta}_t]] \\
&= \mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^2] + \mathbb{E}^{\mathbf{x}_t}[\bar{u}_t^2\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t^2]] + 2\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^T\bar{u}_t\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t]] \\
&= \mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^2] + \mathbb{E}^{\mathbf{x}_t}[\bar{u}_t^2]\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t^2] + 2\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^T\bar{u}_t]\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t]
\end{aligned}$$

$$\begin{aligned}
&(\mathbb{E}^{\mathbf{x}_t}[\mathbb{E}^{V_t}[\mathbf{x}_t + \boldsymbol{\delta}_t]])^2 \\
&= (\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t + \mathbb{E}^{V_t}[\boldsymbol{\delta}_t]])^2 \\
&= (\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t + \bar{u}_t\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t]])^2 \\
&= (\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t] + \mathbb{E}^{\mathbf{x}_t}[\bar{u}_t]\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t]])^2 \\
&= \mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t]^2 + \mathbb{E}^{\mathbf{x}_t}[\bar{u}_t]^2\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t]^2 + 2\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^T]\mathbb{E}^{\mathbf{x}_t}[\bar{u}_t]\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t]
\end{aligned}$$

Therefore

$$\begin{aligned}
\mathrm{Var}_t = {} & \mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^2] + \mathbb{E}^{\mathbf{x}_t}[\bar{u}_t^2]\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t^2] + 2\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^T\bar{u}_t]\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t] \\
& - \mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t]^2 - \mathbb{E}^{\mathbf{x}_t}[\bar{u}_t]^2\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t]^2 - 2\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^T]\mathbb{E}^{\mathbf{x}_t}[\bar{u}_t]\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t]
\end{aligned} \tag{B.14}$$

Following similar procedures, we can write Var_{t+1} as

$$\begin{aligned}
\mathrm{Var}_{t+1} = {} & \mathbb{E}^{\mathbf{x}_{t+1}}[\mathbf{x}_{t+1}^2] + \mathbb{E}^{\mathbf{x}_{t+1}}[\bar{u}_{t+1}^2]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}^2] \\
& + 2\mathbb{E}^{\mathbf{x}_{t+1}}[\mathbf{x}_{t+1}^T\bar{u}_{t+1}]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}] \\
& - \mathbb{E}^{\mathbf{x}_{t+1}}[\mathbf{x}_{t+1}]^2 - \mathbb{E}^{\mathbf{x}_{t+1}}[\bar{u}_{t+1}]^2\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}]^2 \\
& - 2\mathbb{E}^{\mathbf{x}_{t+1}}[\mathbf{x}_{t+1}^T]\mathbb{E}^{\mathbf{x}_{t+1}}[\bar{u}_{t+1}]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}]
\end{aligned} \tag{B.15}$$

We tackle each term separately:

$$\begin{aligned}
\mathbb{E}^{\mathbf{x}_{t+1}}[\mathbf{x}_{t+1}^2] &= \mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[(\mathbf{x}_t + \mathbf{u}_t)^2]\right] && \text{(using eq. B.9, 2 main text)} \\
&= \mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^2] + \mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{u}_t^2]\right] + 2\mathbb{E}^{\mathbf{x}_t}\left[\mathbf{x}_t^T\mathbb{E}^{\Delta_t}[\mathbf{u}_t]\right] && \text{(using eq. B.11)}
\end{aligned}$$

$$
\begin{aligned}
&2\mathbb{E}^{\mathbf{x}_{t+1}}[\mathbf{x}_{t+1}^T\bar{u}_{t+1}]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}] \\
&= 2\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[(\mathbf{x}_t+\mathbf{u}_t)^T\bar{u}_{t+1}]\right]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}] && \text{(using eq. B.9, 2 main text)} \\
&= 2\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{x}_t^T\bar{u}_{t+1}]\right]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}] \\
&+ 2\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{u}_t^T\bar{u}_{t+1}]\right]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}] \\
&= 2\mathbb{E}^{\mathbf{x}_t}\left[\mathbf{x}_t^T\bar{u}_{t+1}\right]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}] && \text{(using eq. B.11 and B.12)} \\
&+ 2\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{u}_t^T\bar{u}_{t+1}]\right]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}]
\end{aligned}
$$

$$
\begin{aligned}
-\mathbb{E}^{\mathbf{x}_{t+1}}[\mathbf{x}_{t+1}]^2 &= -\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{x}_t+\mathbf{u}_t]\right]^2 \\
&= -\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t]^2 - \mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{u}_t]\right]^2 - 2\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^T]\mathbb{E}^{x_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{u}_t]\right]
\end{aligned}
$$

$$
\begin{aligned}
&- 2\mathbb{E}^{\mathbf{x}_{t+1}}[\mathbf{x}_{t+1}^T]\mathbb{E}^{\mathbf{x}_{t+1}}[\bar{u}_{t+1}]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}] \\
&= -2\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[(\mathbf{x}_t+\mathbf{u}_t)^T]\right]\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\bar{u}_{t+1}]\right]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}] \\
&= -2\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{u}_t^T]\right]\mathbb{E}^{\mathbf{x}_t}[\bar{u}_{t+1}]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}] - 2\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^T]\mathbb{E}^{\mathbf{x}_t}[\bar{u}_{t+1}]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}]
\end{aligned}
$$

Assuming stationarity for $\boldsymbol{\gamma}_t$, and thus $\bar{\boldsymbol{\gamma}} := \mathbb{E}^{V_t}[\boldsymbol{\gamma}_t] = \mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}]$, we have

$$
\begin{aligned}
\mathrm{Var}_{\,t+1} - \mathrm{Var}_{\,t} &= 2\left\{\mathbb{E}^{\mathbf{x}_t}\left[\mathbf{x}_t^T\mathbb{E}^{\Delta_t}[\mathbf{u}_t]\right] - \mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^T]\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{u}_t]\right]\right\} \\
&- 2\left\{\mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^T(\bar{u}_t-\bar{u}_{t+1})\bar{\boldsymbol{\gamma}}] - \mathbb{E}^{\mathbf{x}_t}[\mathbf{x}_t^T]\mathbb{E}^{\mathbf{x}_t}[(\bar{u}_t-\bar{u}_{t+1})\bar{\boldsymbol{\gamma}}]\right\} \\
&+ \Big\{\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{u}_t^2]\right] + \mathbb{E}^{\mathbf{x}_{t+1}}[\bar{u}_{t+1}^2]\mathbb{E}^{V_{t+1}}[\boldsymbol{\gamma}_{t+1}^2] - \mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{u}_t]\right]^2 \\
&- \mathbb{E}^{\mathbf{x}_t}[\bar{u}_{t+1}]^2\bar{\boldsymbol{\gamma}}^2 - \mathbb{E}^{\mathbf{x}_t}[\bar{u}_t^2]\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t^2] + \mathbb{E}^{\mathbf{x}_t}[\bar{u}_t]^2\mathbb{E}^{V_t}[\boldsymbol{\gamma}_t^2] \\
&+2\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{u}_t^T\bar{u}_{t+1}]\right]\bar{\boldsymbol{\gamma}} - 2\mathbb{E}^{\mathbf{x}_t}\left[\mathbb{E}^{\Delta_t}[\mathbf{u}_t^T]\right]\mathbb{E}^{\mathbf{x}_t}[\bar{u}_{t+1}]\bar{\boldsymbol{\gamma}}\Big\} \\
&= 2cov(\mathbf{x}_t, \mathbb{E}^{\Delta_t}[\mathbf{u}_t]) + \mathcal{O}(\eta_t\xi_t) + \mathcal{O}(\eta_t^2\rho_t^2) + \mathcal{O}^*
\end{aligned}
$$

where $\xi_t = ||\boldsymbol{g}_t|| - ||\boldsymbol{g}_{t+1}||$ and $\mathcal{O}^*$ are higher order terms. In the last step we use the fact that $||\boldsymbol{g}_t|| = \mathcal{O}(\rho_t)$ (lemma A.2) and thus $||\mathbf{u}_t|| = \eta_t||\nabla f(\mathbf{x}_t)||$ and $\bar{u}_t$ are both $\mathcal{O}(\eta_t\rho_t)$. Notice that $cov(\boldsymbol{v}_1, \boldsymbol{v}_2) := \mathbb{E}[\boldsymbol{v}_1^T\boldsymbol{v}_2] - \mathbb{E}[\boldsymbol{v}_1^T]\mathbb{E}[\boldsymbol{v}_2]$ uses inner product. Thus,

$$
\mathrm{Var}_{\,t+1} = \mathrm{Var}_{\,t} - 2\eta_t cov(\mathbf{x}_t, \mathbb{E}^{\Delta_t}[\boldsymbol{g}_t]) + \mathcal{O}(\eta_t\xi_t) + \mathcal{O}(\eta_t^2\rho_t^2) + \mathcal{O}^* \tag{B.16}
$$

□

B.4 Proof of Theorem 3.2

Theorem 3.2 (SGD under VAP, bounded variance)**.** *Assuming* $f(\boldsymbol{x})$, $\breve{\eta}_t$, *and* v_t *similar to theorem 3.1 above, and further assume that* $f(\boldsymbol{x})$ *has bounded and invertible Hessian,* Ω^* *defined at optimal point* $\boldsymbol{x}^*$*. Let* $\mathrm{Var}_t := \mathbb{E}[\breve{\boldsymbol{x}}_t^2] - \mathbb{E}[\breve{\boldsymbol{x}}_t]^2$, *and* $\breve{\boldsymbol{g}}_t = \nabla f_t(\breve{\boldsymbol{x}}_t)$ *be the gradient, then:*

$$\mathrm{Var}_{t+1} = \mathrm{Var}_t - 2cov(\hat{\boldsymbol{x}}_t, \mathbb{E}^{\Delta_t}[\breve{\boldsymbol{g}}_t]) + \mathcal{O}(\delta_t) \tag{3.2}$$

$$+ \mathcal{O}(\breve{\eta}_t^2 \rho_t^2) + \mathcal{O}^*_{\delta_t} \tag{3.3}$$

near the optima $\boldsymbol{x}^*$*. The covariance* $cov(\boldsymbol{a}, \boldsymbol{b}) := \mathbb{E}[\boldsymbol{a}^T \boldsymbol{b}] - \mathbb{E}[\boldsymbol{a}^T]\mathbb{E}[\boldsymbol{b}]$ *uses inner product.* $\delta_t = ||\boldsymbol{\delta}_t||_\infty$ *and* $\boldsymbol{\delta}_t = \breve{\boldsymbol{x}}_t - \hat{\boldsymbol{x}}_t$*.* $\rho_t = ||\breve{\boldsymbol{x}}_t - \boldsymbol{x}^*||$*.* Δ_t *is a random variable capturing the randomness of update* $\hat{\boldsymbol{u}}_t = -\eta_t \breve{\boldsymbol{g}}_t$ *conditioned on* $\hat{\boldsymbol{x}}_t$ *(see the appendix).*

Proof. The proof is similar to the proof of Theorem 6. Starting off with $\breve{\mathbf{x}}_t = \hat{\mathbf{x}}_t + \boldsymbol{\delta}_t$, we define V_t, Δ_t analogously. We have

$$\begin{aligned} \mathrm{Var}_t &= \mathbb{E}^{\hat{\mathbf{x}}_t}[\hat{\mathbf{x}}_t^2] + \mathbb{E}^{\hat{\mathbf{x}}_t}[\mathbb{E}^{V_t}[\boldsymbol{\delta}_t^2]] + 2\mathbb{E}^{\hat{\mathbf{x}}_t}[\hat{\mathbf{x}}_t^T \mathbb{E}^{V_t}[\boldsymbol{\delta}_t]] \\ &\quad - \mathbb{E}^{\hat{\mathbf{x}}_t}[\hat{\mathbf{x}}_t]^2 - \mathbb{E}^{\hat{\mathbf{x}}_t}[\mathbb{E}^{V_t}[\boldsymbol{\delta}_t^2]] - 2\mathbb{E}^{\hat{\mathbf{x}}_t}[\hat{\mathbf{x}}_t]\mathbb{E}^{\hat{\mathbf{x}}_t^T}[\mathbb{E}^{V_t}[\boldsymbol{\delta}_t]] \end{aligned}$$

Similar algebra as in Theorem 6 leads to

$$\begin{aligned} \mathrm{Var}_{t+1} - \mathrm{Var}_t &= 2cov(\hat{\mathbf{x}}_t, \mathbb{E}^{\Delta_t}[\hat{\mathbf{u}}_t]) + 2cov(\hat{\mathbf{x}}_t, \mathbb{E}^{V_t}[\boldsymbol{\delta}_t] - \mathbb{E}^{\Delta_t}[\mathbb{E}^{V_{t+1}}[\boldsymbol{\delta}_{t+1}]]) \\ &\quad + \mathcal{O}(\delta_t^2) + \mathcal{O}(\breve{\eta}_t^2 \rho_t^2) + \mathcal{O}(\breve{\eta}_t \delta_t) + \mathcal{O}^* \\ &= -2cov(\hat{\mathbf{x}}_t, \mathbb{E}^{\Delta_t}[\breve{\boldsymbol{g}}_t]) + \mathcal{O}(\delta_t) + \mathcal{O}(\breve{\eta}_t^2 \rho_t^2) + \mathcal{O}^*_{\delta_t} \end{aligned}$$

where $\delta_t = ||\boldsymbol{\delta}_t||_\infty$. This is the desired result in the theorem statement. □

Appendix C

Appendix for Chapter 4

C.1 Proof of Theorem 4.1

Theorem 4.1 (Asymptotic consistency)**.** *Let Assumption 4.1 and 4.2 hold, and apply* msPG *to problem* (P). *If the step size* $\eta < (L_f + 2Ls)^{-1}$*, then the global model and local models satisfy:*

1. $\sum_{t=0}^{\infty} \|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2 < \infty$;
2. $\lim_{t\to\infty} \|\mathbf{x}(t+1) - \mathbf{x}(t)\| = 0$, $\lim_{t\to\infty} \|\mathbf{x}(t) - \mathbf{x}^i(t)\| = 0$;
3. *The limit points* $\omega(\{\mathbf{x}(t)\}) = \omega(\{\mathbf{x}^i(t)\}) \subseteq \operatorname{crit} F$.

Proof. Recall that $(t)_+ = \max\{t, 0\}$ is the positive part of t. We start from bounding the difference between the global model $\mathbf{x}$ and the local model $\mathbf{x}^i$ (on any machine i). Indeed, at iteration

t, by the definition of the global and local models in msPG:

$$
\begin{aligned}
\|\mathbf{x}(t)-\mathbf{x}^i(t)\| &= \sqrt{\sum_{j=1}^{p}\|x_j(t)-x_j(\tau_j^i(t))\|^2} \\
\text{(triangle inequality)} &\le \sqrt{\sum_{j=1}^{p}\left(\sum_{k=\tau_j^i(t)}^{t-1}\|x_j(k+1)-x_j(k)\|\right)^2} \\
\text{(Assumption 4.2.1)} &\le \sqrt{\sum_{j=1}^{p}\left(\sum_{k=(t-s)_+}^{t-1}\|x_j(k+1)-x_j(k)\|\right)^2} \qquad \text{(C.1)} \\
&= \left\|\Big(\sum_{k=(t-s)_+}^{t-1}\|x_1(k+1)-x_1(k)\|,\ \cdots,\ \sum_{k=(t-s)_+}^{t-1}\|x_p(k+1)-x_p(k)\|\Big)\right\| \\
&= \left\|\sum_{k=(t-s)_+}^{t-1}\big(\|x_1(k+1)-x_1(k)\|,\ \cdots,\ \|x_p(k+1)-x_p(k)\|\big)\right\| \\
\text{(triangle inequality)} &\le \sum_{k=(t-s)_+}^{t-1}\left\|\big(\|x_1(k+1)-x_1(k)\|,\ \cdots,\ \|x_p(k+1)-x_p(k)\|\big)\right\| \\
&= \sum_{k=(t-s)_+}^{t-1}\|\mathbf{x}(k+1)-\mathbf{x}(k)\|,
\end{aligned}
$$

where in the last equality we used the following property of the Euclidean norm:

$$\|\mathbf{x}\| = \|(x_1,\ \ldots,\ x_p)\| = \|(\|x_1\|,\ \ldots,\ \|x_p\|)\|. \tag{C.2}$$

Equation (C.2) bounds the inconsistency between the global model and the local models. We will repeatedly use it in the following, as it provides a bridge to jump from the global model and the local models back and forth.

Next we bound the progress of the global model $\mathbf{x}(t)$. If $t \in T_i$ (i.e., machine i updates at iteration t), then using the definition of the update operator $U_i(\mathbf{x}^i(t))$ in Equation (4.10) we can rewrite $x_i(t+1)$ as

$$x_i(t+1) = \operatorname{prox}_{g_i}^{\eta}\big(x_i(t)-\eta\nabla_i f(\mathbf{x}^i(t))\big), \tag{C.3}$$

where we recall the proximal map $\operatorname{prox}_{g_i}^{\eta}$ from Definition 4.3. Thus, for all $z \in \mathbb{R}^{d_i}$:

$$g_i\big(x_i(t+1)\big)+\frac{1}{2\eta}\left\|x_i(t+1)-x_i(t)+\eta\nabla_i f\big(\mathbf{x}^i(t)\big)\right\|^2 \le g_i\big(z\big)+\frac{1}{2\eta}\left\|z-x_i(t)+\eta\nabla_i f\big(\mathbf{x}^i(t)\big)\right\|^2. \tag{C.4}$$

Substituting with $z = x_i(t)$ and simplifying yields

$$g_i\big(x_i(t+1)\big) - g_i\big(x_i(t)\big) \le -\frac{1}{2\eta}\|x_i(t+1) - x_i(t)\|^2 - \big\langle \nabla_i f\big(\mathbf{x}^i(t)\big), x_i(t+1) - x_i(t)\big\rangle . \tag{C.5}$$

(If g_i is convex, we can replace $\frac{1}{2\eta}$ with $\frac{1}{\eta}$.) Note that if $t \notin T_i$, then $x_i(t+1) = x_i(t)$ hence Equation (C.5) still trivially holds. On the other hand, Assumption 4.1.2 implies

$$f\big(\mathbf{x}(t+1)\big) - f\big(\mathbf{x}(t)\big) \le \langle \mathbf{x}(t+1) - \mathbf{x}(t), \nabla f\big(\mathbf{x}(t)\big)\rangle + \frac{L_f}{2}\|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2. \tag{C.6}$$

Adding Equation (C.6) and Equation (C.5) (for all i) and recalling $F(\mathbf{x}) = f(\mathbf{x}) + \sum_i g_i(x_i)$, we have

$$\begin{aligned}
F\big(\mathbf{x}(t+1)\big) - F\big(\mathbf{x}(t)\big) \le{}& \tfrac{1}{2}(L_f - 1/\eta)\|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2 \\
&+ \sum_{i=1}^{p} \big\langle x_i(t+1) - x_i(t), \nabla_i f(\mathbf{x}(t)) - \nabla_i f\big(\mathbf{x}^i(t)\big)\big\rangle \\
(\text{ Cauchy-Schwarz }) \le{}& \tfrac{1}{2}(L_f - 1/\eta)\|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2 \\
&+ \sum_{i=1}^{p} \|x_i(t+1) - x_i(t)\| \cdot \|\nabla_i f(\mathbf{x}(t)) - \nabla_i f\big(\mathbf{x}^i(t)\big)\| \\
(\text{ Assumption 4.1.2 }) \le{}& \tfrac{1}{2}(L_f - 1/\eta)\|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2 \\
&+ \sum_{i=1}^{p} \|x_i(t+1) - x_i(t)\| \cdot L_i \|\mathbf{x}(t) - \mathbf{x}^i(t)\| \\
(\text{ Equation (C.2) }) \le{}& \tfrac{1}{2}(L_f - 1/\eta)\|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2 \\
&+ \sum_{i=1}^{p} L_i \|x_i(t+1) - x_i(t)\| \cdot \sum_{k=(t-s)_+}^{t-1} \|\mathbf{x}(k+1) - \mathbf{x}(k)\| \\
(\text{ Assumption 4.1.2 }) \le{}& \tfrac{1}{2}(L_f - 1/\eta)\|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2 \\
&+ L\|\mathbf{x}(t+1) - \mathbf{x}(t)\| \cdot \sum_{k=(t-s)_+}^{t-1} \|\mathbf{x}(k+1) - \mathbf{x}(k)\| \qquad \text{(C.7)} \\
(\; ab \le \tfrac{a^2+b^2}{2} \;) \le{}& \tfrac{1}{2}(L_f - 1/\eta)\|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2 \\
&+ \frac{L}{2} \sum_{k=(t-s)_+}^{t-1} \Big[\|\mathbf{x}(k+1) - \mathbf{x}(k)\|^2 + \|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2\Big] \\
\le{}& \tfrac{1}{2}(L_f + Ls - 1/\eta)\|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2 \\
&+ \frac{L}{2} \sum_{k=(t-s)_+}^{t-1} \|\mathbf{x}(k+1) - \mathbf{x}(k)\|^2.
\end{aligned}$$

Summing the above inequality from m to $n-1$ we have

$$\begin{aligned}F\big(\mathbf{x}(n)\big)-F\big(\mathbf{x}(m)\big)\leq&\tfrac{1}{2}(L_f+Ls-1/\eta)\sum_{t=m}^{n-1}\|\mathbf{x}(t+1)-\mathbf{x}(t)\|^2\\&+\frac{L}{2}\sum_{t=m}^{n-1}\sum_{k=(t-s)_+}^{t-1}\|\mathbf{x}(k+1)-\mathbf{x}(k)\|^2\\\leq&\tfrac{1}{2}(L_f+2Ls-1/\eta)\sum_{t=m}^{n-1}\|\mathbf{x}(t+1)-\mathbf{x}(t)\|^2.\end{aligned}$$

Therefore, as long as $\eta<1/(L_f+2Ls)$, letting $m=0$ we deduce

$$\begin{aligned}\sum_{t=0}^{n-1}\|\mathbf{x}(t+1)-\mathbf{x}(t)\|^2&\leq\frac{2}{1/\eta-L_f-2Ls}[F\big(\mathbf{x}(0)\big)-F\big(\mathbf{x}(n)\big)]\\&\leq\frac{2}{1/\eta-L_f-2Ls}[F\big(\mathbf{x}(0)\big)-\inf_{\mathbf{z}}F(\mathbf{z})].\end{aligned}$$

By Assumption 4.1.1, F is bounded from below hence the right-hand side is finite and independent of n. Letting n goes to infinity completes the proof of Item 1.

Item 2 follows immediately from Item 1 and (C.2), whence it is clear that for all i the limit points satisfy $\omega(\{\mathbf{x}(k)\})=\omega(\{\mathbf{x}^i(k)\})$.

To prove Item 3, let $\mathbf{x}^*$ be a limit point of $\{\mathbf{x}(t)\}_t$, i.e., there exists a subsequence $\mathbf{x}(t_m)\to\mathbf{x}^*$. Since the objective function F is closed we know $\mathbf{x}^*\in\operatorname{dom}F$. To show $\mathbf{x}^*\in\operatorname{crit}F$ we need to exhibit a sequence say $\mathbf{x}(k_m+1)$ such that[1]

$$\mathbf{x}(k_m+1)\to\mathbf{x}^*,\ F(\mathbf{x}(k_m+1))\to F(\mathbf{x}^*),\ \mathbf{0}\leftarrow\mathbf{u}(k_m+1)\in\partial F(\mathbf{x}(k_m+1)).\qquad\text{(C.8)}$$

This is the most difficult part of the proof, and the argument differs substantially from previous work (e.g. [19]).

We first prove the subdifferential goes to zero. Observe from Assumption 4.2.3 that the iterations $\{t,t\in T_i\}$ when machine i updates is infinite. Let $\hat{t}\in T_i$ and by the optimality condition of $x_i(\hat{t}+1)$ in Equation (C.3):

$$-\tfrac{1}{\eta}\left[x_i(\hat{t}+1)-x_i(\hat{t})+\eta\nabla_i f\big(\mathbf{x}^i(\hat{t})\big)\right]\in\partial g_i(x_i(\hat{t}+1)),\qquad\text{(C.9)}$$

---

[1] Technically, from Definition 4.1 we should have the Frechét subdifferential $\hat{\partial}F$ in Equation (C.8), however, a usual diagonal argument allows us to use the more convenient (limiting) subdifferential.

i.e. there exists $u_i(\hat{t}+1) \in \partial g_i(x_i(\hat{t}+1))$ such that

$$\begin{aligned}
\|u_i(\hat{t}+1) + \nabla_i f(\mathbf{x}(\hat{t}+1))\| \leq & \|u_i(\hat{t}+1) + \nabla_i f(\mathbf{x}(\hat{t}))\| \\
& + \|\nabla_i f(\mathbf{x}(\hat{t}+1)) - \nabla_i f(\mathbf{x}(\hat{t}))\| \\
(\text{ Equation (C.9), Assumption 4.1.2 }) \leq & \left\| \tfrac{1}{\eta}(x_i(\hat{t}+1) - x_i(\hat{t}) + \nabla_i f\big(\mathbf{x}^i(\hat{t})\big) - \nabla_i f\big(\mathbf{x}(\hat{t})\big) \right\| \\
& + L_i \|\mathbf{x}(\hat{t}+1) - \mathbf{x}(\hat{t})\| \\
(\text{ triangle inequality, Assumption 4.1.2 }) \leq & \tfrac{1}{\eta}\|x_i(\hat{t}+1) - x_i(\hat{t})\| + L_i \|\mathbf{x}^i(\hat{t}) - \mathbf{x}(\hat{t})\| \\
& + L_i \|\mathbf{x}(\hat{t}+1) - \mathbf{x}(\hat{t})\| \\
(\text{ Equation (C.2) }) \leq & \tfrac{1}{\eta}\|x_i(\hat{t}+1) - x_i(\hat{t})\| + L_i \sum_{k=(\hat{t}-s)_+}^{\hat{t}} \|\mathbf{x}(k+1) - \mathbf{x}(k)\|.
\end{aligned} \tag{C.10}$$

We now use a chaining argument to remove the condition $\hat{t} \in T_i$ above. For each $t \notin T_i$ let $\hat{t}_i$ be the *largest* element in $\{k \leq t : k \in T_i\}$. Thanks to Assumption 4.2.3 $\hat{t}_i$ always exists and $t - \hat{t}_i \leq s$. Therefore, for any $t \notin T_i$, since $x_i(t+1) = x_i(\hat{t}_i+1)$ we can certainly choose $u_i(t+1) \in \partial g_i(x_i(t+1))$ to coincide with $u_i(\hat{t}_i+1) \in \partial g_i(x_i(\hat{t}_i+1))$. Then:

$$\begin{aligned}
\|u_i(t+1) + \nabla_i f(\mathbf{x}(t+1)) - u_i(\hat{t}_i+1) - \nabla_i f(\mathbf{x}(\hat{t}_i+1))\| & \\
= \|\nabla_i f(\mathbf{x}(t+1)) - \nabla_i f(\mathbf{x}(\hat{t}_i+1))\| & \\
(\text{ triangle inequality }) \leq \sum_{k=\hat{t}_i+1}^{t} \|\nabla_i f(\mathbf{x}(k+1)) - \nabla_i f(\mathbf{x}(k))\| & \\
(\text{ Assumption 4.2.3 }) \leq \sum_{k=(t-s+1)_+}^{t} \|\nabla_i f(\mathbf{x}(k+1)) - \nabla_i f(\mathbf{x}(k))\| & \\
(\text{ Assumption 4.1.2 }) \leq L_i \sum_{k=(t-s+1)_+}^{t} \|\mathbf{x}(k+1) - \mathbf{x}(k)\|. &
\end{aligned} \tag{C.11}$$

Combining the two separate cases in Equation (C.10) and Equation (C.11) above we have for all t:

$$\|\mathbf{u}(t+1) + \nabla f(\mathbf{x}(t+1))\| \leq (\sqrt{p}/\eta + 2L) \sum_{k=(t-2s)_+}^{t} \|\mathbf{x}(k+1) - \mathbf{x}(k)\|, \tag{C.12}$$

where of course $\mathbf{u}(t+1) = \big(u_1(t+1), \ \ldots \ , \ u_p(t+1)\big) \in \partial g(\mathbf{x}(t+1))$ and we artificially introduce $\sqrt{p}$ for convenience of subsequent proof. Therefore, from Item 2 we deduce

$$\lim_{t\to\infty} \operatorname{dist}_{\partial F(\mathbf{x}(t+1))}(\mathbf{0}) \to 0. \tag{C.13}$$

Next we deal with the function value convergence in Equation (C.8). For any $\hat{t}_i \in T_i$, using Equation (C.4) with $z = x_i^*$ we have

$$\begin{aligned} g_i(x_i(\hat{t}_i+1)) &+ \frac{1}{2\eta}\left\|x_i(\hat{t}_i+1) - x_i(\hat{t}_i) + \eta\nabla_i f\big(\mathbf{x}^i(\hat{t}_i)\big)\right\|^2 \\ &\le g_i\big(x_i^*\big) + \frac{1}{2\eta}\left\|x_i^* - x_i(\hat{t}_i) + \eta\nabla_i f\big(\mathbf{x}^i(\hat{t}_i)\big)\right\|^2, \end{aligned}$$

which, after rearrangement, yields

$$\begin{aligned} g_i(x_i(\hat{t}_i+1)) \le & g_i\big(x_i^*\big) + \frac{1}{2\eta}\|x_i^* - x_i(\hat{t}_i)\|^2 - \frac{1}{2\eta}\|x_i(\hat{t}_i+1) - x_i(\hat{t}_i)\|^2 \\ & + \langle x_i^* - x_i(\hat{t}_i+1), \nabla_i f(\mathbf{x}^i(\hat{t}_i))\rangle \\ = & g_i\big(x_i^*\big) + \frac{1}{2\eta}\|x_i^* - x_i(\hat{t}_i)\|^2 - \frac{1}{2\eta}\|x_i(\hat{t}_i+1) - x_i(\hat{t}_i)\|^2 \\ & + \langle x_i^* - x_i(\hat{t}_i+1), \nabla_i f(\mathbf{x}^*)\rangle \\ & + \langle x_i^* - x_i(\hat{t}_i+1), \nabla_i f(\mathbf{x}^i(\hat{t}_i)) - \nabla_i f(\mathbf{x}^*)\rangle. \end{aligned} \tag{C.14}$$

We wish to deduce from the above inequality that $g_i(x_i(\hat{t}_i+1)) \to g(x_i^*)$, but we need a uniformization device to remove the dependence on i (hence removing the condition $\hat{t}_i \in T_i$). Observing from Item 2 that

$$\lim_{m\to\infty} \max_{t\in[t_m-s, t_m+s]} \|\mathbf{x}(t+1) - \mathbf{x}^*\| \to 0, \qquad \lim_{m\to\infty} \max_{t\in[t_m-s, t_m+s]} \|\mathbf{x}^i(t+1) - \mathbf{x}^*\| \to 0. \tag{C.15}$$

By Assumption 4.2.3, $[t_m - s, t_m + s] \cap T_i \neq \emptyset$ for all i, using Item 2 again and the Lipschitz continuity of ∇f, we deduce from Equation (C.14) that

$$\limsup_{m\to\infty} \max_{t\in[t_m-s, t_m+s]\cap T_i} g_i(x_i(t+1)) \le g_i(x_i^*). \tag{C.16}$$

Since each machine must update at least once on the intervals $[t_m - s, t_m]$ and $[t_m, t_m + s]$, let $\hat{t}_m^i$ be the largest element of $[t_m - s, t_m] \cap T_i$. Then from the previous inequality we have

$$\limsup_{m\to\infty} \max_{t\in[\hat{t}_m^i, t_m+s]\cap T_i} g_i(x_i(t+1)) \le g_i(x_i^*). \tag{C.17}$$

Since $g_i(x_i(t+1)) = g_i(x_i(t))$ if $t \notin T_i$ and $\hat{t}_m^i \in T_i$, it follows that

$$\max_{t\in[t_m, t_m+s]} g_i(x_i(t+1)) \le \max_{t\in[\hat{t}_m^i, t_m+s]\cap T_i} g_i(x_i(t+1)), \tag{C.18}$$

hence

$$\limsup_{m\to\infty} \max_{t\in[t_m, t_m+s]} g_i(x_i(t+1)) \le g_i(x_i^*). \tag{C.19}$$

Choose any sequence k_m such that $k_m \in [t_m, t_m + s]$. Since $\mathbf{x}(t_m) \to \mathbf{x}^*$, from Item 2 it is clear that

$$\mathbf{x}(k_m + 1) \to \mathbf{x}^*. \tag{C.20}$$

From Equation (C.19) we know for all i, $\limsup_{m\to\infty} g_i(x_i(k_m+1)) \le g_i(x_i^*)$ while using closedness of the function g_i we have

$$\liminf_{m\to\infty} g_i(x_i(k_m+1)) \ge g_i(x_i^*)$$

, and thus in fact

$$\lim_{m\to\infty} g_i(x_i(k_m+1)) = g_i(x_i^*)$$

Since f is continuous, we know

$$\lim_{m\to\infty} F(\mathbf{x}(k_m+1)) = F(\mathbf{x}^*). \tag{C.21}$$

Lastly, combining Equation (C.13), Equation (C.20) and Equation (C.21), it follows from Definition 4.1 that $\mathbf{x}^* \in \operatorname{crit} F$. □

C.2 Proof of Theorem 4.2

Theorem 4.2 (Finite Length). *Let Assumption 4.1, 4.2, 4.3 and 4.4 hold, and apply msPG to problem* (P). *If the step size $\eta < (L_f + 2Ls)^{-1}$ and $\{\mathbf{x}(t)\}$ is bounded, then*

$$\textstyle\sum_{t=0}^{\infty} \|\mathbf{x}(t+1) - \mathbf{x}(t)\| < \infty, \tag{4.12}$$

$$\forall i = 1, \ldots, p,\ \textstyle\sum_{t=0}^{\infty} \|\mathbf{x}^i(t+1) - \mathbf{x}^i(t)\| < \infty. \tag{4.13}$$

Furthermore, $\{\mathbf{x}(t)\}$ and $\{\mathbf{x}^i(t)\}, i = 1, ..., p$, converge to the same critical point of F.

Our proof requires the following simple uniformization of the KŁ inequality in Definition 4.4:
Lemma C.1 (Uniformized KŁ inequality, [25, Lemma 6]). *Let h be a KŁ function and $\Omega \subset \operatorname{dom} h$ be a compact set. If h is constant on Ω, then there exist $\varepsilon, \lambda > 0$ and a function φ as in Definition 4.4, such that for all $\bar{\mathbf{x}} \in \Omega$ and all $\mathbf{x} \in \{\mathbf{x} \in \mathbb{R}^d : \operatorname{dist}_\Omega(\mathbf{x}) < \varepsilon\} \cap [\mathbf{x} : h(\bar{\mathbf{x}}) < h(\mathbf{x}) < h(\bar{\mathbf{x}}) + \lambda]$, one has*

$$\varphi'(h(\mathbf{x}) - h(\bar{\mathbf{x}})) \cdot \operatorname{dist}_{\partial h(\mathbf{x})}(\mathbf{0}) \ge 1.$$

The proof of this Lemma is the usual covering argument.

Proof of Theorem 4.2. We first show that if the global sequence has finite length (i.e. (4.12)) then the local sequences also have finite lenggth (i.e. (4.13)). Indeed,

$$\begin{aligned}\|\mathbf{x}^i(t+1) - \mathbf{x}^i(t)\| \le& \|\mathbf{x}^i(t+1) - \mathbf{x}(t+1)\| + \|\mathbf{x}(t+1) - \mathbf{x}(t)\| + \|\mathbf{x}(t) - \mathbf{x}^i(t)\| \\ (\text{ Equation (C.2) }) \le& \|\mathbf{x}(t+1) - \mathbf{x}(t)\| \ + \sum_{k=(t+1-s)_+}^{t} \|\mathbf{x}(k+1) - \mathbf{x}(k)\| \\ &+ \sum_{k=(t-s)_+}^{t-1} \|\mathbf{x}(k+1) - \mathbf{x}(k)\|.\end{aligned}$$

Therefore, summing from $t = 0$ to $t = n$:

$$\sum_{t=0}^{n} \|\mathbf{x}^i(t+1) - \mathbf{x}^i(t)\| \leq \sum_{t=0}^{n} \left[\sum_{k=(t+1-s)_+}^{t} \|\mathbf{x}(k+1) - \mathbf{x}(k)\| + \sum_{k=(t-s)_+}^{t} \|\mathbf{x}(k+1) - \mathbf{x}(k)\| \right]$$
$$\leq (2s+1) \sum_{t=0}^{n} \|\mathbf{x}(t+1) - \mathbf{x}(t)\|.$$

Letting n tend to infinity we have (4.12) $\implies$ (4.13).

From Theorem 4.1 we know the limit points of $\{\mathbf{x}(t)\}$ and $\{\mathbf{x}^i(t)\}, i = 1, \ldots, p$, coincide, and they are critical points of F.

The only thing left to prove is the finite length property of the global sequence $\mathbf{x}(t)$. If for all large t we have $\mathbf{x}(t+1) = \mathbf{x}(t)$ then the conclusion is trivial. On the other hand, we can remove all iterations t with $\mathbf{x}(t+1) = \mathbf{x}(t)$, without affecting the length of the trajectory. Thus, in the following we assume for all (large) t we have $\mathbf{x}(t+1) \neq \mathbf{x}(t)$. Thanks to Assumption 4.3 and Assumption 4.1.1, it is then clear that the objective value $F(\mathbf{x}(t))$ is strictly decreasing to a limit F^*. Since $\{\mathbf{x}(t)\}$ is assumed to be bounded, the limit point set $\Omega := \omega(\{\mathbf{x}(t)\})$ is nonempty and compact. Obviously for any $\mathbf{x}^* \in \Omega$ we have $F(\mathbf{x}^*) = F^*$. Fix any $\epsilon > 0$, clearly for t sufficiently large we have[2] $\text{dist}_\Omega(\mathbf{x}(t)) \leq \epsilon$. We now have all ingredients to apply the uniformized KŁ inequality in Lemma C.1, which implies that for all sufficiently large t, there exists a continuous and concave function φ (with additional properties listed in Definition 4.4) such that

$$\varphi'\big(F(\mathbf{x}(t)) - F^*\big) \cdot \text{dist}_{\partial F(\mathbf{x}(t))}(\mathbf{0}) \geq 1. \tag{C.22}$$

Since φ is concave, we obtain

$$\begin{aligned} \Delta_{t,t+1} &:= \varphi\big(F(\mathbf{x}(t)) - F^*\big) - \varphi\big(F(\mathbf{x}(t+1)) - F^*\big) \\ &\geq \varphi'\big(F(\mathbf{x}(t)) - F^*\big)\big(F(\mathbf{x}(t)) - F(\mathbf{x}(t+1))\big) \\ \text{(Assumption 4.3, Equation (C.22))} &\geq \frac{\alpha\|\mathbf{x}(t+1) - \mathbf{x}(t)\|^2}{\text{dist}_{\partial F(\mathbf{x}(t))}(\mathbf{0})}. \end{aligned} \tag{C.23}$$

It is clear that the function φ (composed with F) serves as a Lyapunov function. To proceed, we need to upper bound the subdifferential $\partial F(\mathbf{x}(t))$, which has been painstakingly dealt with in the proof of Theorem 4.1.

Using the inequality $2\sqrt{ab} \leq a + b$ for positive numbers we obtain from Equation (C.23): for t sufficiently large,

$$2\|\mathbf{x}(t+1) - \mathbf{x}(t)\| \leq \tfrac{\delta}{\alpha}\Delta_{t,t+1} + \tfrac{1}{\delta}\text{dist}_{\partial F(\mathbf{x}(t))}(\mathbf{0}),$$

---

[2] This is true for any bounded sequence, and we provide a proof for completeness: Suppose not, then there exists $\epsilon > 0$ such that for all n there exists a $t \geq n$ such that $\text{dist}_\Omega(\mathbf{x}(t)) > \epsilon$. Thus, we can extract a subsequence $\{\mathbf{x}(t_m)\}$ such that $\text{dist}_\Omega(\mathbf{x}(t_m)) > \epsilon$. However, since $\{\mathbf{x}(t)\}$ is bounded, we can extract a further subsequence, say $\{\mathbf{x}(t_{m_n})\}$, that converges, i.e. $\text{dist}_\Omega(\mathbf{x}(t_{m_n})) \to 0$, contradiction.

where $\delta > 0$ will be fixed later. Summing the above inequality over t from m (sufficiently large) to n:

$$
\begin{aligned}
2\sum_{t=m}^{n}\|\mathbf{x}(t+1)-\mathbf{x}(t)\| \leq & \sum_{t=m}^{n}\frac{\delta}{\alpha}\Delta_{t,t+1}+\sum_{t=m}^{n}\frac{1}{\delta}\mathrm{dist}_{\partial F(\mathbf{x}(t))}(\mathbf{0}) \\
\text{(telescoping and Equation (C.12))} \leq & \frac{\delta}{\alpha}\varphi\big(F(\mathbf{x}(m))-F^*\big) \\
& +\sum_{t=m}^{n}\frac{\sqrt{p}/\eta+2L}{\delta}\sum_{k=(t-2s)_+}^{t}\|\mathbf{x}(k+1)-\mathbf{x}(k)\| \\
\leq & \frac{\delta}{\alpha}\varphi\big(F(\mathbf{x}(m))-F^*\big) \\
& +\frac{(2s+1)(\sqrt{p}/\eta+2L)}{\delta}\sum_{k=(m-2s)_+}^{m-1}\|\mathbf{x}(k+1)-\mathbf{x}(k)\| \\
& +\frac{(2s+1)(\sqrt{p}/\eta+2L)}{\delta}\sum_{t=m}^{n}\|\mathbf{x}(t+1)-\mathbf{x}(t)\|.
\end{aligned}
$$

Setting $\delta = (2s+1)(\sqrt{p}/\eta+2L)$ and rearranging:

$$
\begin{aligned}
\sum_{t=m}^{n}\|\mathbf{x}(t+1)-\mathbf{x}(t)\| \leq & \frac{(2s+1)(\sqrt{p}/\eta+2L)}{\alpha}\varphi\big(F(\mathbf{x}(m))-F^*\big) \\
& +\sum_{k=(m-2s)_+}^{m-1}\|\mathbf{x}(k+1)-\mathbf{x}(k)\|
\end{aligned}
$$

Since the right-hand side is finite and does not depend on n, letting n tend to infinity completes our proof for Equation (4.12). $\square$

C.3 Proof of Lemma 4.1

Lemma 4.1. *Assume* $\forall t, i, t \in T_i$. *Let the step size* $\eta < \frac{\rho-1}{4C\rho}\frac{\sqrt{\rho}-1}{\sqrt{\rho}^{s+1}-1}$ *for any* $\rho > 1$ *and all* $U_i, i = 1, ..., p$ *be proximal-Lipschitz continuous, then the sequences* $\{\mathbf{x}(t)\}$ *and* $\{\mathbf{x}^i(t)\}, i = 1, ..., p$, *have finite length.*

Follow the same argument of equation (C.2), one can bound $\|\mathbf{x}^i(t) - \mathbf{x}^i(t+1)\|$ similarly as

$$\|\mathbf{x}^i(t)-\mathbf{x}^i(t+1)\| \leq \sum_{k=(t-s)_+}^{t}\|\mathbf{x}(k+1)-\mathbf{x}(k)\|. \tag{C.24}$$

Proof.

$$\begin{aligned}\|\mathbf{x}^i(t)-\mathbf{x}^i(t+1)\|^2 &= \sum_{j=1}^{p}\|x_j(\tau_j^i(t))-x_j(\tau_j^i(t+1))\|^2\\ &\le \sum_{j=1}^{p}\left(\sum_{k=\tau_j^i(t)}^{\tau_j^i(t+1)-1}\|x_j(k+1)-x_j(k)\|\right)^2\\ &\le \sum_{j=1}^{p}\left(\sum_{k=(t-s)_+}^{t}\|x_j(k+1)-x_j(k)\|\right)^2,\end{aligned}$$

and the rest of the proof is completely similar to Eq. (C.1). □

Since Assumption 4.6 holds for all $t > t_L$, we prove the lemma by considering two complementary cases.

Case 1: There exists a $\hat{t} > t_L$ such that

$$\sum_{k=(\hat{t}-s)_+}^{\hat{t}}\|\mathbf{x}(k+1)-\mathbf{x}(k)\| \le \frac{\sqrt{\rho}^{s+1}-1}{\sqrt{\rho}-1}\|\mathbf{x}(\hat{t}+1)-\mathbf{x}(\hat{t})\|$$

Case 2: For all $t > t_L$ case 1 fails.

We will show that case 1 leads to the sufficient decrease property in Assumption 4.3 for all large t, case 2 leads to the finite length of the models.

Case 1: $\hat{t}$ exists.

We start by proving the following lemma.

Lemma C.2. *With Assumption 4.6 and the existence of $\hat{t}$. Set $\eta^{-1} > \frac{4C\rho}{\rho-1}\frac{\sqrt{\rho}^{s+1}-1}{\sqrt{\rho}-1}$, then it holds for all $t > \hat{t}$ that*

$$\|\mathbf{x}(\hat{t}+1)-\mathbf{x}(\hat{t})\| \le \sqrt{\rho}\|\mathbf{x}(\hat{t}+2)-\mathbf{x}(\hat{t}+1)\|.$$

Proof. Using the inequality $\|a\|_2^2 - \|b\|_2^2 \le 2\|a\|\|a-b\|$, we have for all $t > \hat{t} > t_L$

$$\begin{aligned}
&\|\mathbf{x}(t+1)-\mathbf{x}(t)\|_2^2 - \|\mathbf{x}(t+2)-\mathbf{x}(t+1)\|_2^2\\
&\le 2\,\|\mathbf{x}(t+1)-\mathbf{x}(t)\|\,\|(\mathbf{x}(t+1)-\mathbf{x}(t)) - (\mathbf{x}(t+2)-\mathbf{x}(t+1))\|\\
\text{(no skip of update)} &= 2\,\|\mathbf{x}(t+1)-\mathbf{x}(t)\| \left\|\sum_{i=1}^{p} U_i(\mathbf{x}^i(t)) - \sum_{i=1}^{p} U_i(\mathbf{x}^i(t+1))\right\|\\
&\le 2\,\|\mathbf{x}(t+1)-\mathbf{x}(t)\| \sum_{i=1}^{p} \left\|U_i(\mathbf{x}^i(t)) - U_i(\mathbf{x}^i(t+1))\right\|\\
\text{(Assumption 4.6)} &\le 2\,\|\mathbf{x}(t+1)-\mathbf{x}(t)\| \left(\sum_{i=1}^{p} C_i\eta\|\mathbf{x}^i(t)-\mathbf{x}^i(t+1)\|\right)\\
\text{(equation (C.24))} &\le 2\,\|\mathbf{x}(t+1)-\mathbf{x}(t)\| \left(\sum_{i=1}^{p} C_i\eta\left[\sum_{k=(t-s)_+}^{t}\|\mathbf{x}(k+1)-\mathbf{x}(k)\|\right]\right)\\
&= 2C\eta\,\|\mathbf{x}(t+1)-\mathbf{x}(t)\|\left[\sum_{k=(t-s)_+}^{t}\|\mathbf{x}(k+1)-\mathbf{x}(k)\|\right] \qquad \text{(C.25)}
\end{aligned}$$

Now we use an induction argument. Since there exists $\hat{t} > t_L$ such that $\sum_{k=(\hat{t}-s)_+}^{\hat{t}}\|\mathbf{x}(k+1)-\mathbf{x}(k)\| \le \frac{\sqrt{\rho}^{s+1}-1}{\sqrt{\rho}-1}\|\mathbf{x}(\hat{t}+1)-\mathbf{x}(\hat{t})\|$, then set $t=\hat{t}$ in the above inequality, we obtain

$$\begin{aligned}
\left\|\mathbf{x}(\hat{t}+1)-\mathbf{x}(\hat{t})\right\|_2^2 - \left\|\mathbf{x}(\hat{t}+2)-\mathbf{x}(\hat{t}+1)\right\|_2^2 &\le 2C\eta\frac{\sqrt{\rho}^{s+1}-1}{\sqrt{\rho}-1}\left\|\mathbf{x}(\hat{t}+1)-\mathbf{x}(\hat{t})\right\|_2^2\\
\text{(choice of }\eta) &\le \left(1-\frac{1}{\rho}\right)\left\|\mathbf{x}(\hat{t}+1)-\mathbf{x}(\hat{t})\right\|^2.
\end{aligned}$$

After rearranging terms we conclude $\|\mathbf{x}(\hat{t}+1)-\mathbf{x}(\hat{t})\| \le \sqrt{\rho}\|\mathbf{x}(\hat{t}+2)-\mathbf{x}(\hat{t}+1)\|$. Now we assume this relationship holds up to t ($t > \hat{t}$), then (C.25) becomes

$$\begin{aligned}
\|\mathbf{x}(t+1)-\mathbf{x}(t)\|_2^2 - \|\mathbf{x}(t+2)-\mathbf{x}(t+1)\|_2^2 &\le 2C\eta\frac{\sqrt{\rho}^{s+1}-1}{\sqrt{\rho}-1}\|\mathbf{x}(t+1)-\mathbf{x}(t)\|_2^2\\
\text{(choice of }\eta) &\le \left(1-\frac{1}{\rho}\right)\|\mathbf{x}(t+1)-\mathbf{x}(t)\|_2^2
\end{aligned}$$

we obtain $\|\mathbf{x}(t+1)-\mathbf{x}(t)\| \le \sqrt{\rho}\|\mathbf{x}(t+2)-\mathbf{x}(t+1)\|$. This completes the lemma. □

With this bound, inequality (C.7) can be further bounded for $t > \hat{t}$ as

$$\begin{aligned}
F\big(\mathbf{x}(t+1)\big) - F\big(\mathbf{x}(t)\big) \le& \tfrac{1}{2}(L_f - 1/\eta)\|\mathbf{x}(t+1)-\mathbf{x}(t)\|^2\\
&+ L\|\mathbf{x}(t+1)-\mathbf{x}(t)\|\cdot\sum_{k=(t-s)_+}^{t-1}\|\mathbf{x}(k+1)-\mathbf{x}(k)\|.\\
\le& -\alpha\|\mathbf{x}(t+1)-\mathbf{x}(t)\|^2,
\end{aligned}$$

where

$$\begin{aligned}\alpha &\geq \frac{\eta^{-1}-L_f}{2} - \frac{L\sqrt{\rho}(1-\sqrt{\rho}^s)}{1-\sqrt{\rho}} \\ (C > L, \rho > 1) &\geq \frac{2L\rho}{\rho-1}\frac{\sqrt{\rho}^{s+1}-1}{\sqrt{\rho}-1} - \frac{L(\sqrt{\rho}^{s+1}-1)}{\sqrt{\rho}-1} - \frac{L_f}{2} \\ &\geq \frac{L(\sqrt{\rho}^{s+1}-1)}{\sqrt{\rho}-1}(\frac{2\rho}{\rho-1}-1) - \frac{L_f}{2} \\ (L > L_f, \rho > 1) &> 0,\end{aligned}$$

This proves the sufficient decrease for all $t > \hat{t}$ of the objective value. Hence, the finite length property of the models follows from Theorem 4.2.

Case 2: $\hat{t}$ does not exist

In this case we have for all $t > t_L$ it holds that $\sum_{k=(t-s)_+}^{t} \|\mathbf{x}(k+1)-\mathbf{x}(k)\| \geq \frac{\sqrt{\rho}^{s+1}-1}{\sqrt{\rho}-1}\|\mathbf{x}(t+1)-\mathbf{x}(t)\|$. Set $D = \frac{\sqrt{\rho}^{s+1}-1}{\sqrt{\rho}-1}$ and sum the inequality over t from t_L to n yields

$$\begin{aligned}\sum_{k=t_L}^{n} \|\mathbf{x}(k+1)-\mathbf{x}(k)\| &< \frac{1}{D}\sum_{t=t_L}^{n}\sum_{k=(t-s)_+}^{t} \|\mathbf{x}(k+1)-\mathbf{x}(k)\| \\ &< \frac{s+1}{D}\sum_{t=(t_L-s)_+}^{n} \|\mathbf{x}(t+1)-\mathbf{x}(t)\|,\end{aligned}$$

which after rearranging terms becomes

$$(1-\frac{s+1}{D})\sum_{t=t_L}^{n} \|\mathbf{x}(t+1)-\mathbf{x}(t)\| \leq \frac{s+1}{D}\sum_{t=(t_L-s)_+}^{t_L-1} \|\mathbf{x}(t+1)-\mathbf{x}(t)\|.$$

Since $D = \frac{\sqrt{\rho}^{s+1}-1}{\sqrt{\rho}-1} > s+1$ for $\rho > 1$ and t_L is finite, the right hand side of the above inequality is finite, and the left hand side has positive coefficient. Thus the above inequality implies

$$\sum_{t=0}^{n} \|\mathbf{x}(t+1)-\mathbf{x}(t)\| < +\infty.$$

Enlarge $n \to \infty$ gives the finite length property of the global model. By the proof of Section C.2, we know the finite length of global model implies the finite length of all local models.

C.4 Proof of Example 1

We proof case by case, and the scaled version $\gamma g(\mathbf{x}), \gamma > 0$ trivially follows from the same argument.

Cases $g = 0,\ g = \frac{1}{2}\|\cdot\|^2$**:**

When $g = 0$, the update operator in Eq. (4.10) becomes $U_i(\mathbf{x}^i(t)) = -\eta\nabla_i f(\mathbf{x}^i(t))$, which is ηL_i Lipschitz due to Assumption 4.1.2 .

When $g = \frac{1}{2}\|\cdot\|^2$, the update operator becomes for $i = 1, ..., p$

$$U_i(\mathbf{x}^i(t)) = \mathrm{prox}^{\eta}_{\frac{1}{2}\|\cdot\|_2^2}(x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t))) - x_i(t) = -\frac{1}{1+\eta^{-1}}\left(x_i(t) + \nabla_i f(\mathbf{x}^i(t))\right).$$

With which we have

$$\begin{aligned}\|U_i(\mathbf{x}^i(t+1)) - U_i(\mathbf{x}^i(t))\| &\le \frac{1}{1+\eta^{-1}}\|(x_i(t+1) - x_i(t)) + (\nabla_i f(\mathbf{x}^i(t+1)) - \nabla_i f(\mathbf{x}^i(t)))\| \\ &\le \eta(1+L_i)\|\mathbf{x}^i(t+1) - \mathbf{x}^i(t)\|\end{aligned}$$

Cases $g = \ \|\cdot\|_0,\ \|\cdot\|_0 + \|\cdot\|^2,\ \|\cdot\|_{0,2},\ \|\cdot\|_{0,2} + \|\cdot\|^2$**:** For the non-overlapping group norms, we assign each machine a subset of groups of coordinates.

Consider $g = \|\cdot\|_0$, its proximal map on i-th coordinate can be expressed as

$$\mathrm{prox}^{\eta}_{g_i}(z_i) = \begin{cases} z_i, \text{ if } |z_i| > \sqrt{2\eta} \\ 0, \text{ otherwise}\end{cases}$$

The mapping contains a hard threshold, i.e., it filters out those coordinates with magnitude less than $\sqrt{2\eta}$. This implies that any change of the support set of $\mathrm{prox}^{\eta}_{g_i}(z_i)$ will induce a jump of magnitude of at least $\sqrt{2\eta}$. On the other hand, the second assertion of Theorem 4.1 imply that $\lim_{t\to\infty}\|x_i(t+2) - x_i(t+1)\| = 0$, which by local update can be expressed as

$$\lim_{t\to\infty}\|\mathrm{prox}^{\eta}_{g_i}(x_i(t+1) - \eta\nabla_i f(\mathbf{x}^i(t+1))) - \mathrm{prox}^{\eta}_{g_i}(x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t)))\| = 0.$$

Hence by the above equation and the jump of proximal map, the support Ω of $\mathrm{prox}^{\eta}_{g_i}(x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t)))$ (i.e., $x_i(t+1)$) must remain stable for all t sufficiently large. Moreover, the proximal map reduces to identity operator on the support set Ω. Thus, for all t sufficiently large we have

$$\begin{aligned}\|U_i(\mathbf{x}^i(t+1)) - U_i(\mathbf{x}^i(t))\| =& \|\mathrm{prox}^{\eta}_{g_i}(x_i(t+1) - \eta\nabla_i f(\mathbf{x}^i(t+1))) - x_i(t+1) \\ & - \mathrm{prox}^{\eta}_{g_i}(x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t))) - x_i(t)\| \\ \text{(support on } \Omega) =& \|[\mathrm{prox}^{\eta}_{g_i}(x_i(t+1) - \eta\nabla_i f(\mathbf{x}^i(t+1))) - x_i(t+1) \\ & - \mathrm{prox}^{\eta}_{g_i}(x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t))) - x_i(t)]_{\Omega}\| \\ (\mathrm{prox}^{\eta}_{g} \text{ is identity on } \Omega) \le& \ \|\eta\nabla_i f(\mathbf{x}^i(t)) - \eta\nabla_i f(\mathbf{x}^i(t+1))\| \\ \le& \ \eta L_i\|\mathbf{x}^i(t+1) - \mathbf{x}^i(t)\|.\end{aligned}$$

Hence the operator is eventually $\mathcal{O}(\eta)$ Lipschitz.

Next we consider (without loss of generality) $g = \|\cdot\|_0 + \frac{\lambda}{2}\|\cdot\|^2$ where $\lambda > 0$. The proximal map on i-th coordinate is

$$\mathrm{prox}_{g_i}^{\eta}(z_i) = \begin{cases} z_i, \text{ if } |z_i| > \sqrt{2(\eta + \eta^2\lambda)} \\ 0, \text{ otherwise} \end{cases}$$

Thus, the mapping also contains a hard threshold. Following similar argument as previous case, we conclude that the support Ω of $\mathrm{prox}_{g_i}^{\eta}(x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t)))$ (i.e., $x_i(t+1)$) must remain stable for all t sufficiently large, and the proximal map reduces to identity operator on Ω. Consequently, the operator $U_i(\mathbf{x}^i(t))$ is $\mathcal{O}(\eta)$ Lipschitz for all t large.

The proof of group norms $g = \|\cdot\|_{0,2}, \|\cdot\|_{0,2} + \|\cdot\|^2$ then follows by realizing that the proximal maps have hard threshold on group support.

Cases $g = \|\cdot\|_1,\ \|\cdot\|_1 + \|\cdot\|^2$ with eventual stable support set of $\{\mathbf{x}(t)\}$:

For these two cases we assume that the support set of $\{\mathbf{x}(t)\}$ remains unchanged for all large t.

We just need to consider $g = \|\cdot\|_1 + \frac{\lambda}{2}\|\cdot\|^2, \lambda \geq 0$. Its proximal map on vector z_i has the form

$$\mathrm{prox}_g^{\eta}(z_i) = \tfrac{1}{1+\eta\lambda}\mathrm{sgn}(z_i)\left(|z_i| - \eta\right)_+ .$$

Since the support set Ω of $\mathbf{x}(t)$ (i.e. $\mathrm{prox}_g^{\eta}(x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t))))$ is assumed to be stable after some t_L, the above soft-thresholding operator ensures that $|x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t))|_\Omega > \eta$ for all large t, and we obtain

$$\begin{aligned} U_i(\mathbf{x}^i(t)) &= [x_i(t+1) - x_i(t)]_\Omega = [\mathrm{prox}_g^{\eta}\left(x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t))\right) - x_i(t)]_\Omega \\ &= (1+\eta\lambda)^{-1}\left[-\eta\nabla_i f(\mathbf{x}^i(t)) - \eta\mathrm{sgn}\left(x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t))\right)\right]_\Omega - \frac{\eta\lambda}{1+\eta\lambda}[x_i(t)]_\Omega \end{aligned}$$

On the other hand, Theorem 4.1.2 and the Lispchitz gradient of f implies

$$\lim_{t\to\infty} \left\|\left[x_i(t+1) - \eta\nabla_i f(\mathbf{x}^i(t+1))\right] - \left[x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t))\right]\right\| = 0.$$

Then $[\mathrm{sgn}(x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t))]_\Omega$ must eventually remain constant, since otherwise the condition $|x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t))|_\Omega > \eta$ will induce a change of $|x_i(t) - \eta\nabla_i f(\mathbf{x}^i(t)|$ to be at least 2η and violate the above asymptotic condition. In summary, for all large t we have

$$U_i(\mathbf{x}^i(t)) = (1+\eta\lambda)^{-1}\left[-\eta\nabla_i f(\mathbf{x}^i(t)) - \mathrm{Const}\right]_\Omega - \frac{\eta\lambda}{1+\eta\lambda}[x_i(t)]_\Omega$$

which further implies that

$$\begin{aligned} \|U_i(\mathbf{x}^i(t+1)) - U_i(\mathbf{x}^i(t))\| \leq & \|(1+\eta\lambda)^{-1}\left[\eta\nabla_i f(\mathbf{x}^i(t+1)) - \eta\nabla_i f(\mathbf{x}^i(t))\right]_\Omega \\ & + \frac{\eta\lambda}{1+\eta\lambda}[x_i(t+1) - x_i(t)]_\Omega\| \\ \leq & \eta L_i\|\mathbf{x}^i(t+1) - \mathbf{x}^i(t)\| + \eta\lambda\|\mathbf{x}^i(t+1) - \mathbf{x}^i(t)]\| \\ \leq & \eta(L_i + \lambda)\|\mathbf{x}^i(t+1) - \mathbf{x}^i(t)\|. \end{aligned}$$

C.5 Proof of Theorem 4.3

Theorem 4.3 (Global rate of convergence)**.** *If the finite length property in Theorem 4.2 holds, then*

1. $\sum_{t=0}^{\infty} \|\mathbf{e}(t)\| < \infty$*;*
2. $F(\frac{1}{t}\sum_{k=1}^{t} \mathbf{x}(k)) - \inf F \le O\left(t^{-1}\right)$.

Proof. For the first assertion, note that for any n:

$$\begin{aligned}
\sum_{t=0}^{n} \|\mathbf{e}(t)\| =& \eta \sum_{t=0}^{n} ||\big(\nabla_1 f(\mathbf{x}^1(t)) - \nabla_1 f(\mathbf{x}(t)), \\
& \quad \dots , \ \nabla_p f(\mathbf{x}^p(t)) - \nabla_p f(\mathbf{x}(t))\big)|| \\
\text{(triangle inequality, Assumption 4.1.2)} \ \le& \eta \sum_{t=0}^{n} \sum_{i=1}^{p} L_i \|\mathbf{x}(t) - \mathbf{x}^i(t)\| \\
\text{(Equation (C.2))} \ \le& \eta \sum_{t=0}^{n} \left(\sum_{i=1}^{p} L_i\right) \sum_{k=(t-s)_+}^{t-1} \|\mathbf{x}(k+1) - \mathbf{x}(k)\| \\
=& L\eta \sum_{t=0}^{n} \sum_{k=(t-s)_+}^{t-1} \|\mathbf{x}(k+1) - \mathbf{x}(k)\| \\
\le& Ls\eta \sum_{t=0}^{n-1} \|\mathbf{x}(t+1) - \mathbf{x}(t)\| .
\end{aligned}$$

Letting n tend to infinity we obtain

$$\sum_{t=0}^{\infty} \|\mathbf{e}(t)\| \le Ls\eta \sum_{t=0}^{\infty} \|\mathbf{x}(t+1) - \mathbf{x}(t)\| < \infty.$$

For the second assertion, we first recall from [127] that the inexact proximal gradient algorithm in Equation (4.15) has the following bound, provided that F is convex:

$$F\left(\frac{1}{t}\sum_{k=1}^{t} \mathbf{x}(k)\right) - F^* \le \frac{(\|\mathbf{x}(0) - \mathbf{x}^*\| + 2A_t)^2}{2t\eta}, \ \text{ where } \ A_t = \sum_{k=0}^{t} \eta \|\mathbf{e}(k)\|.$$

The second assertion thus follows from the first one (assuming convexity). □

Appendix D

Appendix for Chapter 5

D.1 Convergence analysis

We provide a self-contained convergence proof in this section. The skeleton of our convergence proof follow closely from [82] and [69]. There are a few subtle modification and improvements that we need to add due to our weaker definition of approximate oracle call that is nearly correct only in expectation. The delayed convergence is new and interesting for the best of our knowledge, which uses a simple result in "load balancing" [110].

Note that for the cleanness of the presentation, we focus on the primal and primal-dual convergence of the version of the algorithms with pre-defined step sizes and additive approximate subroutine, it is simple to extend the same analysis for line-search variant and multiplicative approximation.

D.1.1 Primal Convergence

Lemma D.1. *Denote the gap between current $f(x^{(k)})$ and the optimal $f(x^*)$ to be $h(x^{(k)})$. The iterative updates in Algorithm 6 (with arbitrary fixed stepsize γ or by the line search) obey*

$$\mathbb{E}h(x^{(k+1)}) \leq (1 - \frac{\gamma\tau}{n})\mathbb{E}h(x^{(k)}) + \frac{\gamma^2(1+\delta)}{2}C_f^{\tau}.$$

where the expectation is taken over the joint randomness all the way to iteration $k+1$.

Proof. Let $x := x^{(k)}$ for notational convenience. We prove the result for Algorithm 6 first. Apply the definition of $C_f^{(S)}$ and then apply the definition of the additive approximation in (5.5), to get

$$
\begin{aligned}
f(x^{(k+1)}_{\text{line}-\text{search}}) \le f(x^{(k+1)}_{\gamma}) &= f(x + \gamma \sum_{i \in S} (s_{[i]} - x_{[i]})) \\
&\le f(x) + \gamma \sum_{i \in S} \langle s_{[i]} - x_{[i]}, \nabla_{[i]} f(x) \rangle + \frac{\gamma^2}{2} C_f^{(S)} \\
&= f(x) + \gamma \langle s_{[S]} - x_{[S]}, \nabla_{[S]} f(x) \rangle + \frac{\gamma^2}{2} C_f^{(S)}
\end{aligned}
$$

Subtract $f(x^*)$ on both sides we get:

$$
h(x^{(k+1)}) \le h(x^{(k)}) + \gamma \langle s_{[S]} - x^{(k)}_{[S]}, \nabla_{[S]} f(x^{(k)}) \rangle + \frac{\gamma^2}{2} C_f^{(S)}
$$

Now take the expectation over the entire history then apply (5.5) and definition of the surrogate duality gap (5.6), we obtain

$$
\begin{aligned}
\mathbb{E}h(x^{(k+1)}) \le & \mathbb{E}h(x^{(k)}) + \mathbb{E}\left\{ \gamma \langle s_{[S]} - x^{(k)}_{[S]}, \nabla_{[S]} f(x^{(k)}) \rangle \right\} + \mathbb{E}\frac{\gamma^2}{2} C_f^{(S)} \\
= & \mathbb{E}h(x^{(k)}) + \gamma \mathbb{E}\left\{ \langle s_{[S]}, \nabla_{[S]} f(x^{(k)}) \rangle - \min_{s \in \mathcal{M}^{(S)}} \langle s, \nabla_{[S]} f(x^{(k)}) \rangle \right\} \\
& - \gamma \mathbb{E}\left\{ \langle x^{(k)}_{[S]}, \nabla_{[S]} f(x^{(k)}) \rangle - \min_{s \in \mathcal{M}^{(S)}} \langle s, \nabla_{[S]} f(x^{(k)}) \right\} + \frac{\gamma^2}{2} C_f^{\tau} \\
\le & \mathbb{E}h(x^{(k)}) + \frac{\gamma^2 \delta}{2} C_f^{\tau} - \gamma \mathbb{E}_{x^k} \mathbb{E}_{S|x^k} \sum_{i \in S} g^{(i)}(x^{(k)}) + \frac{\gamma^2}{2} C_f^{\tau} \\
= & \mathbb{E}h(x^{(k)}) + \frac{\gamma^2 \delta}{2} C_f^{\tau} - \gamma \mathbb{E}_{x^k} \frac{\tau}{n} g(x^{(k)}) + \frac{\gamma^2}{2} C_f^{\tau} \qquad \text{(D.1)} \\
\le & (1 - \frac{\gamma \tau}{n}) \mathbb{E}h(x^{(k)}) + \frac{\gamma^2 (1 + \delta)}{2} C_f^{\tau}.
\end{aligned}
$$

The last inequality follows from the property of the surrogate duality gap $g(x^{(k)}) \ge h(x^{(k)})$ due to the fact that $g(x) \ge f(x) - f^*(\cdot)$. This completes the proof of the descent lemma. □

Now we are ready to state the proof for Theorem 5.1.

Proof of Theorem 5.1. We follow the proof in Theorem C.1 in [82] to prove the statement for Algorithm 6. The difference is that we use a different and carefully chosen sequence of step size.

Take $C = h_0 + n(1 + \delta) C_f^{\tau}$, and denote $\mathbb{E}h(x^{(k)})$ as h_k for short hands. The inequality in Lemma D.1 simplifies to

$$
h_{k+1} \le \left(1 - \frac{\gamma \tau}{n}\right) h_k + \frac{\gamma^2}{2n} C.
$$

Now we will prove $h_k \le \frac{2nC}{\tau^2 k + 2n}$ for $\gamma_k = \frac{2n\tau}{\tau^2 k + 2n}$ by induction. The base case $k = 0$ is trivially true since $C > h_0$. Assuming that the claim holds for k, we apply the induction hypothesis and the above inequality is reduced to

$$\begin{aligned}
h_{k+1} &\leq (1-\frac{\gamma\tau}{n})h_k + \frac{\gamma^2}{2n}C \leq \frac{2nC}{\tau^2 k+2n}\left[1-\frac{\gamma\tau}{n}+\frac{\tau^2 k+2n}{2n}\frac{\gamma^2}{2n}\right] \\
&= \frac{2nC}{\tau^2 k+2n}\left[\frac{\tau^2 k+2n}{\tau^2 k+2n}-\frac{2n\tau}{\tau^2 k+2n}\cdot\frac{\tau}{n}+\frac{(2n\tau)^2}{4n^2(\tau^2 k+2n)}\right] \\
&= \frac{2nC}{\tau^2 k+2n}\cdot\frac{\tau^2 k+2n-\tau^2}{\tau^2 k+2n} \leq \frac{2nC}{\tau^2 k+2n}\cdot\frac{\tau^2 k+2n-\tau^2+\tau^2}{\tau^2 k+2n+\tau^2} \\
&= \frac{2nC}{\tau^2(k+1)+2n}.
\end{aligned}$$

This completes the induction and hence the proof for the primal convergence for Algorithm 6. □

D.1.2 Convergence of the surrogate duality gap

Proof of Theorem 5.2. We mimic the proof in [82, Section C.3] for the analogous result closely, and we will use the same notation for h_k and C as in the proof for primal convergence, moreover denote $g_k = \mathbb{E}g(x^{(k)})$ First from (D.1) in the proof of Lemma D.1, we have

$$h_{k+1} \leq h_k - \frac{\gamma\tau}{n}g_k + \frac{\gamma^2}{2n}C.$$

Rearrange the terms, we get

$$g_k \leq \frac{n}{\gamma\tau}(h_k - h_{k+1}) + \frac{\gamma C}{2\tau}. \tag{D.2}$$

The idea is that if we take an arbitrary convex combination of $\{g_1, ..., g_K\}$, the result will be within the convex hull, namely between the minimum and the maximum, hence proven the existence claim in the theorem. By choosing weight $\rho_k := k/S_K$ where normalization constant $S_K = \frac{K(K+1)}{2}$ and taking the convex combination of both side of (D.2), we have

$$\begin{aligned}
\mathbb{E}(\min_{k\in[K]} g_k) &\leq \sum_{k=0}^{K}\rho_k g_k \leq \frac{n}{\tau}\sum_{k=1}^{K}\rho_k(\frac{h_k}{\gamma_k}-\frac{h_{k+1}}{\gamma_k}) + \sum_{k=0}^{K}\rho_k\gamma_k\frac{C}{2\tau} \\
&= \frac{n}{\tau}(\frac{h_0\rho_0}{\gamma_0} - h_{K+1}\frac{\rho_k}{\gamma_k}) + \frac{n}{\tau}\sum_{k=0}^{K-1}h_{k+1}(\frac{\rho_{k+1}}{\gamma_{k+1}}-\frac{\rho_k}{\gamma_k}) + \sum_{k=0}^{K}\rho_k\gamma_k\frac{C}{2\tau} \\
&\leq \frac{n}{\tau}\sum_{k=0}^{K-1}h_{k+1}(\frac{\rho_{k+1}}{\gamma_{k+1}}-\frac{\rho_k}{\gamma_k}) + \sum_{k=0}^{K}\rho_k\gamma_k\frac{C}{2\tau}
\end{aligned} \tag{D.3}$$

Note that $\rho_0 = 0$, so we simply dropped a negative term in last line. Applying the step size $\gamma_k = 2n\tau/(\tau^2 k + 2n)$, we get

$$\begin{aligned}
\frac{\rho_{k+1}}{\gamma_{k+1}} - \frac{\rho_k}{\gamma_k} &= \frac{k+1}{S_K}\frac{\tau^2(k+1)2n}{2n\tau} - \frac{k}{S_K}\frac{\tau^2 k + 2n}{2n\tau} \\
&= \frac{1}{2nS_K\tau}\left[\tau^2(k+1)^2 + 2n(k+1) - \tau^2 k^2 - 2nk\right] \\
&= \frac{\tau^2(2k+1)+2n}{2nS_K\tau}.
\end{aligned}$$

Plug the above back into (D.3) and use the bound $h_{k+1} \leq 2nC/(\tau^2(k+1)+2n)$, we get

$$\begin{aligned}
\mathbb{E}(\min_{k\in[K]} g_k) &\leq \sum_{k=0}^{K}\rho_k g_k \leq \frac{nC}{\tau^2 S_K}\sum_{k=0}^{K-1}\frac{\tau^2(2k+1)+2n}{2n}\frac{2n}{\tau^2(k+1)+2n} + \sum_{k=0}^{K}\frac{k}{S_K}\frac{2n\tau}{\tau^2 k + 2n}\frac{C}{2\tau} \\
&= \frac{nC}{\tau^2 S_K}\left[\sum_{k=0}^{K-1}(1 + \frac{\tau^2 k}{\tau^2(k+1)+2n}) + \sum_{k=1}^{K}\frac{k\tau^2}{(\tau^2 k + 2n)}\right] \\
&\leq \frac{nC}{\tau^2 S_K}\left[2K + K\right] = \frac{2nC}{\tau^2(K+1)}\cdot 3.
\end{aligned}$$

This completes the proof for $K \geq 1$. □

Proof of Convergence with Delayed Gradient The idea is that we are going to treat the updates calculated from the delayed gradients as an additive error and then invoke our convergence results that allow the oracle to be approximate. We will first present a lemma that we will use for the proof of Theorem 5.4.

Lemma D.2. *Let $x \in \mathcal{M}$, $\|\cdot\|$ be a norm, $\mathrm{Diam}(\mathcal{M})_{\|\cdot\|} \leq D$, L be the gradient Lipschitz constant of f with respect to the given norm $\|\cdot\|$, which has a dual norm $\|\cdot\|_*$. Moreover, let $\tilde{x}$ be at most κ steps away from x and the largest stepsize in the past κ steps, and*

$$\begin{aligned}
s^* &:= \operatorname*{argmin}_{s\in\mathcal{M}}\langle s, \nabla f(x)\rangle \\
\tilde{s} &:= \operatorname*{argmin}_{s\in\mathcal{M}}\langle s, \nabla f(\tilde{x})\rangle
\end{aligned}$$

Then, we have

$$\langle \tilde{s} - x, \nabla f(x)\rangle \leq \langle s^* - x, \nabla f(x)\rangle + \gamma\kappa D^2 L$$

Proof. Because $\tilde{s}$ minimizes $\langle s, \nabla f(\tilde{x})\rangle$ over $s \in \mathcal{M}$ and s^* is feasible, we can write

$$\langle s^* - \tilde{s}, \nabla f(\tilde{x})\rangle \geq 0.$$

Using this and Hölder's inequality, we can write

$$\begin{aligned}
\langle \tilde{s} - x, \nabla f(x)\rangle - \langle s^* - x, \nabla f(x)\rangle &\leq \langle \tilde{s} - s^*, \nabla f(x) - \nabla f(\tilde{x})\rangle \\
&\leq \|\tilde{s} - s^*\|\|\nabla f(\tilde{x}) - \nabla f(x)\|_* \\
&\leq DL\|\tilde{x} - x\|.
\end{aligned}$$

It remains to bound $\|\tilde{x} - x\|$.

$$\|\tilde{x} - x\| = \left\| \tilde{x} - \tilde{x} - \sum_{i=1}^{\kappa} \gamma_{-i}(s_{-i} - x_{-i}) \right\| \leq \gamma\kappa \max_i \|s_{-i} - x_{-i}\| \leq \gamma\kappa D,$$

where we used the fact that x is at most κ steps away from $\tilde{x}$. Assume γ_{-i} is the stepsize used and $\langle s_{-i}, x_{-i}\rangle$ are the actual updates that had been performed in the nearest ith parameter update before we get to x. □

The second lemma that we need is the following.

Lemma D.3. *Let $\mathcal{M}$ be a convex set. Let $x_0 \in \mathcal{M}$. Let m be any positive integer. For $i = 1, ..., m$, let $x_i = x_{i-1} + \gamma_i(s_i - x_{i-1})$ for some $0 \leq \gamma_i \leq 1$ and $s_i \in \mathcal{M}$. Then there exists an $s \in \mathcal{M}$ and $\gamma \leq \sum_{i=1}^{m} \gamma_i$, such that $x_m = \gamma(s - x_0) + x_0$.*

Proof. We prove by induction. When $m = 1$, $s = s_1$ and $\gamma = \gamma_1$. Assume for any $m = k - 1$, that the claim holds assume the condition is true, then by the recursive formula,

$$\begin{aligned}
x_k &= x_{k-1} + \gamma_k(s_k - x_{k-1}) \\
&= x_0 + \gamma(s - x_0) + \gamma_k[s_k - x_0 - \gamma(s - x_0)] \\
&= x_0 - (\gamma + \gamma_k - \gamma_k\gamma)x_0 + (\gamma - \gamma_k\gamma)s + \gamma_k s_k \\
&= x_0 + (\gamma + \gamma_k - \gamma_k\gamma)\left[\frac{\gamma - \gamma_k\gamma}{\gamma + \gamma_k - \gamma_k\gamma}s + \frac{\gamma_k}{\gamma + \gamma_k - \gamma_k\gamma}s_k - x_0\right] \\
&= x_0 + (\gamma + \gamma_k - \gamma_k\gamma)(s' - x_0)
\end{aligned}$$

Note that s' is a convex combination of s_k and s therefore by convexity $s' \in \mathcal{M}$. Substitute $\gamma \leq \sum_{i=1}^{k-1} \gamma_i$, we get

$$\gamma + \gamma_k - \gamma_k\gamma \leq \sum_{i=1}^{k} \gamma_i.$$

This completes the inductive proof for all m. □

The third Lemma that we will need is the following characterization of the expected "max load" in randomized load balancing.

Lemma D.4 ([110, 119])**.** *Suppose m balls are thrown independently and uniformly at random into n bins. Then, the maximum number of balls in a bin Y satisfies*

$$\mathbb{E}Y \leq \begin{cases} \frac{3\log n}{\log(n/m)} & \text{if } m < n/\log n, \\ c'\log n & \text{if } m < cn\log n, \\ \frac{m}{n} + O(\sqrt{\frac{2m}{n}\log n}) & \text{if } m \gg n\log n. \end{cases}$$

where c' is a constant that depends only on c.

Proof of Theorem 5.4. The proof involves a sharpening of the Lemma D.2 for the BCFW and minibatch setting, where $x \in \mathcal{M} = \mathcal{M}^{(1)} \times ... \times \mathcal{M}^{(n)}$ is a product domain. The proof idea is to exploit this property. Let the current update be on coordinate block index subset S. For each $j \in S$, let the corresponding worker be delayed by $\varkappa_j$ steps, and the corresponding parameter vector be $\tilde{x}$. $\varkappa_j$ is a random variable.

As in the proof of Lemma D.2, we can bound the suboptimality of the approximate subroutine for solving problem j:

$$\begin{aligned}\text{Suboptimality}(\tilde{s}_j) &\leq \langle \tilde{s}_j - s_j^*, \nabla_j f(\tilde{x}) - \nabla_j f(x)\rangle \leq \|\tilde{s}_j - s_j^*\| \|\nabla_j f(\tilde{x}) - \nabla_j f(x)\|_* \\ &\leq D^{(j)}_{\|\cdot\|} L^{(j)}_{\|\cdot\|} \|\tilde{x} - x\| = D^{(j)}_{\|\cdot\|} L^{(j)}_{\|\cdot\|} \left\| \sum_{i=1}^{\varkappa_j} \gamma_{-i}(s_{-i} - x_{-i}) \right\| \qquad \text{(D.4)} \\ &\leq D^{1}_{\|\cdot\|} L^{1}_{\|\cdot\|} \sum_{i=1}^{\varkappa_j} \gamma_{-i} \|(s_{-i} - x_{-i})\| \\ &\leq D^{1}_{\|\cdot\|} L^{1}_{\|\cdot\|} \sum_{i=1}^{\varkappa_j} \gamma_{-i} D^{\tau}_{\|\cdot\|} \leq \varkappa_j \gamma_{-\varkappa_j} D^{1}_{\|\cdot\|} L^{1}_{\|\cdot\|} D^{\tau}_{\|\cdot\|}.\end{aligned}$$

Let $\kappa := \mathbb{E}\varkappa_j$, take expectation on both sides we get

$$\mathbb{E}\ \text{Suboptimality}(\tilde{s}_j) \leq \mathbb{E}(\varkappa_j \gamma_{-\varkappa_j}) D^{1}_{\|\cdot\|} L^{1}_{\|\cdot\|} D^{\tau}_{\|\cdot\|}$$

Repeat the same argument for each $i \in S$, we get

$$\mathbb{E}\ \text{Suboptimality}(\tilde{s}) \leq \mathbb{E}(\varkappa_j \gamma_{-\varkappa_j}) \tau D^{1}_{\|\cdot\|} L^{1}_{\|\cdot\|} D^{\tau}_{\|\cdot\|}.$$

To put it into the desired format in (5.5), we solve the following inequality for δ

$$\frac{\gamma \delta C_f^{\tau}}{2} \geq \mathbb{E}(\varkappa_j \gamma_{-\varkappa_j}) \tau D^{\tau}_{\|\cdot\|} D^{1}_{\|\cdot\|} L^{1}_{\|\cdot\|}$$

we get

$$\delta \geq \frac{2\tau}{C_f^{\tau}} \mathbb{E}\left(\frac{\varkappa_j \gamma_{-\varkappa_j}}{\gamma}\right) D^{\tau}_{\|\cdot\|} D^{1}_{\|\cdot\|} L^{1}_{\|\cdot\|}.$$

By the specification of the stepsizes, we can calculate for each k,

$$\frac{\gamma_{-\varkappa_j}}{\gamma} = \frac{\tau^2 k + 2n}{\tau^2(\max(k - \varkappa_j, 0)) + 2n}.$$

Note that we always enforce $\varkappa_j$ to be smaller than $\frac{k}{2}$ (otherwise the update is dropped), we can therefore upper bound $\mathbb{E}(\frac{\varkappa_j \gamma_{-\varkappa_j}}{\gamma})$ by 2κ. This gives us the the first bound (5.9) on δ in Theorem 5.4.

To get the second bound on δ, we start from (D.4) and bound $\|\tilde{x} - x\|$ differently. Let S be the set of $\tau\varkappa_j$ coordinate blocks that were updated in the past $\varkappa_j$ iterations. In the cases where fewer than $\tau\varkappa_j$ blocks were updated, just arbitrarily pick among the coordinate blocks that were updated 0 times so that $|S| = \tau \varkappa_j$. $\tilde{x} - x$ is supported only on S. Suppose coordinate block $i \in S$ is updated by m times, as below

$$\tilde{x}_{(i)} = \sum_{j=1}^{m} \gamma_j (s_j - [x_j]_{(i)})$$

for some sequence of $0 \le \gamma_1, ..., \gamma_m \le 1$ and $s_1, ..., s_m \in \mathcal{M}_i$ and recursively $[x_j]_{(i)} = [x_{j-1}]_{(i)} + \gamma_j (s_j - [x_{j-1}]_{(i)})$ ($x_0 = x$). Apply Lemma D.3 for each coordinate block, we know that there exist $s_{(i)} \in \mathcal{M}_i$ in each block $i \in S$ such that

$$\tilde{x}_{(i)} = x_{(i)} + \gamma_{(i)} (s_{(i)} - x_{(i)})$$

with

$$\gamma_{(i)} \le \sum_{j \in \text{ iterations where } i \text{ is updated}} \gamma_j \le m \gamma_{\max}. \tag{D.5}$$

Note that $s_{(i)} \in \mathcal{M}_i$ for each $i \in S$ implies that their concatenation $s_{(S)} \in \mathcal{M}_S$. Also $\gamma_{\max} \le \gamma_{-\varkappa_j}$. Therefore

$$\|\tilde{x} - x\| = \left\| \sum_{i \in S} \gamma_{(i)} (s_{(i)} - x_{(i)}) \right\| \le m \gamma_{\max} \|s_{(S)} - x_{(S)}\| \le Y \gamma_{-\varkappa_j} D_{\|\cdot\|}^{\tau \varkappa_j}$$

where Y is a random variable that denotes the number of updates received by the most updated coordinate block (the maximum load). Apply a previously used argument to get $\gamma_{-\varkappa_j} < 2\gamma$, take expectation on both sides, to get the following by the law of total expectations and (D.5)

$$\begin{aligned} \mathbb{E}\|\tilde{x} - x\| &\le \mathbb{E}\left(Y \gamma_{-\varkappa_j} D_{\|\cdot\|}^{\tau \varkappa_j} \right) = \mathbb{E}\left[\mathbb{E}\left(Y \gamma_{-\varkappa_j} D_{\|\cdot\|}^{\tau \varkappa_j} \middle| \varkappa_j \right) \right] = \mathbb{E}\left[\gamma_{-\varkappa_j} D_{\|\cdot\|}^{\tau \varkappa_j} \mathbb{E}\left(Y | \varkappa_j \right) \right] \\ &\le 2\gamma \mathbb{E}\left[D_{\|\cdot\|}^{\tau \varkappa_j} \mathbb{E}\left(Y | \varkappa_j \right) \right] \end{aligned} \tag{D.6}$$

This expectation is taken over the entire history of minibatch choice and delay associated with each update. When we condition on $\varkappa_j$, the conditional expectation of Y becomes the load-balancing problem.

By Lemma D.4 when $\kappa_{\max} \tau \le \frac{n}{\log n}$, it follows from (D.6) that

$$\mathbb{E}\|\tilde{x} - x\| \le 2\gamma \mathbb{E} D_{\|\cdot\|}^{\varkappa_j \tau} \frac{3 \log n}{\log(n / \varkappa_j \tau)} \le \frac{3 \log n}{\log[n / (\tau \kappa_{\max})]} 2\gamma \mathbb{E} D_{\|\cdot\|}^{\varkappa_j \tau}.$$

When $\kappa_{\max} \tau < c n \log n$,

$$\mathbb{E}\|\tilde{x} - x\| \le 2\gamma \mathbb{E} D_{\|\cdot\|}^{\varkappa_j \tau} O(\log n) \le O(\log n) 2\gamma \mathbb{E} D_{\|\cdot\|}^{\varkappa_j \tau}.$$

When $\kappa_{\max} \tau \gg n \log n$, then

$$\mathbb{E}\|\tilde{x} - x\| \le (1 + o(1)) \frac{\tau \kappa_{\max}}{n} 2\gamma \mathbb{E} D_{\|\cdot\|}^{\varkappa_j \tau}.$$

Repeating the above results for each block $j \in S$, and summing them up leads to an upper bound for $\frac{\gamma \delta C_f^\tau}{2}$ and the proof of (5.9) is complete by solving for δ. $\square$

D.2 Proofs of other technical results

Relationship of the curvatures.

Proof of Lemma 5.1. $C_f^{(S)} \leq C_f$ follows from the fact that

$$\langle y_{(S)} - x_{(S)}, \nabla_{(S)} f(x) \rangle = \langle y_{[S]} - x_{[S]}, \nabla f(x) \rangle,$$

and $s_{[S]} \in \mathcal{M}$. In other words, the $\arg\sup$ of (5.3) is a feasible solution in the $\sup$ to compute the global C_f. Similar argument holds for the proof $C_f^{(i)} \leq C_f^{(S)}$ as $i \in S$.

In the second part,

$$C_f^\tau = \frac{1}{\binom{n}{\tau}} \sum_{T \subset [n], |T| = \tau} C_f^{(T)}.$$

We can use $C_f^{(S)} \geq C_f(j)$ from the first inequality of the lemma, to get the inequality below.

$$C_f^\tau = \frac{1}{\binom{n}{\tau}} \sum_{j \in [n]} \sum_{T \in P_j} C_f^{(T)} \geq \frac{1}{\binom{n}{\tau}} \sum_{j \in [n]} \sum_{T \in P_j} C_f^{(j)} = \frac{1}{\binom{n}{\tau}} \sum_{j \in [n]} \binom{n}{\tau} \frac{\tau}{n} C_f^{(j)} = \frac{\tau}{n} C_f^{\otimes} \geq \frac{1}{n} C_f^{\otimes}$$

The relaxation of C_f^τ to C_f is trivial since $C_f^{(T)} \leq C_f$ holds for any $T \subseteq [n]$ from the first part of the lemma. $\square$

Bounding C_f^τ using expected boundedness and expected incoherence

Proof of Theorem 5.3. By Definition of H, for any $x, z \in \mathcal{M}$, $\gamma \in [0, 1]$

$$f(x + \gamma(z - x)) \leq f(x) + \gamma(z - x)^T \nabla f(x) + \frac{\gamma^2}{2}(z - x)^T H (z - x).$$

Rearranging the terms we get

$$\frac{2}{\gamma^2} \left[f(x + \gamma(z - x)) - f(x) - \gamma(z - x)^T \nabla f(x) \right] \leq (z - x)^T H (z - x)$$

The definition of set curvature (5.3) is written in an equivalent notation with $z = x_{[S^c]} + s_{[S]}$ and $y = x + \gamma(z - x) = x + \gamma(s_{[S]} - x_{[S]})$. So we know the support of $z - x$ is constrained to be within the coordinate blocks S.

Plugging this into the definition of (5.3) we get an analog of Equation (2.12) in [68] for $C_f^{(S)}$.

$$\begin{aligned}
C_f^{(S)} &= \sup_{\substack{x,z\in\mathcal{M},\gamma\in[0,1]\\ z_{(S^c)}-x_{(S^c)}=0}} \frac{2}{\gamma^2}\left[f(x+\gamma(z-x))-f(x)-\gamma(z-x)^T\nabla f(x)\right]\\
&\leq \sup_{\substack{x,z\in\mathcal{M},\\ z_{(S^c)}-x_{(S^c)}=0}} (z-x)^T H(z-x) = \sup_{\substack{x,z\in\mathcal{M},\\ z_{(S^c)}-x_{(S^c)}=0}} s_{(S)}^T H s_{(S)}\\
&\leq \sup_{w\in\mathcal{M}^{(S)}} (2w^T)H_S(2w) = 4\left\{\sup_{w_i\in\mathcal{M}^{(i)}\forall i\in S}\sum_{i\in S} w_i^T H_{ii} w_i + \sum_{i,j\in S,i\neq j} w_i^T H_{ii} w_j\right\}\\
&\leq 4\left\{\sum_{i\in S}\sup_{w_i} w_i^T H_{ii} w_i + \sum_{i,j\in S,i\neq j}\sup_{w_i,w_j} w_i^T H_{ii} w_j\right\}\\
&\leq 4(\sum_{i\in S} B_i + \sum_{i,j\in S,i\neq j}\mu_{ij}).
\end{aligned}$$

Take expectation for all possible S of size τ and we obtain the lemma statement. □

Proof of the example with sublinear dependence of κ

Proof of Lemma 5.2. We first show that a continuous extension of $D_{\|\cdot\|}^{\theta}$ is concave in θ

$$\begin{aligned}
D_{\|\cdot\|}^{\theta} &= \max_{S\subset[n]||S|=\theta}\ \sup_{x,y\in\mathcal{M}^{(S)}} \|x-y\|\\
&= \max_{S\subset[n]||S|=\theta}\ \sup_{x,y\in\mathcal{M}^{(S)}} \sqrt{\sum_{i\in S}\|x_{(i)}-y_{(i)}\|^2}\\
&= \sqrt{\max_{S\subset[n]||S|=\theta}\sum_{i\in S}\ \sup_{x_{(i)},y_{(i)}\in\mathcal{M}^{(i)}}\|x_{(i)}-y_{(i)}\|^2}
\end{aligned}$$

The supremum is obtained by sorting and the function in the square root is concave function of θ, when we extend the support of this function to $\mathbb{R}_+$ through linear interpolation. By the composition theorem, the square root of that is also a concave function in τ. We call this function $\tilde{D}_{\|\cdot\|}^{\theta}$. Note that $\tilde{D}_{\|\cdot\|}^{\theta} = D_{\|\cdot\|}^{\theta}$ when $\theta\in[n]$ such that if we take expectation over the any discrete distribution over θ, their expectations are the same. It follows from Jensen's inequality that

$$\begin{aligned}
\mathbb{E}D_{\|\cdot\|}^{\varkappa\tau} &= \mathbb{E}\tilde{D}_{\|\cdot\|}^{\varkappa\tau} \leq \tilde{D}_{\|\cdot\|}^{\mathbb{E}\varkappa\tau}\\
&\leq D_{\|\cdot\|}^{\lceil\mathbb{E}\varkappa\tau\rceil}\\
&= \sqrt{\max_{S\subset[n]\big||S|=\lceil\mathbb{E}\varkappa\tau\rceil}\sum_{i\in S}\ \sup_{x_{(i)},y_{(i)}\in\mathcal{M}^{(i)}}\|x_{(i)}-y_{(i)}\|^2}\\
&\leq \sqrt{\lceil\mathbb{E}\varkappa\rceil}D_{\|\cdot\|}^{\tau}.
\end{aligned}$$

□

Proof of specific examples

Proof of Example D.1. First of all, $H = \lambda A^T A$. Since all columns of A have the same magnitude $\sqrt{2}/n$. By the Holder's inequality and the 1-norm constraint in every block, we know $B_i = \frac{2}{n^2\lambda}$ for any i therefore $B = \frac{2}{n^2\lambda}$. Secondly, by well-known upper bound for the area of the spherical cap, which says for any fixed vector z and random vector a on a unit sphere in $\mathbb{R}^d$,

$$\mathbb{P}(|\langle z, a\rangle| > \epsilon\|z\|) \leq 2e^{\frac{-d\epsilon^2}{2}},$$

we get

$$\mathbb{P}(\mu_{ij} > 2\sqrt{\frac{20\log d}{d}}) \leq \frac{2}{d^{10}}.$$

Take union bound over all pairs of labels we get the probability as claimed. $\square$

Proof of Example D.2. The matrix $D^T D$ is tridiagonal with 2 on the diagonal and -1 on the off-diagonal. If we vectorize U by concatenating $u = [u_1; ...; u_{n-1}]$, the Hessian matrix for u will be $H = \Pi I_d \otimes (D^T D)\Pi^T$ where Π is some permutation matrix. Without calculating it explicitly, we can express

$$\begin{aligned}&u_S^T H_S u_S = u_S^T(D^T \otimes 1_d)(D^T \otimes 1_d)^T u_S\\ =&\sum_{i\in S} u_i^T \begin{bmatrix} D^T_{:,i}\\ D^T_{:,i}\\ \vdots\\ D^T_{:,i}\end{bmatrix} \begin{bmatrix} D_{:,i} & D_{:,i} & \dots & D_{:,i}\end{bmatrix} u_i + \sum_{i,j\in S, i\neq j} u_i^T \begin{bmatrix} D^T_{:,i}\\ D^T_{:,i}\\ \vdots\\ D^T_{:,i}\end{bmatrix} \begin{bmatrix} D_{:,j} & D_{:,j} & \dots & D_{:,j}\end{bmatrix} u_j.\end{aligned}$$

We note that for any $|i-j| \geq 2$, the second term is 0. Apply the constraint that $\|u_i\|_2 \leq \lambda$ and the fact that the ℓ_2 operator norm of $\begin{bmatrix} D_{:,j} & D_{:,j} & \dots & D_{:,j}\end{bmatrix}$ is $\sqrt{2d}$, we get $B_i = 2\lambda^2 d$. Similarly, $2(n-2)$ nonzero obeys $\mu_{ij} = \lambda^2 d$. This allows us to obtain an upper bound

$$C_f^\tau \leq 4\left[2\tau\lambda^2 d + \frac{2(n-2)\tau(\tau-1)}{(n-2)(n-1)}\lambda^2 d\right] \leq 16\tau\lambda^2 d.$$

which scales with τ. $\square$

D.2.1 Pseudocode for the Multicore Shared Memory Architecture

We present pseudocode for the multicore shared memory setting here. It is the same except that each worker becomes a thread, the network buffer of servers become the a data structure, the workers' network buffer becomes the shared parameter vector and the workers can write to the data structure or the shared parameter vector directly.

Algorithm 7 AP-BCFW: Asynchronous Parallel Block-Coordinate Frank-Wolfe (Shared memory)

```
———————————SERVER THREAD——————————
Input: An initial feasible $x^{(0)}$, mini-batch size $\tau$, number of workers $T$.
0.  Write $x^{(0)}$ to shared memory. Declare a container (a queue or a stack).
for $k$ = 1,2,... ($k$ is the iteration number.) do
    1.  Keep popping the container until we have $\tau$ updates on $\tau$ disjoint blocks. Denote the index set by $S$.
    2.  Set step size $\gamma = \frac{2n\tau}{\tau^2 k+2n}$.
    3.  Write sparse updates $x^{(k)} = x^{(k-1)} + \gamma \sum_{i \in S}(s_{[i]} - x^{(k-1)}_{[i]})$ into the shared memory.
    if converged then
        Broadcast STOP signal to all threads and break.
    end if
end for
Output: $x^{(k)}$.
———————————WORKER THREADS——————————
while no STOP signal received do
    a. Randomly choose $i \in [n]$.
    b. Calculate partial gradient $\nabla_{(i)} f(x)$ using $x$ in the shared memory and solve (5.2).
    c. Push $\{i, s_{(i)}\}$ to the container.
end while
```

The above pseudo code can be further simplified when $\tau = 1$. In particular, we do not need a server any more. Each worker can simply write to the shared memory bus. The probability of two workers writing to the same block is small as we analyzed in Section D.4.2. The algorithm essentially lock-free as in [114] modulo requiring the updates of each coordinate block to be atomic. [114] is stronger in that it allows each scalar addition to be atomic.

There is an additional restriction due to the fixed predefined sequence of step sizes, which in fact requires a centralized shared counter that is atomic, so that no two threads have simultaneously the same k. In practice, we can simply choose a fixed sequence of stepsize for each worker separately.

Algorithm 8 AP-BCFW: Asynchronous Parallel Block-Coordinate Frank-Wolfe (Lock-Free Shared-Memory)

```
Input: An initial feasible $x^{(0)}$, number of workers $T$, a centralized counter.
0. Write $x^{(0)}$ to shared memory.
————INDEPENDENTLY ON EACH THREAD———
while not converged do
  a. Randomly choose $i \in [n]$.
  b. Calculate partial gradient $\nabla_{(i)} f(x)$ using $x$ in the shared memory and solve (5.2).
  c. Read centralized counter for $k$. Set step size $\gamma = \frac{2n}{k+2n}$.
  d. Add $\gamma(s_{(i)} - x_{(i)})$ to block $i$ of the shared memory.
  e. Increment the counter $k = k + 1$.
end while
————————————————————————————-
if converged then
  Output: $x^{(k)}$. and break.
end if
```

D.3 Application to Structural SVM

We briefly review structural SVMs and show how to solve the associated convex optimization problem using our AP-BCFW method.

In structured prediction setting, the task is to predict a structured output $\mathbf{y} \in \mathcal{Y}$, given $\mathbf{x} \in \mathcal{X}$. For example, $\mathbf{x}$ could be the pixels in the picture of a word, $\mathbf{y}$ could be the sequence of characters in the word. A feature map $\phi : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^d$ encodes compatibility between inputs and outputs. A linear classifier parameter $\mathbf{w}$ is learned from data so that $\operatorname{argmax}_{\mathbf{y}\in\mathcal{Y}} \langle \mathbf{w}, \phi(\mathbf{x}, \mathbf{y}) \rangle$ gives the output for an input $\mathbf{x}$. Suppose we have the training data $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^n$ to learn $\mathbf{w}$. Define $\psi_i(\mathbf{y}) := \phi(\mathbf{x}_i, \mathbf{y}_i) - \phi(\mathbf{x}_i, y)$ and let $L_i(\mathbf{y}) := L(\mathbf{y}_i, \mathbf{y})$ denote the loss incurred by predicting $\mathbf{y}$ instead of the correct output $\mathbf{y}_i$. The classifier parameter $\mathbf{w}$ is learned by solving the optimization problem

$$\begin{aligned} &\min_{\mathbf{w},\xi} \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} \xi_i \\ &\text{s.t } \langle \mathbf{w}, \psi_i(\mathbf{y}) \rangle \geq L(\mathbf{y}_i, \mathbf{y}) - \xi_i \quad \forall i, \mathbf{y} \in \mathcal{Y}(\mathbf{x}_i). \end{aligned} \tag{D.7}$$

We solve the dual of this problem using our method. We introduce some more notation to formulate the dual. Denote $\mathcal{Y}_i := \mathcal{Y}(\mathbf{x}_i)$, the set of possible labels for $\mathbf{x}_i$. Note that $|\mathcal{Y}_i|$ is exponential in the length of label $\mathbf{y}_i$. Let $m = \sum_{i=1} |\mathcal{Y}_i|$. Let $A \in \mathbb{R}^{d\times m}$ denote a matrix whose m columns are given by $\{\frac{1}{\lambda n}\psi_i(\mathbf{y}) \mid i \in [n], \mathbf{y} \in \mathcal{Y}_i\}$. Let $b \in \mathbb{R}^m$ be a vector given by the entries

$\{\frac{1}{n}L_i(\mathbf{y}) \mid i \in [n], \mathbf{y} \in \mathcal{Y}_i\}$. The dual of (D.7) is given by

$$\begin{aligned} \min_{\alpha \in \mathbb{R}^m} f(\alpha) &:= \frac{\lambda}{2}\|A\alpha\|^2 - b^T\alpha \\ \text{s.t} \sum_{\mathbf{y} \in \mathcal{Y}_i} \alpha_i(\mathbf{y}) &= 1 \quad \forall i \in [n], \alpha \geq 0 \end{aligned} \tag{D.8}$$

The primal solution $\mathbf{w}$ can be retrieved from the dual solution α from the relation $\mathbf{w} = A\alpha$ obtained from KKT conditions. Also note that the domain $\mathcal{M}$ of (D.8) is exactly the product of simplices $\mathcal{M} = \Delta_{|\mathcal{Y}_1|} \times \cdots \times \Delta_{|\mathcal{Y}_n|}$.

The subproblem in equation (5.2) takes a well-known form in the Frank-Wolfe setup for solving (D.8). The gradient is given by

$$\nabla f(\alpha) = \lambda A^T A\alpha - b = \lambda A^T \mathbf{w} - b$$

whose $(i, \mathbf{y})$-th component is given by $\frac{1}{n}\left(\langle \mathbf{w}, \psi_i(\mathbf{y})\rangle - L_i(\mathbf{y})\right)$. Define $H_i(\mathbf{y}; \mathbf{w}) := L_i(\mathbf{y}) - \langle \mathbf{w}, \psi_i(\mathbf{y})\rangle$ so that the $(i, \mathbf{y})$-th component of the gradient is $-\frac{1}{n}H_i(\mathbf{y}; \mathbf{w})$. In the subproblem (5.2), the domain $\mathcal{M}^{(i)}$ is the simplex $\Delta_{\mathcal{Y}_i}$ and the block gradient $\nabla_{(i)} f(\alpha)$ is linear. So, the objective is minimized at a corner of the simplex $\mathcal{M}^{(i)}$ and the optimum value is simply given by $\min_{\mathbf{y}} \nabla_{(i)} f(\alpha)$ which can be rewritten as $\max_{\mathbf{y}} H_i(\mathbf{y}; \mathbf{w})$. Further, the corner can be explicitly written as the indicator vector $e^{\mathbf{y}_i^*} \in \mathcal{M}^{(i)}$ where $\mathbf{y}_i^* = \operatorname{argmax}_{\mathbf{y}} H_i(\mathbf{y}; \mathbf{w})$. It turns out that this maximization problem can be solved efficiently for several problems. For example, when the output is a sequence of labels, a dynamic programming algorithm like Viterbi can be used.

As mentioned before, m is too large to update the dual variable α directly. So, we make an update to the primal variable $\mathbf{w} = A\alpha$ instead. The Block-Coordinate Frank-Wolfe update for the i-th block maybe written as $\alpha_{(i)}^{k+1} = \alpha_i^k + \gamma(s_i - \alpha_{(i)}^k)$ where γ is the step-size. Recalling that the optimal s_i is $e^{\mathbf{y}_i^*}$, by multiplying the previous equation by A_i, we arrive at $\mathbf{w}_i^{(k+1)} = \mathbf{w}_i^k + \gamma(A_{i,y_i^*} - \mathbf{w}_i^{(k)})$ where $\mathbf{w}_i^{(k)} := A_i\alpha_{(i)}$. From this definition of $\mathbf{w}_i^{(k)}$, the primal update is obtained by noting that $\mathbf{w}^{(k)} = \sum_i \mathbf{w}_i^{(k)}$. Explicitly, the primal update is given by $\mathbf{w}^{(k+1)} = \mathbf{w}^k + \gamma(A_{i,y_i^*} - \mathbf{w}_i^{(k)})$. Note that $A_{i,y_i^*} = \frac{1}{\lambda n}\psi_(\mathbf{y}_i^*)$. This Block-Coordinate version can be easily extended to AP-BCFW. In our shared memory implementation, for OCR dataset, we do the line search computation and $\mathbf{w}_i^{(k)}$ update step on the workers instead of the server because these computations turn out to be expensive enough to make the server the bottleneck even for modest number of workers.

D.4 Other technical results and discussions

D.4.1 Oracle assumption and heterogeneous blocks

Recall that our results rely on the oracle assumption that $\mathcal{O}$ provides updates that are iid uniform over $[n]$ (Assumption A1). We discuss the implications and limitations of this assumption and then propose possible solutions.

Consider the setting where $\mathcal{O}$ consists of T possibly heterogeneous workers and each worker samples iid from $[n]$. As we discussed before, A1 holds under the additional condition that the time needed to complete one subroutine solve for Block i by Worker j does not depend on i.

Consider the simple example due to an anonymous reviewer: Let $\tau = 1$, $T = 2$ and there are a total of two blocks. Block 1 takes only a millisecond an Block 2 takes a year to solve for both workers. In this case, the first update received by the server is with probability $3/4$ for Block 1 and only $1/4$ for Block 2.

This could potentially limit the use of our parallel algorithm for applications such as structured predictions where sentences having different lengths, or cases where there are different sparsity level over data points/constraints depending on how we formulate the problem.

This is in fact not a problem unique to us, Assumption A1 is implicitly required in most existing analysis for asynchronous stochastic algorithms [e.g., 98, 114]. As a result, they all share the same woe. One could argue that parallelization is the wrong problem to address when block subroutines significantly differ with each other. Efforts should be spent on perhaps solving the expensive subproblem in parallel. But still, even mild heterogeneity over blocks invalidates our convergence result.

Henceforth, we propose two simple ways to address this issue and discuss their pros and cons.

Padding: A naive solution is to per-calculate the time-complexity with respect to each block and inject artificial time padding on each user such that all blocks have the same time complexity.

Pre-select S: An alternative is to let the server randomly choose a coordinate subset S of size τ, and the workers can only work on S, either by independently sample from S or work on whichever that is not available.

Neither of the two solutions is completely satisfactory. The padding approach ensures all results in the paper to hold including those for the delayed oracles, but inevitably, the time to complete each block now depends on the most expensive block. The second approach has milder dependence on the worst block, in fact it depends only on the time for the fastest worker to solve the slowest problem in each chosen S. However, it requires sending an updated parameter to all workers in every iteration. It could still be robust to heterogeneous workers when τ is several times larger than T, and when workers work asynchronously within the mini-batch, we prove in Proposition D.1 that the number of collisions is small.

Fully asynchronous parallelism over blocks with heterogeneous blocks without dependence on the slowest block remains an important open problem.

D.4.2 Controlling collisions in distributed setting

In the distributed setting, different workers might end up working on the same slot.

In Algorithm 6, different workers may end up working on the same coordinate block and the server will drop a number of updates in case of collision. The following proposition shows that for this potential redundancy is not excessive is small and for a large range of τ, we also show additional strong concentration to its mean.

Proposition D.1. *In the distributed asynchronous update scheme above:*

i) The expected number of subroutine calls from all workers to complete each iteration is $\tau + \sum_{i=1}^{\tau-1} \frac{i}{n-i}$*.*

ii) If $0.02n < \tau < 0.6n$*, with probability at least* $1 - \exp(-n/60)$*, no more than* 2τ *random draws (*2τ *subroutine calls in total from all workers) suffice to complete each iteration.*

Proof. The first claim is the well-known coupon collector problem.

The second claim requires an upper bound of the expectation. In expectation, we need $\frac{n}{n-k}$ balls to increase the unique count from k to $k+1$. So in expectation we need

$$1 + \frac{n}{n-1} + \frac{n}{n-2} + ... + \frac{n}{n-\tau+1} = \tau + \sum_{i=1}^{\tau-1} \frac{i}{n-i}$$

$$\leq \tau + \frac{1+2+\cdots+(\tau-1)}{n-\tau+1} = \tau + \frac{\tau(\tau-1)}{2(n-\tau+1)} < \tau\left[1 + \frac{1}{2(n/\tau-1)}\right].$$

To see the second claim, first defined f_t to be the number of non-empty bins after t random ball throws, which can be consider as a function of the t iid ball throws $X_1, X_2, ..., X_t$. It is clear that if we change only one of the X_i, f_t can be changed by at most 1. Also, note that the probability that any one bin being filled is $1-(1-\frac{1}{n})^t$, so $\mathbb{E}f_t = n\left[1-\left(1-\frac{1}{n}\right)^t\right]$.

By the McDiarmid's inequality, $\mathbb{P}\left[f_t < \mathbb{E}f_t - \epsilon\right] \leq \exp\left[-\frac{2\epsilon^2}{t}\right]$. Take $t = 2\tau$, and $\epsilon = \mathbb{E}f_{2\tau} - \tau$, then

$$\mathbb{P}\left[f_{2\tau} < \tau\right] \leq \exp\left[-\frac{1}{\tau}\left(n\left[1-\left(1-\frac{1}{n}\right)^{2\tau}\right]-\tau\right)^2\right] \leq \exp\left[-\frac{1}{\tau}\left(n\left[1-e^{-\frac{2\tau}{n}}\right]-\tau\right)^2\right]$$

$$= \exp\left[-n\cdot\frac{n}{\tau}\left(1-e^{-\frac{2\tau}{n}}-\frac{\tau}{n}\right)^2\right] \leq \exp\left[-Cn\right],$$

where C is some constant which is the smaller of the two evaluations of the function $\frac{n}{\tau}\left(1-e^{-\frac{2\tau}{n}}-\frac{\tau}{n}\right)^2$ at $\tau = 0.02n$ and $\tau = 0.6n$ (where the function is concave between the two). As a matter of fact, C can be taken as $\frac{1}{60}$.

Let g_τ be the number of balls that one throws that fills τ bins, the result is proven by noting that

$$\mathbb{P}(g_\tau \leq 2\tau) = \mathbb{P}(f_{2\tau} \geq \tau) \geq 1 - \exp\left[-Cn\right].$$

□

D.4.3 Curvature and Lipschitz Constant

In this section, we illustrate the relationship between the coordinate curvature constant, coordinate gradient Lipschitz conditions, and work out the typical size of the constants in Theorem 5.4. For the sake of discussion, we will focus on the quadratic function $f(x) = \frac{x^T Ax}{2} + bTx$. We start by showing that for quadratic function. The constant that one can get via choosing a specific norm can actually match the curvature constant. To be completely explicit, we define gradient Lipschitz constant $L_{\|\cdot\|}$ with respect to a norm $\|\cdot\|$, this requires that for any x, y,

$$\|\nabla f(y) - \nabla f(x)\|_* \leq L\|x - y\|.$$

where $\|\cdot\|_*$ is the dual norm.

Proposition D.2. *For quadratic functions with Hessian $A \succeq 0$, there exists a norm $\|\cdot\|$ such that the curvature constant $C_f = [D_{\|\cdot\|}]^2 L_{\|\cdot\|}$.*

Proof. We will show that this norm is simply the A-norm, $\|\cdot\|_A = \sqrt{(\cdot)^T A(\cdot)}$. The upper bound $C_f \leq [D_{\|\cdot\|_A}]^2 L_{\|\cdot\|_A}$ is a direct application of the result in [69, Appendix D]. To show a lower bound it suffices to construct $s, x \in \mathcal{M}, \gamma \in [0, 1]$ and $y = \gamma s + (1 - \gamma)x$ such that

$$\frac{2}{\gamma^2}(f(y) - f(x) - \langle y - x, \nabla f(x)\rangle) = [D_{\|\cdot\|_A}]^2 L_{\|\cdot\|_A}.$$

For quadratic functions,

$$\frac{2}{\gamma^2}[f(y) - f(x) - \langle y - x, \nabla f(x)\rangle] = \frac{1}{2}(y - x)^T A(y - x) = \frac{1}{\gamma^2}\|y - x\|_A^2$$

Take $\gamma = 1$ and y, x on the boundary of $\mathcal{M}$ such that $\|y - x\|_A = D_{\|\cdot\|_A}$, as a result, we get $C_f \geq D_{\|\cdot\|_A}]^2$. It remains to show that the gradient Lipschitz constant with respect to $\mathcal{A}$-norm is 1, which directly follows from the Taylor expansion. □

Similar arguments work for $C_f^{(i)}$ and $C_f^{(S)}$ under the same norm. Clearly, this means that the corresponding restriction of the subset domain has $A_{i,i}$-norm or $A_{(S)}$-norm.

We now consider the approximation constants due to the delays in Theorem 5.4, and work out more explicit bounds for quadratic functions and carefully chosen norm. Recall that the simple bound (5.8) has constant δ in the order of

$$\frac{\kappa\tau L^1_{\|\cdot\|} D^1_{\|\cdot\|} D^\tau_{\|\cdot\|}}{C_f^\tau}.$$

Suppose we use the A-norm, then $L^1_{\|\cdot\|} = L^\tau_{\|\cdot\|} = 1$, and $C_f^\tau = [D^\tau_{\|\cdot\|}]^2$, the bound can be reduced to

$$\delta = O(\frac{\tau D^1_{\|\cdot\|}}{D^\tau_{\|\cdot\|}}) = O(\kappa\sqrt{\tau}).$$

where the last step requires $\mathcal{M}_i$ to be all equivalent and A to be block-diagonal with identical $A_{(i)}$.

Similarly the strong bound (5.9) has constant δ in the order of

$$\delta = \tilde{O}\left(\frac{\tau L^1_{\|\cdot\|} D^1_{\|\cdot\|} D^{\kappa\tau}_{\|\cdot\|}}{C^\tau_f}\right) = \tilde{O}\left(\frac{\tau D^1_{\|\cdot\|} D^{\kappa\tau}_{\|\cdot\|}}{[D^\tau_{\|\cdot\|}]^2}\right) = \tilde{O}(\sqrt{\kappa\tau})$$

Again, the last step requires a strong assumption that $\mathcal{M}_i$ to be all equivalent and A to be block-diagonal with identical diagonal blocks. While these calculations only apply to specific case of a quadratic function with a lot of symmetry, we conjecture that in general the flexibility of choosing the norm will allow the ratio of these boundedness constants and C^τ_f to be a well-controlled constant and the typical dependence on the system parameter τ and κ should stay within the same ball park.

D.4.4 Examples and illustrations

In this section, we now derive specific instances of the Theorem 5.3 for the structural SVM and Group Fused Lasso. For the structural SVM, a simple generalization of [82, Lemmas A.1, A.2] shows that in the worst case, using $\tau > 1$ offers no gain at all. Fortunately, if we consider more specific problems, using larger τ does yield faster convergence. We provide two such examples below.

Example D.1 (Structural SVM for multi-label classification (with random data)). *We describe the application to structural SVMs in detail in Section D.3 (please see this section for details on notation). Here, we describe the convergence rate for this application. According to [149], the compatibility function* $\phi(x, y)$ *for multiclass classification will be* $[0, ..., 0, x^T, 0, ...0]^T/\lambda n$ *where the only nonzero block that we fill with the feature vector is the* (y)*th block. So* $\psi_i(x_i, j) = \phi(x_i, y_i) - \phi(x_i, j)$ *looks like* $[0, ..., 0, x_i^T, 0\ , ...0,\ -x_i^T, 0,\ ...0]^T/\lambda n$*. This already ensures that* $B = \frac{2}{n^2\lambda}$ *provided* x_i *lie on a unit sphere. Suppose we have* K *classes and each class has a unique feature vector drawn randomly from a unit sphere in* $\mathbb{R}^d$*; furthermore, for simplicity assume we always draw* $\tau < K$ *data points with* τ *distinct labels*[1] $\mu \leq \sqrt{\frac{c \log d}{d}\frac{2}{n^2\lambda}}$*, for some constant c. In addition, if* $d \geq \tau^2\sqrt{c \log d}$*, then with high probability*

$$C^\tau_f \leq \frac{8\tau + 8\tau^2\sqrt{\frac{c\log d}{d}}}{n^2\lambda} = O\left(\frac{c\tau}{n^2\lambda}\right),$$

which yields a convergence rate $O(\frac{R^2}{\lambda\tau k})$*, where* $R := \max_{i\in[n], y\in\mathcal{Y}_i} \|\psi_i(y)\|_2$ *using notation from Lemmas A.1 and A.2 of [82].*

This analysis suggests that a good rule-of-thumb is that we should choose τ *to be at most the number of categories for the classification. If each class is a mixture of random draws from the unit sphere, then we can choose* τ *to be the underlying number of mixture components.*

---

[1] This is an oversimplification but it offers a rough rule-of-thumb. In practice, C^τ_f should be in the same ballpark as our estimate here.

Example D.2 (Group Fused Lasso). *The Group Fused Lasso aims to solve (typically for $q = 2$)*

$$\min_X \quad \tfrac{1}{2}\|X - Y\|_F^2 + \lambda\|XD\|_{1,q}, \qquad q > 1, \tag{D.9}$$

where $X, Y \in \mathbb{R}^{d\times n}$, and column y_t of Y is an observed noisy d-dimensional feature vector at time $1 \le t \le n$. The matrix $D \in \mathbb{R}^{n\times(n-1)}$ is the differencing matrix that takes the difference of feature vectors at adjacent time points (columns). The formulation aims to filter the trend that has some piecewise constant structures. The dual to (D.9) *is*

$$\begin{aligned} &\max_U \; -\frac{1}{2}\|UD^T\|_F^2 + \mathrm{tr} UD^TY^T \\ &\textit{s.t.} \quad \|U_{:,t}\|_p \le \lambda,\ \forall t = 1, ..., n-1, \end{aligned}$$

where p is conjugate to q, i.e., $1/p + 1/q = 1$. This block-constrained problem fits our structure (5.1)*. For this problem, we find that $B \le 2\lambda^2 d$ and $\mu \le 2\lambda^2 d/(n-1)$, which yields*

$$C_f^\tau \le 16\tau\lambda^2 d.$$

Consequently, the rate of convergence becomes $O(\frac{n^2\lambda^2 d}{\tau k})$. In this case, batch FW will have a better rate of convergence than BCFW.[2]

Example D.3 (Structural SVM worst-case bound)**.** *For structural SVM with arbitrary data (including even pathological/trivial data), using notation from Lemmas A.1 and A.2 of [82], define $R := \max_{i\in[n], y\in\mathcal{Y}_i} \|\psi_i(y)\|_2$. Then we can provide an upper bound*

$$B, \mu \le \frac{R^2}{\lambda n^2} \quad \Longrightarrow \quad C_f^\tau \le \frac{4\tau^2 R^2}{\lambda n^2}. \tag{D.10}$$

In this case, for any $\tau = 1, ..., n$, the rate of convergence will be the same $O(\frac{R^2}{\lambda k})$.

An illustration for the group fused lasso Figure D.1 shows a typically application for group fused lasso (filtering piecewise constant multivariate signals whose change poitns are grouped together).

D.4.5 Comparison to parallel block coordinate descent

With some understanding on C_f^τ, we can now explicitly compare the rate of convergence in Theorem 5.1 with parallel BCD [98, 125] under the assumption of $\mu = O(B/\tau)$ — a fair and equally favorable case to all of these methods. We acknowledge that more general treatments of ESO property in more recent extensions of [125] in a similar flavor as our (5.7) (see e.g., [118]) but similar results are not available for the asynchronous version. To facilitate comparison, we

---

[2] Observe that C_f^τ does not have an n^2 term in the denominator to cancel out the numerator. This is because the objective function is not appropriately scaled with n like it does in the structural SVM formulation.

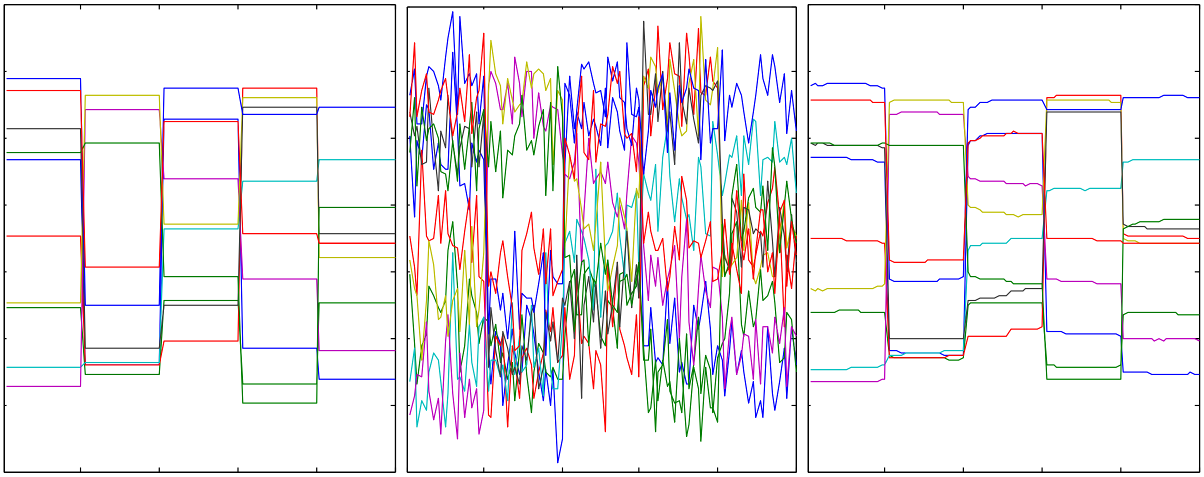


Figure D.1: Illustration of the signal data used in the Fused Lasso experiments. We show the original signal (left), the noisy signal given to the algorithm (middle), and the signal recovered after performing the fused lasso optimization (right).

will convert the constants in all three methods to block coordinate gradient Lipschitz constant L_i, which obeys

$$f(x + s_{[i]}) \leq f(x) + \langle s_{[i]}, \nabla f(x) \rangle + L_i \|s_{[i]}\|^2, \tag{D.11}$$

for any $x \in \mathcal{M}$, $s_{(i)} \in \mathcal{M}_i$. Observe that $B_i \leq 4L_i \mathrm{diam}(\mathcal{M}_i)^2 = L_i \max_{x_i^*, x_i \in \mathcal{M}_i} \|x_i - x_i^*\|^2$, so

$$B \leq \frac{1}{n} \sum_i L_i \max_{x_i, x_i^*} \|x_i - x_i^*\| \tag{D.12}$$

$$\leq \frac{1}{n} \sum_i L_i \max_x \|x - x^*\|^2 = \mathbb{E}_i(L_i) R^2 \tag{D.13}$$

where $R := \max_x \|x - x^*\|$. The rate of convergence for the three methods (with τ oracle calls considered as one iteration) are given below.

Method	Rate
AP-BCFW (Ours)	$O_p\left(\frac{n\mathbb{E}_i(L_i)R^2}{\tau k}\right)$
P-BCD[3]	$O_p\left(\frac{n\mathbb{E}_i(L_i)R^2}{\tau k}\right)$
AP-BCD[4]	$O_p\left(\frac{n \max_i L_i R^2}{\tau k}\right)$

The comparison illustrates that these methods have the same $O(1/k)$ rate and almost the same dependence on n and τ despite the fact that we use a much simpler linear oracle. Nothing comes for free though: Nesterov acceleration does not apply for Frank-Wolfe based methods in general, while a careful implementation of parallel coordinate descents can achieve $O(1/k^2)$ rate without any full-vector interpolation in every iteration [46]. Also, Frank-Wolfe methods usually need additional restrictive conditions or algorithmic steps to get linear convergence for strongly convex problems.

---

[3] In [125, Theorem 19]

[4] In [98, Theorem 3]

These facts somewhat limits the applicability of our method to cases when projection can be computed as efficiently as (5.2). However, as is surveyed in [69], there are many interesting cases when (5.2) is much cheaper than projections, e.g., projection onto a nuclear norm ball takes $O(n^3)$ while (5.2) takes only $O(n^2)$.

Lastly, we note that in the fully asynchronous setting, we obtained an exponential improvement on the dependence of delay comparing to that in [98]. It is unclear whether this is a unique property of the block-coordinate Frank-Wolfe algorithm or similar results can be obtained for projection based block-coordinate descent.

Bibliography

[1] Accurate, large minibatch sgd: Training imagenet in 1 hour. xviii, 98, 107, 111, 112, 113

[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. URL `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`. 17

[3] P.-A. Absil, R. Mahony, and B. Andrews. Convergence of the iterates of descent methods for analytic cost functions. *SIAM Journal on Optimization*, 16(2):531–547, 2005. 64

[4] Alekh Agarwal and John C. Duchi. Distributed delayed stochastic optimization. In *Advances in Neural Information Processing Systems 24*, pages 873–881. 2011. 62, 67, 69, 76

[5] Damla S Ahipasaoglu, Peng Sun, and Michael J Todd. Linear convergence of a modified frank–wolfe algorithm for computing minimum-volume enclosing ellipsoids. *Optimisation Methods and Software*, 23(1):5–19, 2008. 77

[6] Amr Ahmed, Mohamed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012. 12, 41

[7] Carlos M Alaíz, Álvaro Barbero, and José R Dorronsoro. Group fused lasso. In *Artificial Neural Networks and Machine Learning–ICANN 2013*, pages 66–73. Springer, 2013. 78

[8] Apache. Apache hadoop. https://hadoop.apache.org/, . 41, 59, 62

[9] Apache. Apache mahout. http://mahout.apache.org/, . 2

[10] Hedy Attouch and Jerome Bolte. On the convergence of the proximal algorithm for nonsmooth functions involving analytic features. *Mathematical Programming*, 116(1-2):5–16, 2009. ISSN 0025-5610. 63, 64

[11] Hedy Attouch, Jerome Bolte, Patrick Redont, and Antoine Soubeyran. Proximal alternating minimization and projection methods for nonconvex problems: An approach based on the Kurdyka-Łojasiewicz inequality. *Mathematics of Operations Research*, 35(2):438–457, 2010. 64, 68

[12] Hedy Attouch, Jerome Bolte, and BenarFux Svaiter. Convergence of descent methods for semi-algebraic and tame problems: proximal algorithms, forward-backward splitting, and regularized Gauss-Seidel methods. *Mathematical Programming*, 137(1-2):91–129, 2013. 63

[13] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Quantifying eventual consistency with pbs. *The VLDB Journal*, 23(2):279–302, 2014. 114

[14] Amir Beck and Marc Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J. Img. Sci.*, 2(1):183–202, 2009. 62, 64

[15] Amir Beck and Marc Teboulle. Smoothing and first order methods: A unified framework. *SIAM Journal on Optimization*, 22(2):557–580, 2012. 65

[16] Amir Beck and Luba Tetruashvili. On the convergence of block coordinate descent type methods. *SIAM Journal on Optimization*, 23(4):2037–2060, 2013. 78

[17] Aurélien Bellet, Yingyu Liang, Alireza Bagheri Garakani, Maria-Florina Balcan, and Fei Sha. Distributed Frank-Wolfe algorithm: A unified framework for communication-efficient sparse learning. *CoRR*, abs/1404.2644, 2014. 90

[18] Dimitri P. Bertsekas and John N. Tsitsiklis. Convergence rate and termination of asynchronous iterative algorithms. In *Proceedings of the 3rd International Conference on Supercomputing*, pages 461–470, 1989. 75

[19] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. 62, 67, 75, 140

[20] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738. 2, 3

[21] Kevin Bleakley and Jean-Philippe Vert. The group fused Lasso for multiple change-point detection. arXiv preprint arXiv:1106.4199, 2011. 78

[22] David M. Blei, Andrew Ng, and Michael Jordan. Latent dirichlet allocation. *JMLR*, 3: 993–1022, 2003. 13

[23] Jerome Bolte, Aris Daniilidis, and Adrian Lewis. The Łojasiewicz inequality for nonsmooth subanalytic functions with applications to subgradient dynamical systems. *SIAM Journal on Optimization*, 17:1205–1223, 2007. 63

[24] Jéróme Bolte, Aris Danilidis, Olivier Ley, and Laurent Mazet. Characterizations of Łojasiewicz inequalities and applications: Subgradient flows, talweg, convexity. *Transactions of the American Mathematical Society*, 362(6):3319–3363, 2010. 64

[25] Jerome Bolte, Shoham Sabach, and Marc Teboulle. Proximal alternating linearized minimization for nonconvex and nonsmooth problems. *Mathematical Programming*, 146(1-2): 459–494, 2014. 62, 63, 64, 65, 143

[26] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004. ISBN 0521833787. 11

[27] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends in Machine Learning*, 3(1):1–122, 2010. 71

[28] Kristian Bredies, Dirk A Lorenz, and Peter Maass. A generalized conditional gradient method and its connection to an iterative shrinkage method. *Computational Optimization and Applications*, 42(2):173–193, 2009. 77

[29] K Canini, T Chandra, E Ie, J McFadden, K Goldman, M Gunter, J Harmsen, K LeFevre, D Lepikhin, TL Llinares, et al. Sibyl: A system for large scale supervised machine learning. *Technical Talk*, 2012. 1, 113

[30] Antonios Chalkiopoulos. *Programming MapReduce with Scalding*. Packt Publishing Ltd, 2014. 114

[31] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016. xvii, xviii, 9, 10, 106, 107, 109

[32] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi`. 9, 101

[33] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *Artificial Intelligence and Statistics*, pages 192–204, 2015. 112

[34] Kenneth L Clarkson. Coresets, sparse greedy approximation, and the Frank-Wolfe algorithm. *ACM Transactions on Algorithms (TALG)*, 6(4):63, 2010. 77

[35] Michael Collins, Amir Globerson, Terry Koo, Xavier Carreras, and Peter L Bartlett. Exponentiated gradient algorithms for conditional random fields and max-margin markov networks. *JMLR*, 9:1775–1822, 2008. 78

[36] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 37–48, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-10-2. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/cui`. xvii, 9, 100

[37] Henggang Cui, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky, Qirong Ho, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, et al. Exploiting iterative-ness for parallel ml computations. In *SoCC*, pages 1–14. ACM, 2014. 53

[38] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter

server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016. 9, 10, 101

[39] Wei Dai, Jinliang Wei, Xun Zheng, Jin Kyu Kim, Seunghak Lee, Junming Yin, Qirong Ho, and Eric P Xing. Petuum: a framework for iterative-convergent distributed ML. *arXiv:1312.7651*, 2013. 79

[40] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P. Xing. Analysis of high-performance distributed ml at scale through parameter server consistency models. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, 2015. 10, 62, 69, 74, 98, 101, 106, 109

[41] Wei Dai, Chia Dai, Shuhui Qu, Juncheng Li, and Samarjit Das. Very deep convolutional neural networks for raw waveforms. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2017*. IEEE, 2017. 110

[42] Jeff Dean. Achieving rapid response times in large online services. In *Berkeley AMPLab Cloud Seminar*, 2012. 101

[43] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012. 9, 10, 106, 114

[44] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12: 2121–2159, 2011. 11

[45] Jianqing Fan and Runze Li. Variable selection via nonconcave penalized likelihood and its oracle properties. *Journal of the American Statistical Association*, 96(456):1348–1360, 2001. 62, 67

[46] Olivier Fercoq and Peter Richtárik. Accelerated, parallel, and proximal coordinate descent. *SIAM Journal on Optimization*, 25(4):1997–2023, 2015. 172

[47] H.R. Feyzmahdavian and M. Johansson. On the convergence rates of asynchronous iterations. In *2014 IEEE 53rd Annual Conference on Decision and Control*, pages 153–159, 2014. 75

[48] H.R. Feyzmahdavian, A. Aytekin, and M. Johansson. A delayed proximal gradient method with linear convergence rate. In *2014 IEEE International Workshop on Machine Learning for Signal Processing*. 62, 67, 69, 76

[49] Rina Foygel, Michael Horrell, Mathias Drton, and John D Lafferty. Nonparametric reduced rank regression. In *NIPS'12*, pages 1628–1636, 2012. 78

[50] Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1-2):95–110, 1956. 2, 77

[51] Robert M. Freund and Paul Grigas. New analysis and results for the frank–wolfe method. *Mathematical Programming*, 155(1):199–230, 2014. ISSN 1436-4646. 78

[52] Satoru Fujishige and Shigueo Isotani. A submodular function minimization algorithm based on the minimum-norm base. *Pacific Journal of Optimization*, 7(1):3–17, 2011. 78

[53] Masao Fukushima and Hisashi Mine. A generalized proximal point algorithm for certain non-convex minimization problems. *International Journal of Systems Science*, 12(8):989–1000, 1981. 62, 64, 65

[54] Dan Garber and Elad Hazan. A linearly convergent conditional gradient algorithm with applications to online and stochastic optimization. *arXiv:1301.4666*, 2013. 77

[55] Alexander Genkin, David D. Lewis, and David Madigan. Large-scale bayesian logistic regression for text categorization. *Technometrics*, page 2007. 11

[56] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010. 105

[57] Joseph E Gonzalez, Peter Bailis, Michael I Jordan, Michael J Franklin, Joseph M Hellerstein, Ali Ghodsi, and Ion Stoica. Asynchronous complex analytics in a distributed dataflow architecture. *arXiv preprint arXiv:1510.07092*, 2015. 98

[58] Thomas L. Griffiths and Mark Steyvers. Finding scientific topics. *PNAS*, 101(suppl. 1): 5228–5235, 2004. 12

[59] Stefan Hadjis, Ce Zhang, Ioannis Mitliagkas, Dan Iter, and Christopher Ré. Omnivore: An optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*, 2016. 98, 104, 106, 114

[60] Zaid Harchaoui, Anatoli Juditsky, and Arkadi Nemirovski. Conditional gradient algorithms for norm-regularized smooth convex optimization. *Mathematical Programming*, 152(1-2):75–112, 2015. ISSN 0025-5610. 77

[61] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems (TiiS)*, 2016. 17

[62] Elad Hazan and Satyen Kale. Projection-free online learning. In *ICML'12*, 2012. 77

[63] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015. 21

[64] Geoffrey Hinton. Neural networks for machine learning. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`, 2012. 11, 17

[65] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Greg Ganger, and Eric Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in Neural Information Processing Systems (NIPS) 26*, pages 1223–1231. 2013. URL `http://media.nips.cc/nipsbooks/nipspapers/paper_files/nips26/631.pdf`. xviii, 9, 10, 11, 13, 14, 18, 45, 46, 47, 49, 53, 58, 62, 63, 67, 69, 70, 74, 76, 98, 108, 114

[66] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pages 1729–1739, 2017. 111

[67] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. 110

[68] Martin Jaggi. *Sparse convex optimization methods for machine learning*. PhD thesis,

Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 20013, 2011, 2011. 77, 161

[69] Martin Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In *ICML'13*, pages 427–435, 2013. 77, 78, 81, 82, 85, 153, 168, 172

[70] Stefanie Jegelka, Francis Bach, and Suvrit Sra. Reflection methods for user-friendly submodular optimization. In *NIPS'13*, pages 1313–1321, 2013. 78

[71] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems*, pages 315–323, 2013. 109

[72] Michael I Jordan et al. On statistics, computation and scalability. *Bernoulli*, 19(4):1378–1390, 2013. 95

[73] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016. xviii, 111, 112, 113

[74] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A Gibson, and Eric P Xing. Strads: a distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 5. ACM, 2016. 14

[75] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 11, 17

[76] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013. xiii, 14, 16

[77] Yehuda Koren, Robert M. Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009. 55

[78] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. Imagenet classification with deep convolutional neural networks. In P. Bartlett, F.C.N. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1106–1114. 2012. URL `http://books.nips.cc/papers/files/nips25/NIPS2012_0534.pdf`. 11

[79] Abhimanu Kumar, Alex Beutel, Qirong Ho, and Eric P Xing. Fugue: Slow-worker-agnostic distributed learning for big models on big data. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pages 531–539, 2014. 14

[80] Krzysztof Kurdyka. On gradients of functions definable in o-minimal structures. *Annales de l'institut Fourier*, 48(3):769–783, 1998. 63, 64

[81] Simon Lacoste-Julien and Martin Jaggi. On the global linear convergence of Frank-Wolfe optimization variants. In *NIPS'15*, pages 496–504, 2015. 77

[82] Simon Lacoste-Julien, Martin Jaggi, Mark Schmidt, and Patrick Pletscher. Block-coordinate Frank-Wolfe optimization for structural svms. In *ICML'13*, pages 53–61, 2013. xvii, 77, 78, 81, 82, 86, 89, 90, 153, 154, 155, 169, 170

[83] Jean Lafond, Hoi-To Wai, and Eric Moulines. Convergence analysis of a stochastic projection-free algorithm. *arXiv:1510.01171*, 2015. 77

[84] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010. 101

[85] J Langford, L Li, and A Strehl. Vowpal wabbit online learning project, 2007. 59

[86] John Langford, Er J. Smola, and Martin Zinkevich. Slow learners are fast. In *In NIPS*, pages 2331–2339, 2009. 9, 10, 11

[87] Quoc Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg Corrado, Jeff Dean, and Andrew Ng. Building high-level features using large scale unsupervised learning. In *International Conference in Machine Learning*, 2012. 1

[88] Larry J LeBlanc, Edward K Morlok, and William P Pierskalla. An efficient approach to solving the road network equilibrium traffic assignment problem. *Transportation Research*, 9(5):309–318, 1975. 78

[89] Y LeCun, L Bottou, G Orr, and K Muller. Efficient backprop in neural networks: Tricks of the trade (orr, g. and müller, k., eds.)[j]. *Lecture Notes in Computer Science*, 1524. 98, 111

[90] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998. 16, 17

[91] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A Gibson, and Eric P Xing. On model parallelization and scheduling strategies for distributed machine learning. In *NIPS'14*, pages 2834–2842, 2014. 90

[92] Mu Li, David G Andersen, and Alexander Smola. Distributed delayed proximal gradient methods. *Big Learning NIPS Workshop*, 2013. 67, 75

[93] Mu Li, Li Zhou Zichao Yang, Aaron Li Fei Xia, David G. Andersen, and Alexander Smola. Parameter server for distributed machine learning. *NIPS workshop*, 2013. 58, 59, 75, 98

[94] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598, 2014. xviii, 62, 63, 69, 70, 75, 79, 108

[95] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015. 9, 10, 11

[96] Ji Liu and Stephen J. Wright. Asynchronous stochastic coordinate descent: Parallelism and convergence properties. *SIAM Journal on Optimization*, 25(1):351–376, 2015. 62, 67, 69

[97] Ji Liu, Przemyslaw Musialski, Peter Wonka, and Jieping Ye. Tensor completion for estimating missing values in visual data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(1):208–220, 2013. 78

[98] Ji Liu, Stephen J Wright, Christopher Ré, and Victor Bittorf. An asynchronous parallel stochastic coordinate descent algorithm. *JMLR*, 2014. 79, 83, 85, 90, 91, 166, 170, 171, 172

[99] Jun Liu, Jianhui Chen, and Jieping Ye. Large-scale sparse logistic regression. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 547–556. ACM, 2009. 11

[100] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010. 41, 62

[101] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *PVLDB*, 2012. 41, 58

[102] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. 41

[103] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018. 111

[104] Rahul Mazumder, Jerome H. Friedman, and Trevor Hastie. Sparsenet: Coordinate descent with nonconvex penalties. *Journal of the American Statistical Association*, 106(495): 1125–1138, 2011. 62, 67

[105] Brendan McMahan and Matthew Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. In *Advances in Neural Information Processing Systems*, pages 2915–2923, 2014. 101, 113

[106] H Brendan McMahan. Follow-the-regularized-leader and mirror descent: Equivalence theorems and l1 regularization. In *International Conference on Artificial Intelligence and Statistics*, pages 525–533, 2011. 11, 17

[107] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2013. 1, 11

[108] Frank McSherry, Michael Isard, and Derek G Murray. Scalability! but at what cost? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015. 114

[109] Ioannis Mitliagkas, Ce Zhang, Stefan Hadjis, and Christopher Ré. Asynchrony begets momentum, with an application to deep learning. In *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*, pages 997–1004. IEEE, 2016. xvii, 19, 23, 104, 105

[110] Michael Mitzenmacher. The power of two choices in randomized load balancing. *Parallel and Distributed Systems, IEEE Transactions on*, 12(10):1094–1104, 2001. 153, 157

[111] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010. 12

[112] Willie Neiswanger, Chong Wang, and Eric Xing. Asymptotically exact, embarrassingly parallel mcmc. *arXiv preprint arXiv:1311.4780*, 2013. 95

[113] Yu Nesterov. Efficiency of coordinate descent methods on huge-scale optimization problems. *SIAM Journal on Optimization*, 22(2):341–362, 2012. 78

[114] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS'11*, pages 693–701, 2011. 79, 85, 90, 163, 166

[115] Hua Ouyang and Alexander G Gray. Fast stochastic Frank-Wolfe algorithms for nonlinear SVMs. In *SDM*, 2010. 77

[116] Russell Power and Jinyang Li. Piccolo: building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association. 41

[117] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999. 17

[118] Zheng Qu and Peter Richtárik. Coordinate descent with arbitrary sampling ii: Expected separable overapproximation. *arXiv preprint arXiv:1412.8063*, 2014. 170

[119] Martin Raab and Angelika Steger. Balls into bins - a simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998. 157

[120] Benjamin Recht, Christopher Re, Stephen J. Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011. 9, 41, 62, 69, 76

[121] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex Smola. Stochastic variance reduction for nonconvex optimization. In *International conference on machine learning*, pages 314–323, 2016. 109

[122] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. In *International Conference on Learning Representations*, 2018. xiii, 17, 20

[123] Jason Rennie. 20 newsgroups. http://qwone.com/ jason/20Newsgroups/. 17

[124] P. Richtárik and M. Takáč. Distributed Coordinate Descent Method for Learning with Big Data. *arXiv*, 2013. 71

[125] Peter Richtárik and Martin Takáč. Parallel coordinate descent methods for big data optimization. *Mathematical Programming*, pages 1–52, 2015. 78, 79, 83, 90, 170, 171

[126] R.T. Rockafellar and R.J.B. Wets. *Variational Analysis*. Springer, 1997. 63, 64

[127] Mark Schmidt, Nicolas L. Roux, and Francis R. Bach. Convergence rates of inexact proximal-gradient methods for convex optimization. In *Advances in Neural Information Processing Systems 24*, pages 1458–1466. 2011. 70, 151

[128] Steven L Scott, Alexander W Blocker, Fernando V Bonassi, Hugh A Chipman, Edward I George, and Robert E McCulloch. Bayes and big data: The consensus monte carlo algorithm. *International Journal of Management Science and Engineering Management*, 11 (2):78–88, 2016. 95

[129] Yiyuan She. Thresholding-based iterative selection procedures for model selection and shrinkage. *Electronic Journal of Statistics*, 3:384–415, 2009. 62

[130] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 1, 11

[131] Alexander Smola and Shravan Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1-2):703–710, September 2010. ISSN 2150-8097. 9, 10, 13

[132] Daniel Soudry and Yair Carmon. No bad local minima: Data independent training error guarantees for multilayer neural networks. *arXiv preprint arXiv:1605.08361*, 2016. 112

[133] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 2014. 110

[134] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 607–614, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0632-4. doi: 10.1145/1963405.1963491. URL `http://doi.acm.org/10.1145/1963405.1963491`. 41

[135] Jian Tang, Zhaoshi Meng, Xuanlong Nguyen, Qiaozhu Mei, and Ming Zhang. Understanding the limiting factors of topic modeling via posterior contraction analysis. In *International Conference on Machine Learning*, pages 190–198, 2014. 95

[136] Ben Taskar, Carlos Guestrin, and Daphne Koller. Max-margin Markov networks. In *NIPS'04*, pages 25–32. MIT Press, 2004. xvii, 86

[137] Paul Tseng. On the rate of convergence of a partially asynchronous gradient projection algorithm. *SIAM Journal on Optimization*, 1(4):603–619, 1991. 62, 67, 75

[138] J.N. Tsitsiklis, D.P. Bertsekas, and M. Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Transactions on Automatic Control*, 31 (9):803–812, 1986. 75

[139] John N Tsitsiklis, Dimitri P Bertsekas, Michael Athans, et al. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Transactions on Automatic Control*, 31(9), 1986. 90

[140] Leslie G. Valiant. A bridging model for parallel computation. *Communications of ACM*, 33(8):103–111, 1990. 62

[141] Stefan Wager, Sida Wang, and Percy Liang. Dropout training as adaptive regularization. In *Advances in Neural Information Processing Systems*, pages 351–359, 2013. 110

[142] Yi Wang, Hongjie Bai, Matt Stanton, Wen-Yen Chen, and Edward Y. Chang. Plda: Parallel latent dirichlet allocation for large-scale applications. In *Proceedings of the 5th International Conference on Algorithmic Aspects in Information and Management*, AAIM '09, pages 301–314, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02157-2. 59

[143] Yu-Xiang Wang, Veeranjaneyulu Sadhanala, Wei Dai, Willie Neiswanger, Suvrit Sra, and Eric P Xing. Parallel and distributed block-coordinate frank-wolfe algorithms. 2016. 109

[144] Zhaoran Wang, Han Liu, and Tong Zhang. Optimal computational and statistical rates of convergence for sparse nonconvex learning problems. *The Annals of Statistics*, 42(6): 2164–2201, 2014. 62

[145] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 381–394. ACM, 2015. 10, 18, 95, 98, 108

[146] E.P. Xing, Q. Ho, W. Dai, Jin-Kyu Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. *Big Data, IEEE Transactions on*, PP(99):1–1, 2015. doi: 10.1109/TBDATA.2015.2472014. 93

[147] Limin Yao, David Mimno, and Andrew McCallum. Efficient methods for topic model inference on streaming document collections. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 937–946, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9. 12, 55

[148] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, September 1974. ISSN 0001-0782. doi: 10.1145/361147.361115. URL `http://doi.acm.org/10.1145/361147.361115`. 54

[149] Chun-Nam John Yu and Thorsten Joachims. Learning structural svms with latent variables. In *ICML'09*, pages 1169–1176. ACM, 2009. 169

[150] Hsiang-Fu Yu, Hung-Yi Lo, Hsun-Ping Hsieh, Jing-Kai Lou, Todd G McKenzie, Jung-Wei Chou, Po-Han Chung, Chia-Hua Ho, Chun-Fu Chang, Yin-Hsuan Wei, et al. Feature engineering and classifier ensemble for kdd cup 2010. *KDD Cup*, 2010. 11

[151] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *ICDM*, pages 765–774, 2012. 59

[152] Yaoliang Yu, Xun Zheng, Micol Marchetti-Bowick, and Eric P. Xing. Minimizing nonconvex non-separable functions. In *The* 17^{th} *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2015. 62, 65

[153] Jinhui Yuan, Fei Gao, Qirong Ho, Wei Dai, Jinliang Wei, Xun Zheng, Eric Po Xing, Tie-Yan Liu, and Wei-Ying Ma. Lightlda: Big topic models on modest computer clusters. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1351–1361. ACM, 2015. 1, 10

[154] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. pages 10–10, 2010. 41, 59, 62, 114

[155] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012. 17

[156] Cun-Hui Zhang. Nearly unbaised variable selection under minimax concave penalty. *Annals of Statistics*, 38(2):894–942, 2010. 62, 67

[157] Cun-Hui Zhang and Tong Zhang. A general theory of concave regularization for high-dimensional sparse estimation problems. *Statistical Science*, 27(4):576–593, 2012. 62, 67

[158] Ruiliang Zhang and James T. Kwok. Asynchronous distributed admm for consensus optimization. In *ICML*, 2014. 9, 10

[159] Xinhua Zhang, Yaoliang Yu, and Dale Schuurmans. Accelerated training for matrix-norm regularization: A boosting approach. In *NIPS'12*, pages 2915–2923, 2012. 78

[160] Xinhua Zhang, Yao-Liang Yu, and Dale Schuurmans. Polar operators for structured sparse estimation. In *NIPS'13*, pages 82–90, 2013. 77

[161] Yi Zhou, Yaoliang Yu, Wei Dai, Yingbin Liang, and Eric P. Xing. On convergence of model parallel proximal gradient algorithm for stale synchronous parallel system. In *The 19th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2016. 101

[162] Yunzhang Zhu, Xiaotong Shen, and Wei Pan. Simultaneous grouping pursuit and feature selection over an undirected graph. *Journal of the American Statistical Association*, 108 (502):713–725, 2013. 62