

EFFICIENT AND SCALABLE MACHINE LEARNING
FOR DISTRIBUTED EDGE INTELLIGENCE

A Dissertation

by

JYOTIKRISHNA DASS

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Chair of Committee,	Rabi Mahapatra
Committee Members,	Eun Jung Kim
	Xia Hu
	Raktim Bhattacharya
Head of Department,	Scott Schaefer

August 2021

Major Subject: Computer Engineering

Copyright 2021 Jyotikrishna Dass

ABSTRACT

In the era of big data and IoT, devices at the edges are becoming increasingly intelligent, and processing the data closest to the sources is paramount. However, conventional machine learning works with the centralized framework of collecting data from various edge sources and storing it on the high-performance cloud to support computationally intensive iterative algorithms. As a result, such a framework is infeasible for training on embedded edge devices with limited resources and a tight power budget. This dissertation proposes to integrate ideas from fields of machine learning and systems for designing efficient and scalable algorithms for distributed training of machine learning models amenable for edge computing with limited hardware and computing resources. The resulting decentralized machine learning framework aims to keep the data private, reduce latency, save communication bandwidth, be energy-efficient, and handle streaming data.

DEDICATION

In loving memory of my grandparents Hadiani Dass, Shiba Prasad Dass, Sarangadhara Laxman, and Premalata Laxman, whom I lost in the course of my education and could not bid them a last farewell! This is for them and the sacrifices they made to nurture and educate my parents, Supriya Dass and Dr. Jagannath Dass; on shoulders of and with blessings from whom I could embark on my pilgrimage to self-discovery.

—
TVAMEVA MATA CHA PITA TVAMEVA, TVAMEVA BANDHU CHA SAKHA TVAMEVA |
TVAMEVA VIDYA DRAVINAM TVAMEVA, TVAMEVA SARVAM MAMA DEVA DEVA ||

*Verse 28, Pandava Gita*¹

—

¹http://mahabharata-resources.org/related/Pandava_Gita_translation.pdf

ACKNOWLEDGMENTS

This dissertation is a fruit of hours of efforts, discussions, patience, and perseverance spread across days, nights, weeks, months, and years that saw three presidents sit at the White House and the world facing its most devastating pandemic in our lifetime. I shall certainly fail in my duty if I do not express my deep sense of gratitude to the hardworking and selfless frontline workers, researchers, and policy-makers for keeping me and my family back home, healthy and vaccinated during such challenging time.

PhD is a marathon which I lived each moment with its ups and downs, some setbacks here and there but still running, never giving up, and completing milestones along the way. As I approach the final leg of this spiritual journey, which has been a slice of life, beautiful in its both pain and glory, I take this opportunity to thank the following people who have helped shape me as a researcher, teacher, mentor, and friend during my candidature.

I wish to express my sincere gratitude and appreciation to Professor Rabi Mahapatra who as my thesis advisor has been a source of valuable inspiration, patient guidance, invaluable suggestions, and heartfelt encouragement. The trust and the independence he provided me since my early days in the lab to explore and shape my research interests is the reason why I continue to love my work and the academic lifestyle. During our discussions, he always helped me see the bigger picture be it in research or in life in general, and that has been the hallmark of his expert mentorship. He always mentioned that whatever one sought and needed is already within oneself; this teaching has helped me dive deeper to grow both spiritually and academically during my PhD.

I am thankful to my dissertation committee members, Dr. Eun Jung Kim, Dr. Xia Hu, and Dr. Raktim Bhattacharya for their timely advice, and support in my research. I learnt about grant writing from my collaboration with Dr. Eun Jung Kim. and Dr. Shuiwang Ji. My first success with research and publication during my PhD was possible in collaboration with Dr. Raktim Bhattacharya. Our work inspired me to identify and lay the foundation of my research which now has

taken a form of this dissertation. I am grateful to Dr. Xia Hu for his mentorship, inspiration, networking and welcoming me to be a regular attendee in his lab group meetings. These interactions helped me expand my knowledge on topics beyond my research. This dissertation would not have been completed if it was not for Dr. Hu's selfless consideration to support me as a research assistant last Spring semester to help me channel all my efforts to completing my research as a part of this dissertation.

I very much appreciate the technical discussions and collaboration with Dr. Vivek Sarin on our multiple research publications. I take this opportunity to also thank my other collaborators, namely, Kooktae Lee, Dharanidhar Dang, V.N.S. Prithvi Sakuru, Yashwardhan Narawane, Nathan Purwosumarto, and Rengang Yang. My interactions with them helped me grow as a researcher and a mentor. There is no bigger joy as a PhD candidate than learning from senior PhD students while also getting opportunities to mentor young masters and undergraduate students on research. I thank them for their faith in me and for their contributions as co-authors in our research publications.

I owe a great deal to Dr. Aakash Tyagi for his sincere mentorship, trust and friendship whenever I needed someone to share my happiness and sorrows. My love for teaching has strengthened under his expert leadership, and observing his empathy and care towards the students. Some of my most cherished memories have been teaching CSCE 312 as both instructor and guiding students in the labs as his TA. I also express my sincere thanks to Dr. Joseph Hurley, and Dr. Michael Quinn who entrusted me as their TA for their programming courses CSCE 111/121/206. I am also thankful to Dr. Dylan Shell and Dr. Vikram Kinra for their guidance during my appointment as Graduate Teaching Fellow. I feel honored to have the opportunity to share the space and time with all of them and the thousands of undergraduate students, peer teachers, and graders in my various teaching roles spanning my graduate studies. These interactions have helped me touch the lives of many young students that have made me a better teacher, broadened my perspective about life, and enriched my experience at Texas A&M University. I gratefully acknowledge the financial support via various teaching assistantships and conducive research facilities I have received from the Department of Computer Science & Engineering at Texas A&M University.

I would also like to thank my amazing labmates Dharanidhar Dang, Biplab Patra, V.N.S. Prithvi Sakuru, Yashwardhan Narawane, Akash Sahoo, Jerry Yiu, and Karl Ott for the priceless memories and discussions about research and life. A special thank you to my dear friends, Vipul, Rahul, Vasav, Shweta, Ateesh, Sakshi, Sudhanshu, Prathamesh, Nihar, Rohit, Waseem, Chinmay, Sreyashree, Vaishali, Abhinaba, Suma, Siri and Pulakesh for their friendship during my graduate studies. My life at College Station has been made richer and joyful with them to share my experiences. I have lost count of how many graduation ceremonies I have attended in all these years and hope to see them soon as I prepare to close my chapter at Texas A&M. My heartfelt gratitude to Shalvi for her selfless affection and her constant encouragement in my undergraduate days which put me in path to pursue PhD. I am indebted to her for all she did for me and my mother as I adjusted to my graduate life in the USA. I am grateful to my partner, Shreya, who has been my pillar of strength and support, and my family away from home. I can not thank her enough for her love, patience, and understanding. She makes our bond stronger despite the physical distance between us. I can not wait to begin the next chapter of our lives. It's long overdue and I look forward to making memories together.

Personally, I owe a lot to my sister, Shuvalaxmi and my parents for their untiring patience and devotion. Our parents chose to sacrifice their time with their children and live alone miles away in India so that we could get quality education and find our purpose in life. They have suffered personal losses in family and spent their time in grief all by themselves, shielding us from all the pain. My sincere appreciation to all my extended family members back in India for being always present for them when they needed me the most. With the close of this chapter of my life, it is time to give my parents everything we have to offer, make up for the lost time and take care of them. I am immensely proud of my talented sister to be finishing her PhD at Texas Tech University and securing a tenure-track job. She will be the first woman PhD from any of our extended family and her journey is a great source of inspiration for all of us, and for her future students. I am immensely indebted to my father who did his PhD from IIT Delhi and whose dissertation at the IIT library inspired and sowed the seeds of curiosity for me at an early age. He is my strongest supporter even

though he does not show it! My mother is my world and she has been my guiding light throughout my life. She inspires me with her strength and faith at times of adversity. She has sacrificed the most to nurture us and make us capable beings with heart of gold. I wish our parents could make it to our graduation ceremonies but the current circumstances with pandemic might very well take that opportunity away once again. Such is the unpredictability of life that I have been unfortunate to not have them witness any of my graduations in-person. I hope and pray for a better tomorrow for all of us together.

Finally, I would like to thank the support staff in the CSE department and at TAMU for a great learning atmosphere, and my sincere acknowledgments to the anonymous reviewers, area chairs, and editors of various conferences and journals for their selfless time, invaluable suggestions, and constructive feedback on my research articles culminating into various peer-reviewed publications as a part of this dissertation.

Though every PhD is a unique and personal journey for each candidate, it truly takes a village to see one make it till the end. Thank y'all for being a part of my incredible journey, and for being my loudest cheerleaders. Thanks and Gig'em!

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by dissertation committee consisting of Professor Rabi Mahapatra (advisor), Professor Eun Jung Kim, and Professor Xia Hu, of the Department of Computer Science and Engineering and Professor Raktim Bhattacharya of the Department of Aerospace Engineering.

The theoretical analysis in Chapter 2 was performed in collaboration with Kookate Lee of the Department of Aerospace Engineering and was published in [1]. The experimental analysis depicted in Chapter 3 and Chapter 7 were conducted in part by Rengang Yang and Nathan Purwosumarto, respectively, of the Department of Computer Science and Engineering. The mathematical brainstorming for Chapter 4 was done in part with Professor Vivek Sarin of the Department of Computer Science and Engineering and was published in [2]. The experimental analysis depicted in Chapter 4 and Chapter 6 were conducted in part by V.N.S. Prithvi Sakuru and Yashwardhan Narawane, respectively, of the Department of Computer Science and Engineering and were published in [2] and [3], respectively.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study at Texas A&M University was supported by Graduate Teaching Assistantships from the Department of Computer Science and Engineering, Graduate Teaching Fellowship from the College of Engineering.

NOMENCLATURE

ADMM	Alternating Direction Method of Multipliers
AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuit
CPU	Central Processing Unit
DNN	Deep Neural Networks
FPGA	Field-Programmable Gate Array
GPU	Graphic Processing Unit
HPC	High Performance Computing
IoT	Internet of Things
IPM	Interior Point Methods
KRR	Kernel Ridge Regression
LSDA	Lazily Synchronized Dual Ascent
ML	Machine Learning
QP	Quadratic Programming
QRSVM	QR decomposition based Support Vector Machines
RIVER	Rapid Incremental Solver
SGD	Stochastic Gradient Descent
SMO	Sequential Minimal Optimization
SVM	Support Vector Machines
TSDA	Tightly Synchronized Dual Ascent
TSQR	Tall and Skinny QR decomposition

TABLE OF CONTENTS

	Page
ABSTRACT	ii
DEDICATION	iii
ACKNOWLEDGMENTS	iv
CONTRIBUTORS AND FUNDING SOURCES	viii
NOMENCLATURE	ix
TABLE OF CONTENTS	x
LIST OF FIGURES	xv
LIST OF TABLES.....	xxi
1. INTRODUCTION.....	1
1.1 Challenges	4
1.1.1 Interior Point Method	4
1.1.2 Sequential Minimal Optimization	5
1.1.3 Stochastic Gradient Descent	5
1.1.4 Gaps	6
1.2 Motivation	6
1.3 Dissertation Philosophy.....	7
1.4 Contributions	9
1.5 Organization.....	12
2. RELAXED SYNCHRONIZATION FOR PARALLEL QUADRATIC PROGRAMMING	13
2.1 Introduction.....	13
2.1.1 Motivation	14
2.1.2 Contributions	14
2.2 Related Work	15
2.3 Problem Formulation	16
2.4 Lazily Synchronized Dual Ascent (LSDA) Algorithm.....	18
2.4.1 Theory.....	19
2.4.2 Optimal Synchronization Period.....	21
2.4.3 Implementation	23

2.4.4	Theoretical Speedup	24
2.5	Experiment and Results	26
2.5.1	Hardware Description	27
2.5.2	Experimental Setup	28
2.5.3	Results and Discussions	28
2.5.3.1	Synchronization Period	28
2.5.3.2	Convergence	31
2.5.3.3	Computation Time	32
2.5.3.4	Communication Time	33
2.5.3.5	Speedup	35
2.6	Summary	35
3.	HOUSEHOLDER SKETCH FOR MACHINE LEARNING	36
3.1	Introduction	36
3.1.1	Motivation	36
3.1.2	Contributions	38
3.2	Related Work	39
3.3	Least Mean Squares - QR (LMS-QR)	39
3.3.1	Theory	40
3.3.2	Accelerating Least Mean Squares Solvers	42
3.4	Distributed LMS-QR	44
3.5	Experiment and Results	49
3.5.1	Hardware Description	50
3.5.2	Experimental Setup	50
3.5.3	Results and Discussions	51
3.5.3.1	Sequential Training Time	51
3.5.3.2	Distributed Training Time	53
3.5.3.3	Scalability	54
3.5.3.4	Numerical Stability	55
3.6	Summary	56
4.	MEMORY-EFFICIENT FRAMEWORK FOR DISTRIBUTED MACHINE LEARNING	57
4.1	Introduction	57
4.1.1	Motivation	58
4.1.2	Contributions	58
4.2	Related Work	59
4.3	Support Vector Machines	60
4.3.1	Formulation	60
4.3.2	Challenges	63
4.4	Memory-Efficient Framework	63
4.4.1	QRSVM	63
4.4.1.1	Formulation	63
4.4.1.2	Benefits	64

4.4.2	Optimization	65
4.4.2.1	Dual Ascent Algorithm	66
4.4.2.2	Optimal Step Size	66
4.4.2.3	Complexity Analysis	69
4.4.3	Distributed QRSVM.....	71
4.4.3.1	Distributed QR decomposition	71
4.4.3.2	Parallel Dual Ascent.....	74
4.5	Experiment and Results	75
4.5.1	Hardware Description	75
4.5.2	Experimental Setup.....	76
4.5.3	Results and Discussions.....	76
4.5.3.1	Convergence	76
4.5.3.2	Kernel Approximation Quality	77
4.5.3.3	Scalability.....	78
4.5.3.4	Distributed Training Time	79
4.5.3.5	Optimal Step Size	81
4.5.3.6	Comparison	82
4.6	Summary	83
5.	COMMUNICATION-EFFICIENT FRAMEWORK FOR SCALABLE MACHINE LEARNING	84
5.1	Introduction.....	84
5.1.1	Motivation	84
5.1.2	Contributions	84
5.2	Related Work	85
5.3	Communication-Efficient Framework.....	86
5.3.1	Implementing Distributed QR Decomposition.....	86
5.3.2	Implementing Parallel Dual Ascent	88
5.4	Experiment and Results	92
5.4.1	Hardware Description	93
5.4.2	Experimental Setup.....	93
5.4.3	Results and Discussions.....	93
5.4.3.1	Convergence	94
5.4.3.2	Scalability.....	94
5.4.3.3	Distributed Training Time	96
5.4.3.4	Comparison	100
5.5	Summary	102
6.	MULTIPLE FPGA-BASED SYSTEM FOR ENERGY-EFFICIENT TRAINING	104
6.1	Introduction.....	104
6.1.1	Motivation	105
6.1.2	Contributions	106
6.2	Related Work	107
6.3	System Overview.....	108

6.4	Accelerator Design	109
6.4.1	Microarchitecture	109
6.4.1.1	Architecture for Distributed QR Decomposition	110
6.4.1.2	Architecture for Parallel Dual Ascent	113
6.4.2	Interface Design	116
6.5	Experiment and Results	117
6.5.1	Hardware Description	118
6.5.2	Experimental Setup	119
6.5.3	IP Synthesis	119
6.5.4	Results and Discussions	120
6.5.4.1	Training Time	120
6.5.4.2	Scalability	122
6.5.4.3	Energy Efficiency	124
6.5.4.4	Comparison	126
6.6	Summary	128
7.	RAPID INCREMENTAL SOLVER FOR FEDERATED LEARNING	130
7.1	Introduction	130
7.1.1	Motivation	131
7.1.2	Contributions	131
7.2	Related Work	134
7.3	Preliminaries	135
7.3.1	Problem Setup	136
7.3.2	Problem Statement	137
7.4	Rapid Incremental Solver	138
7.4.1	RIVER-STREAM	140
7.4.2	RIVER-TRIBUTARY	143
7.4.3	RIVER-BASIN	145
7.5	Complexity Analysis	146
7.5.1	Computation Time	146
7.5.2	Memory Consumption	148
7.5.3	Communication Overhead	148
7.6	Experiment and Results	149
7.6.1	Hardware Description	149
7.6.2	Experimental Setup	149
7.6.3	Results and Discussions	150
7.6.3.1	Scalability	150
7.6.3.2	Accuracy	157
7.6.3.3	Timing Breakdown Analysis	158
7.7	Summary	160
8.	CONCLUSIONS	161
8.1	Conclusions	161
8.2	Future Directions	163

8.2.1	Secure Multi-Party Decentralized Machine Learning	163
8.2.2	Codesigned AutoML Systems for Distributed Edge Intelligence	164
8.2.3	Systems for Lifelong Multi-Agent Learning	165
REFERENCES	167

LIST OF FIGURES

FIGURE	Page
1.1 A typical large-scale machine learning process.	2
1.2 A typical centralized machine learning framework.	3
1.3 Requirements for Distributed Edge Intelligence.	7
1.4 Research Focus for efficient and scalable machine learning for distributed edge intelligence.	8
1.5 Decentralized network where all edge devices are connected to each other.	9
1.6 Decentralized network where all edge devices are connected to each other and an Edge server.	10
1.7 A typical IoT based network where multiple edge servers may communicate via cloud while major computations are handled by edge devices.	10
2.1 An illustration of the spectral radius for LSDA algorithm along variation of P . Reprinted with permission from [1].	23
2.2 Theoretical result of speedup for LSDA algorithm with respect to TSDA algorithm for varying cluster size N . Reprinted with permission from [1].	27
2.3 LSDA algorithm: Number of iterations (k) vs Synchronization Period (P). The number of iterations required for the TSDA algorithm to converge is constant. Reprinted with permission from [1].	30
2.4 LSDA algorithm: Computation Time vs Synchronization Period, cluster size $N = \{10, 20, 32, 40\}$. Reprinted with permission from [1].	30
2.5 LSDA: Computation Time vs Synchronization period, cluster size $N = 32$. Reprinted with permission from [1].	31
2.6 Dual variable solution (y) vs Number of iterations (k). LSDA algorithm converges to the optimal solution of the dual variable significantly faster than the TSDA algorithm. Reprinted with permission from [1].	32
2.7 TSDA algorithm: Total execution time vs Cluster size. Across various cluster sizes, it is observed that communication time is more dominating than the actual computation time. Reprinted with permission from [1].	33

2.8	LSDA algorithm: Total execution time vs Cluster size. The total execution time taken for LSDA is significantly less than for TSDA. Reprinted with permission from [1].	34
2.9	Speedup in overall execution time of LSDA algorithm with respect to TSDA algorithm for varying cluster sizes. The variation in speedup is shown with the mean value as well as the standard deviation for each cluster size. Reprinted with permission from [1].	34
3.1	Sequential training time {RIDGE, LASSO, ELASTIC} (a)-(c): vs data size, n with feature dimension, $d = \{3, 5, 7\}$, (d)-(f): vs d with $n = 24M$, (g)-(i): vs n with hyper-parameter size $ \mathbb{A} $, (j): LINREG vs n , (k): vs $ \mathbb{A} $ for 3D Road Network, (l): vs $ \mathbb{A} $ for Household Power Consumption. Cross validation folds, $ m = 3$ for synthetic datasets (a)-(j) and $ m = 2$ for real datasets (k)-(l). Reprinted with permission from [4].	52
3.2	Breakdown of DISTRIBUTED RIDGE-QR training time with zoomed insets depicting communication time (a): Stage 1: DISTRIBUTED HOUSEHOLDER-QR, (b): Stage 2: DISTRIBUTED MULTIPLY-QC and RIDGE, (c): Combined percentage. Reprinted with permission from [4].	54
3.3	Comparing scalability of various algorithms to solve RIDGE problem on synthetic datasets of size $n \times d$ (a)-(c): Various $n = \{500K, 1M, 2M\}$, $d = 100$, (d)-(f): DISTRIBUTED RIDGE-QR with $d = \{5, 10, 25, 50, 100\}$. Reprinted with permission from [4].	55
3.4	(a)-(b): Comparing DISTRIBUTED RIDGE-QR / KERNELRIDGE-QR (linear kernel), and RIDGE-ADMM for $10M \times 10$ synthetic data based on (a) Computation time (b) Accuracy, w^* is solution from scikit-learn RIDGE, (c) Accuracy of LINREG-QR and LINREG-BOOST on Household Power Consumption dataset ($\sim 2M \times 8$), w^* is solution from scikit-learn LINEARREGRESSION. Reprinted with permission from [4].	56
4.1	QRSVM technique transforms a 6×6 dense and non-separable Hessian (coefficient) matrix into a sparse block diagonal matrix, where, the first 2×2 block is full rank and the second 4×4 block is a diagonal submatrix. Dense regions are colored. The two blocks in the transformed matrix on the Right are outlined in blue. Here, $n = 6$ and $p = 2$. Reprinted with permission from [2].	65
4.2	Optimal scaling factor, P^* . Reprinted with permission from [2].	68
4.3	QRSVM framework comprises of two main stages, namely, (1) QR decomposition of the approximated input matrix $\hat{\mathbf{A}}$ that yields Householder reflectors and a matrix \mathbf{R} , and (2) Dual ascent method to solve the QRSVM problem for obtaining \mathbf{w} , which is the normal to the hyperplane (for linear SVM), and identifying set of support vectors. Reprinted with permission from [2].	70

4.4	Distributed QR decomposition: The orthogonal matrices, \mathbf{Q} , are stored as sets of their Householder reflectors, denoted as $\{\mathbf{q}\}$. Rather than sending the entire $(n/p) \times k$ sized matrix \mathbf{R}_i , we only communicate its informational content, denoted as $(\mathbf{R}_i)_{k \times k}$ i.e. the upper triangular $k \times k$ block. Thereby, we reduce the size of the matrices being communicated across the distributed network by avoiding redundant zero-blocks. At the master core, instead of creating the complete matrix with redundant zero-blocks, we <i>gather</i> the $(\mathbf{R}_i)_{k \times k}$ from all the worker nodes to generate a stacked-up representation called \mathbf{R}_{gather} . On QR decomposition, it factorizes into $\{\mathbf{q}_f\}$ and \mathbf{R}_f . We retrieve the original informational content in \mathbf{R}_g as $(\mathbf{R}_f)_{k \times k}$ and on zero-padding the $\{\mathbf{q}_f\}_i$ blocks, we generate the reflectors $\{\mathbf{q}_g\}$ for \mathbf{Q}_g . Reprinted with permission from [2].	73
4.5	Convergence of QRSVM for HIGGS dataset: (a) Using QRSVM we converge to a reasonable value of the optimal cost within 20 iterations. (b) QRSVM converges relatively faster compared to LIBLINEAR. Here, we illustrate for 250 iterations. LIBLINEAR was not able to converge to the optimal cost value in 1500 iterations while QRSVM converged to the optimum in 80 iterations. Reprinted with permission from [2].	77
4.6	Training error trend during model learning phase for the datasets. a9a training takes 166 iterations to converge and covtype takes 512 iterations. Stopping error threshold: 10^{-3} and $p = 16$. Reprinted with permission from [2].	78
4.7	Low-rank Gaussian kernel approximation results using MEKA and other methods. Reprinted with permission from [5].	79
4.8	Scalability of QRSVM (a) with sample size n : A synthetic dataset having a fixed approximated kernel rank- $k = 18$ and increasing n was used to test scalability with number of instances. (b) with rank k : A synthetic dataset with fixed number of instances, $n = 100,000$ and increasing rank was used to test scalability with rank- k (dimensionality). Reprinted with permission from [2].	80
4.9	Optimal step size: For both datasets, optimal step size is observed to be 1.9. Reprinted with permission from [2].	82
5.1	At master core: Memory improvement and representation of QR factors.	88
5.2	A two-level implementation of distributed QR decomposition of $\hat{\mathbf{A}}$. The orthogonal matrices are stored as sets of their Householder reflectors, denoted as $\{q\}$. $(\mathbf{R}_i)_{k \times k}$ are gathered from all the cores and $\hat{\mathbf{A}}_g$ is assembled at the master core. $\hat{\mathbf{A}}_g = \{\mathbf{q}_f\} \mathbf{R}_f$ is computed which can be converted to original global factors, $\{\mathbf{q}_g\}$ and \mathbf{R}_g by appending appropriate rows of zeros as depicted in Chapter 4. However, the proposed implementation here uses memory-efficient representations of the global factors for $\hat{\mathbf{A}}_g$, i.e., $\{\mathbf{q}_f\}$ and $(\mathbf{R}_f)_{k \times k}$. Reprinted with permission from [6].	89

5.3	Convergence rate of <i>distributed QRSVM</i> algorithm: Training error on benchmark datasets (covtype, webspam and SUSY) approaches the stopping threshold (10^{-3}) in $t_c = 1075$, $t_c = 569$, and $t_c = 2096$ iterations, respectively, using the optimal step size $\eta^* = 0.9$. Reprinted with permission from [6].	95
5.4	Timing (in seconds) analysis for computation and communication in different stages of distributed QRSVM. (a) Stage 1: $T_{localQR}$ and $T_{masterQR}$ are the computation time, whereas T_{gather} is the communication time. (b) Stage 2: T_{update} denotes the computation time for all iterations of parallel dual ascent, while T_{2g+2s} refers to the communication time spent in transformations $\hat{\beta}$ to β . Datasets: covtype ($p = 16$), webspam ($p = 32$), SUSY ($p = 64$). Reprinted with permission from [6].	98
5.5	Overall training time, T_{train} analysis (in seconds) for benchmarks covtype ($p = 16$), webspam ($p = 32$) and SUSY ($p = 64$). A majority of the training time is spent in iterative computations in Stage 2 (parallel dual ascent), while communication overhead is negligible. Reprinted with permission from [6].	99
6.1	Illustration of a multiple-FPGA system with $p = 4$ compute units: Here, each unit comprises an FPGA that has IP logic along with a host. Each FPGA has internal memory (BRAM) and external DRAM (DDR) memory. The communication among compute units is either through a bi-directional ring (FPGA Link) or via PCIe bus interconnection. Reprinted with permission from [3].	108
6.2	Computational flow graph for Distributed QR Decomposition. Reprinted with permission from [3].	110
6.3	Hardware kernels for (a) inner product, $\langle \vec{x}, \vec{y} \rangle$ (b) saxpy, $\vec{x} = \vec{x} + \alpha\vec{y}$. The vectorized kernels process $W = 4$ elements in each pass. Reprinted with permission from [3].	112
6.4	Data layout in column-major order and memory interface for on-chip Block RAM (Full-duplex) and off-chip DDR (Half-duplex) with the IP. Reprinted with permission from [3].	113
6.5	Computational flow graph for Parallel Dual Ascent. Reprinted with permission from [3].	115
6.6	FPGA block diagram and memory interface for a single compute unit in a multiple FPGA based system. Reprinted with permission from [3].	116
6.7	Increasing throughput of $\langle \vec{x}, \vec{y} \rangle$ by working with two copies of the inner product kernel, in parallel. Reprinted with permission from [3].	118
6.8	Strong scaling analysis: FPGA implementation achieves near linear parallel speedup for larger datasets such as Skin, Webspam and Covtype, as we double the #FPGA units (or QRSVM IP cores). For small dataset MNIST, going beyond $p = 4$ seems to be overkill. Reprinted with permission from [3].	123

6.9	Weak scaling analysis on SUSY: (a) Weak scaling efficiency decreases as T_{QR} is constant while the dominant T_{DA} is expected to increase due to more #iterations associated with growing sample size (b) Training time per iteration is constant as desired. Reprinted with permission from [3].	125
6.10	Energy consumption in FPGA under strong scaling scenario along with theoretical curves for ideal and no scalability cases. Benchmarks: Skin and Covtype. Reprinted with permission from [3].	126
6.11	(a) Energy consumption in FPGA under weak scaling scenario along with theoretical curves for ideal and no scalability cases. (b) Energy consumption per FPGA unit while maintaining equal workload on each FPGA unit. Benchmark: SUSY. Reprinted with permission from [3].	127
7.1	Various streaming setups, round $k \in [3]$ (a) Stream: only one worker, where data is generated in every round (b) Tributary: fixed number of workers, where data is generated by each worker in every round (c) Basin: dynamic number of workers, where in each round different groups of workers participate in the network with their data	139
7.2	Workflow for RIVER-Stream on a single worker	143
7.3	Workflow for RIVER-TRIBUTARY with $p = 2$ workers	144
7.4	Workflow for RIVER-BASIN with number of active workers, $p = \{2, 2, 3\}$ joining in round, $k = \{1, 2, 3\}$, respectively	144
7.5	Tributary: scaling across batch size, n (a) RIVER (b) Xy-Cumulative (c) QR-Cumulative (d) Recursive-LS	151
7.6	Basin: scaling across batch size, n (a) RIVER (b) Xy-Cumulative (c) QR-Cumulative (d) Recursive-LS	151
7.7	Tributary time, various n (a) 500×100 (b) 1000×100 (c) 2500×100 (d) 5000×100 (e) $10K \times 100$ (f) $50K \times 100$	152
7.8	Basin time (a) 500×10 (b) 1000×10 (c) 2500×10	152
7.9	Tributary: scaling across feature, d (a) RIVER (b) Xy-Cumulative (c) QR-Cumulative (d) Recursive-LS	154
7.10	Basin: scaling across feature, d (a) RIVER (b) Xy-Cumulative (c) QR-Cumulative (d) Recursive-LS	154
7.11	Tributary time, various d (a) 2500×5 (b) 2500×10 (c) 2500×50 (d) 2500×100 (e) 2500×500 (f) 2500×1000	155
7.12	Basin time, $500 \times d$ (a) 5 (b) 10 (c) 50 (d) 100	155

7.13 Tributary: scaling across workers p (a) RIVER, large n , $50K \times 100$ (b) RIVER, large d , 2500×1000	156
7.14 Tributary: scaling across workers p (a) Xy-Cumulative (b) QR-Cumulative (c) Recursive-LS	156
7.15 Tributary: model error relative to Xy-Cumulative (a) 500×100 (b) 2500×10 (c) 2500×100 (d) 2500×1000 . QR-Cumulative results are mapped to right Y-axis.....	157
7.16 Basin: model error relative to Xy-Cumulative (a) 500×10 (b) 500×100 (c) 1000×50 (d) 2500×10 . QR-Cumulative results are mapped to right Y-axis.....	158
7.17 Tributary: Timing breakdown analysis (a) RIVER (b) QR-Cumulative (c) Xy-Cumulative	159
7.18 Basin: Timing breakdown analysis (a) RIVER (b) QR-Cumulative (c) Xy-Cumulative	159

LIST OF TABLES

TABLE	Page
4.1	Distributed-QRSVM: Timing details. Reprinted with permission from [2]. 81
4.2	Distributed-QRSVM: Parameter values. Reprinted with permission from [2]. 81
5.1	Benchmark dataset description. Reprinted with permission from [6]. 93
5.2	Parameter details for various datasets. Reprinted with permission from [6]. 94
5.3	Scalability of distributed-QRSVM. Training time (in seconds) and Speedup, S_p wrt sequential-QRSVM (S_p is shown in parenthesis). Reprinted with permission from [6]. 96
5.4	Comparing <i>dis</i> -QRSVM with PSVM, P-packSVM and Chapter 4 framework on T_{train} (in seconds) for <i>covtype</i> dataset. Here, - represents data is unavailable. Reprinted with permission from [6]. 101
5.5	Comparing the improved <i>dis</i> -QRSVM with prior implementation in Chapter 4 on Stage 2 computation time, T_{update} and communication time, T_{2g+2s} (in seconds) for <i>covtype</i> dataset. Reprinted with permission from [6]. 102
6.1	Benchmark dataset description. Reprinted with permission from [3]. 119
6.2	Utilization for FPGA <i>Xilinx Virtex xcvu9p-flgb2104-2-i</i> . Reprinted with permission from [3]. 120
6.3	Training time, T_p^{FPGA} (in seconds) using p QRSVM IP cores or FPGA units. T_{QR}, T_{DA} : Compute time for QR decomposition and Dual Ascent on FPGA. Percentage of T_{FPGA} spent in computation, <i>comp</i> and communication, <i>comm</i> . SUSY is used for Weak scaling analysis while rest are used to demonstrate strong scaling analysis. Reprinted with permission from [3]. 121
6.4	Comparison with embedded edge processor (ARM Cortex A15) and cloud processor (Broadwell) platforms. In a multiple compute system, #units, p , corresponds to #FPGA units (QRSVM IP cores), #ARM processors, and #Broadwell processors. Training time (in s), T_p^{FPGA} , T_p^{ARM} , and T_p^{Broad} . Energy consumption (in kJ), E_p^{FPGA} , E_p^{ARM} , and E_p^{Broad} . Reprinted with permission from [3]. 129

1. INTRODUCTION

Machine Learning (ML) is a sub-field of Artificial Intelligence (AI) that gives the the “computers the ability to learn without being explicitly programmed” (Arthur Samuel, 1959). Infact, ML is at the heart of AI. Nowadays, it is difficult to think of an application or an industry which has not been touched by AI. “AI is the new electricity” (Andrew Ng, 2017) which has revolutionized our modern lives and more and more companies and industries are adopting it in their applications. Facebook is at the forefront of applying face recognition using DeepFace [7] to automatically tag uploaded photos to 97.47% accuracy, which is almost equal to human eyes accuracy of 97.65%. More recently, face recognition using ML has been adopted by major smart phone makers like Apple, OnePlus, Google, Samsung, etc as a key feature in identity verification services and software applications. Other ML-based applications include machine translation by Google [8], speech recognition in virtual assistants such as Google Assistant [9], Alexa by Amazon, Cortana by Microsoft, and Siri by Apple, and most recently a specific type of ML called Deep Learning (DL) is being extensively used for autonomous driving car technology such as Tesla Autopilot, Waymo, and Uber [10].

With the advent of big data, machine learning is being carried out in large-scale to extract useful knowledge and identify important patterns from tones of data. A typical ML process involves two main stages: *Training* and *Inference* as depicted in Figure 1.1. In the training stage, the data is analyzed using iterative learning algorithms until a desired accuracy is achieved on the target attribute. The output of this stage is a trained ML model that has learned to map the data attributes to the target. In the inference stage, the trained model is deployed to predict the outcome on new data samples for which the target is unavailable. In a typical ML-based application, training is a one-time process with heavy computations generally done in High Performance Computing (HPC) based cloud servers using large volumes of data while inference is used all the time in the edge devices such as smart phones, laptops, etc on single data sample or small data sets using relatively inexpensive computations.

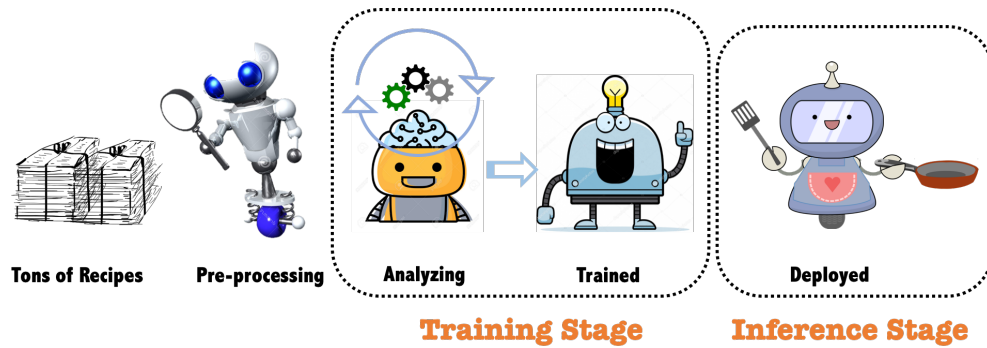
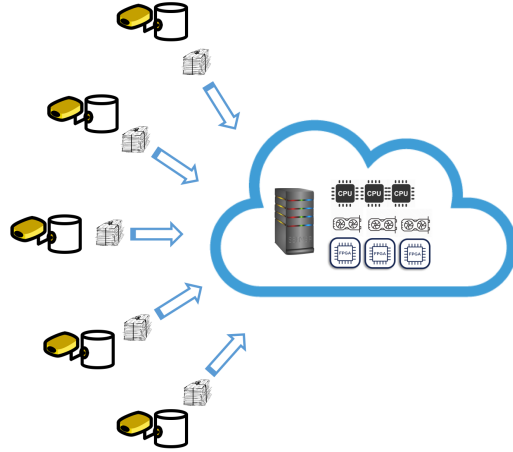
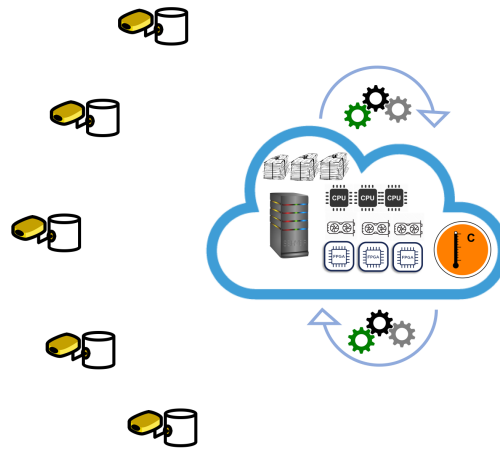


Figure 1.1: A typical large-scale machine learning process.

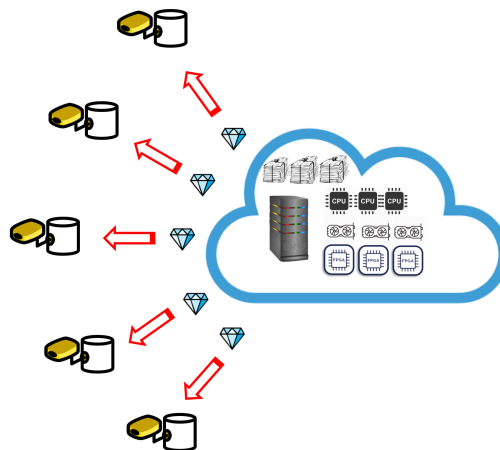
Mathematically, training a machine learning model is solving an ML optimization problem where the objective is to minimize some loss function. Such optimization problem is solved using a learning algorithm that is generally iterative and sequential. The solution of such optimization problem is a set of values that represent the parameter of the ML model. Examples of ML problems are Support Vector Machines (SVM), Kernel Ridge Regression (KRR), Deep Neural Networks (DNN) that are typically solved using iterative learning algorithms such as Interior Point Methods (IPM), Sequential Minimal Optimization (SMO), Stochastic Gradient Descent (SGD), etc. Due to the iterative and inherently sequential nature of learning algorithms running on large-scale data sets, ML training is computationally more intensive and challenging than the inference stage. To support such heavy computations and huge memory, ML training is conventionally done on HPC cloud server comprising of network of multiple high power Central Processing Units (CPUs), Graphic Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs). This leads to a centralized framework for large-scale machine learning as illustrated in Figure 1.2



(a) Data collection at the edge and transfer to HPC cloud.



(b) Centralized ML training at HPC cloud.



(c) Deploying ML model for inference at the edge.

Figure 1.2: A typical centralized machine learning framework.

Centralized ML framework comprises of three steps

1. Data is collected at source edges and transferred to an HPC cloud server where it is collectively stored.
2. ML training is performed in the cloud by running iterative learning algorithm on high-power CPU, or GPU, or FPGA etc.
3. The trained ML model is deployed back at edge devices for inference.

HPC cloud server has massive storage and is financially expensive to build and maintain. Hence, such cloud-based centralized framework can only be hosted by major technology companies with access to huge data and big financial capital. Moreover, centralizing the data in cloud puts the privacy of the user at huge risk while frequent data transfers from source to cloud over large distances lead to high-network latency. With billions of connected smart devices at the edge of Internet of Things (IoT) network with increasing computational capabilities, it has become paramount to bring intelligence on edge by processing (or training) the data closest to the source rather than on cloud.

1.1 Challenges

Here we discuss some popular training algorithms which are used to solve ML-based optimization problems in general.

1.1.1 Interior Point Method

Interior Point Method (IPM) is used to minimize the dual form of objective function in a convex quadratic programming problem. The most popular form of it is primal-dual IPM developed by [11]. The basic idea of IPM is to incorporate Newton or Quasi-Newton methods with number of iterations proportional to $\log(\frac{1}{\epsilon})$, where, ϵ is desired accuracy. However, with increasing data sizes n , IPM suffer from high memory requirement $O(n^2)$ and computational complexity of $O(n^3)$ per iteration which are prohibitive. Hence, it became imperative to parallelize IPM to solve such optimization problems faster. [12] proposed Parallel Support Vector Machines (PSVM) which was

the first attempt to parallelize SVM problem in machine learning. At the core of PSVM is a parallel implementation of IPM and Incomplete Cholesky Factorization (ICF). However, ICF is difficult to parallelize and has a sequential component that enforces the limit to extent of parallelization as sample size grows. The best reported computational complexity is $O(\frac{n^2}{p})$ per iteration and memory requirement of $O(\frac{n^{1.5}}{p})$, where, p denotes number of parallel machines.

1.1.2 Sequential Minimal Optimization

This belongs to class of decomposition algorithms where the large QP problem is broken down into series of small and manageable sub-problems that are solved sequentially until the solution of overall problem is achieved. The most popular form is Osuna decomposition [13] which had been used to solve SVM problems. Sequential Minimal Optimization (SMO) developed by [14] is a special case of Osuna algorithm where the SVM problem is decomposed into smallest possible sub-problems, of size two, which can be solved analytically without the need to construct any matrix. SMO is now widely used in popular open-source packages LIBSVM [15] and *SVM^{light}* [16]. The computational complexity of SMO is between linear and quadratic in number of samples. However, these do not fit to the distributed setting of IoT enabled devices. Attempt to parallelize SMO via parallel gradient projection method was done by [17] which repeatedly cycles through the data set, computing kernels on demand. However, the maximum workload was limited to few thousand samples which fall drastically short to modern demands.

1.1.3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an iterative method for optimizing a differentiable objective function. It is popularly used for training deep neural networks where it is coupled with forward evaluation of loss and backward propagation of error to fine tune the weight parameters of the model. It is basically a stochastic approximation of gradient descent optimization when number of training samples become huge. Rather than computing the total loss associated with each and every training sample that can be very slow for large sample size, SGD works with directly updating the weight parameter with the loss associated with a randomly selected sample in every

iteration. Minibatch SGD works by choosing a middle path between true gradient and approximated gradient computer over one sample at a time, i.e, to compute gradient on mini-batch of training samples.

1.1.4 Gaps

We observe following gaps in the current body of work which limit their applications for achieving decentralized machine learning.

- Expensive communication and synchronization overhead.
- Memory requirement is quadratic with sample size.
- Computational cost per iteration is quadratic to cubic with sample size.
- Parallel techniques are limited to smaller workload and do not scale.

1.2 Motivation

The modern society is powered by billions of IoT devices which is bringing a revolution in technological concepts such as smart homes, smart retail, smart energy, smart mobility etc. It is projected that number of connected devices is going to scale to 29 billion by 2022 [Ericsson]. These edge devices in IoT present new requirements to run AI applications that can not be met by conventional centralized ML framework. Specifically, there is a need for distributed edge intelligence for making real-time predictions with low-latency for mission critical applications. Moreover, there is a vast pool of untapped private data stored in these devices that users are not comfortable in uploading on the cloud. In fact, the available public data shared with big firms is just a tip of the iceberg while there exists huge market potential in extracting knowledge from the data stored across billions of these devices ¹. This has to be achieved by running learning algorithms on these connected devices without moving the original data thereby ensuring privacy of the user. Unlike HPC-based cloud server, these edge devices usually have relatively smaller compute capabilities with limited storage. With billions of such connected devices, it is possible to leverage their idle processing

¹https://decentralizedml.com/DML_whitepaper_31Dec_17.pdf







01	Protect Data Privacy		<ul style="list-style-type: none"> • Keep data decentralized and local on devices • Design privacy-preserving ML algorithms
02	Reduce Latency		<ul style="list-style-type: none"> • Efficient and scalable training algorithms • Cheap inference calculations to enable real-time analytics
03	Save Bandwidth		<ul style="list-style-type: none"> • Communicate less data during training • Reduce synchronizations and idling during training
04	Energy Efficiency		<ul style="list-style-type: none"> • Efficient computation and communication process • Build energy-efficient hardware accelerators for Green AI
05	Build Robust Models		<ul style="list-style-type: none"> • Devise fault-tolerance for device failures or stragglers • Accurate and robust model to data perturbations
06	Streaming Data		<ul style="list-style-type: none"> • Incremental (federated) learning to update the global model • Discard data after each update for memory and privacy

Figure 1.3: Requirements for Distributed Edge Intelligence.

power with efficient use of network bandwidth to build a decentralized computing framework for distributed machine learning and reduce the dependency on a centralized processor in cloud. To support training computations on the resource-constrained edge devices and build greener AI solutions, we need to design energy-efficient hardware accelerators to accelerate the computations with focus on energy-efficiency. Finally, with data being continuously generated and collected at edge devices, it is paramount to build robust training models that could be incrementally updated on streaming data batches without storing the previously collected data between successive streaming rounds. Such framework should incorporate fault-tolerance capabilities in the event any edge device fails or straggles during training. We summarize the requirements for distributed edge intelligence is Figure 1.3.

1.3 Dissertation Philosophy

Current trend in industry and academic research is to accelerate inference stage on edge devices while the computationally expensive training is done in cloud utilizing high performance compute capabilities of power hungry CPUs, and GPUs. In this dissertation, we integrate ideas from various fields such as distributed networks, parallel algorithms, and computer hardware as illustrated in

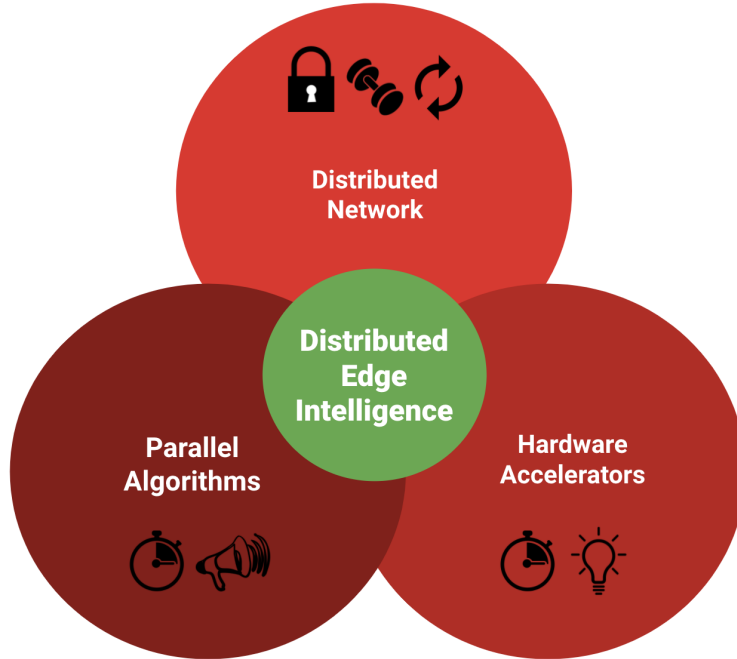


Figure 1.4: Research Focus for efficient and scalable machine learning for distributed edge intelligence.

Figure 1.4 to design *Efficient and Scalable Machine Learning for Distributed Edge Intelligence*.

We propose to decentralize machine learning by bringing training on edge devices and unlocking the potential of untapped private data, and to utilize idle processing power of edge devices connected over a distributed network. Some plausible decentralized ML frameworks are illustrated in Figures 1.5, 1.6 and 1.7. A decentralized ML framework should comprise following steps.

1. Data is collected and stored at the respective source edges.
2. ML training is performed in parallel across multiple edge devices (with a possibility of communication via edge server or cloud).
3. The trained model is simultaneously deployed at edge devices for inference.

We design *distributed training algorithms* that are computationally fast, scalable, memory-efficient, have low communication overhead, robust and handle streaming data. Moreover, such algorithms will be well suited for running in parallel on a network of small low-powered devices

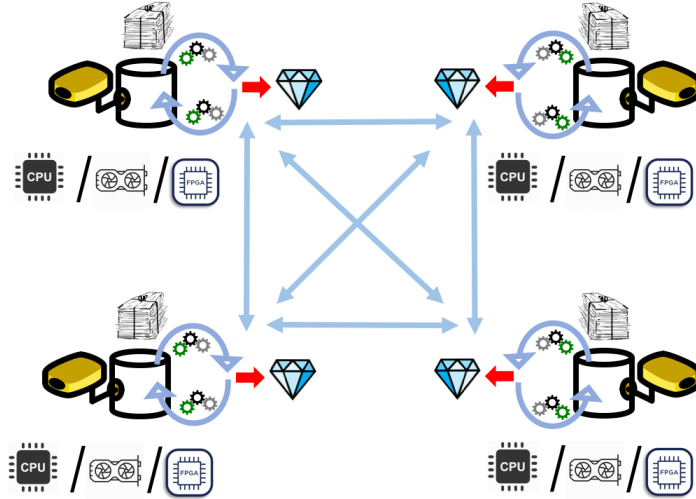


Figure 1.5: Decentralized network where all edge devices are connected to each other.

with high energy-efficiency and prediction accuracy.

1.4 Contributions

The goal of this dissertation is to develop decentralized machine learning framework by addressing the gaps in current solvers listed in Section 1.1.4 while satisfying the requirements for distributed edge intelligence listed in Figure 1.3. The key contributions are summarized as follows

1. We propose a relaxed synchronization training approach with an analytically derived optimal synchronization time period to reduce communication frequency and idling in iterative solver for parallel quadratic programming problems. We show analytically and experimentally that lazy synchronization is numerically stable and converges to the same result as the conventional tightly synchronized implementation. Furthermore, the convergence speed of the proposed algorithm is faster with lazy synchronization. The proposed algorithm is implemented in a 40-node distributed system in the Amazon Elastic Computing infrastructure. We show a $160\times$ speedup in solution time for a large-scale quadratic programming problem. We empirically demonstrate that the relaxed synchronization technique reduces communication overhead by 99.65% in comparison to the tightly synchronization implementation.

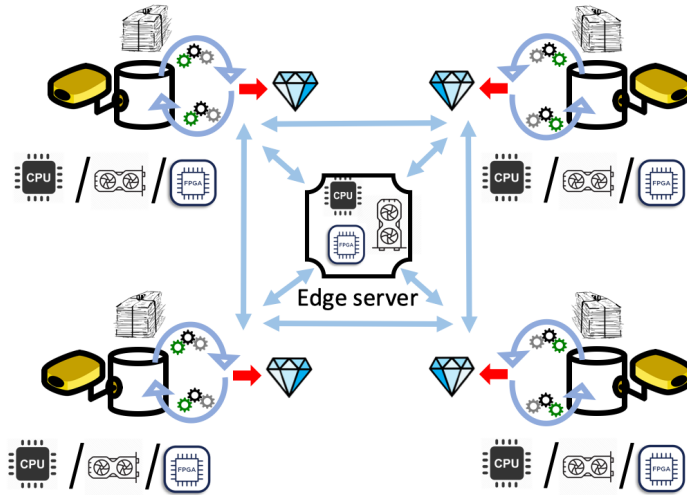


Figure 1.6: Decentralized network where all edge devices are connected to each other and an Edge server.

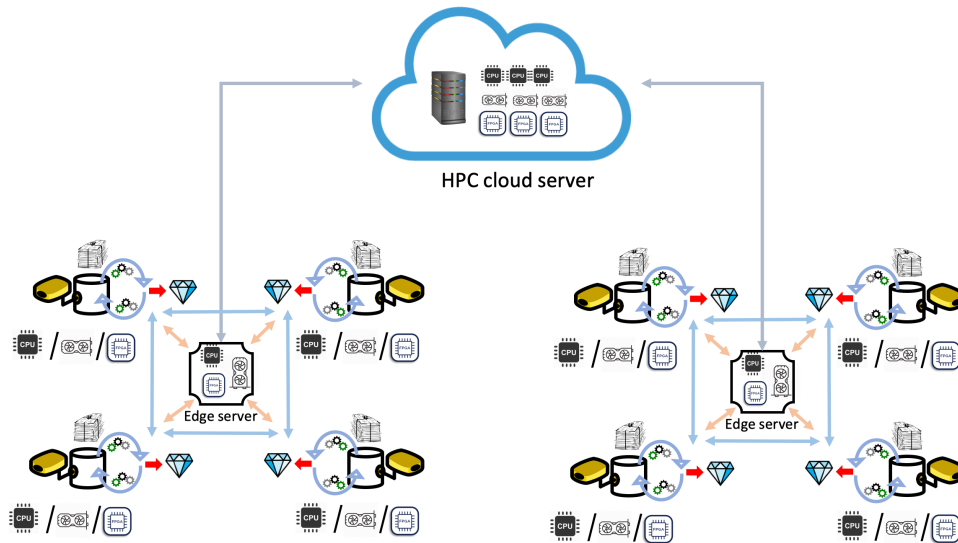


Figure 1.7: A typical IoT based network where multiple edge servers may communicate via cloud while major computations are handled by edge devices.

2. We advocate for distributed Householder sketches as fast and accurate machine learning to parallelize summary generation across workers and apply it for training machine learning model globally. We find it to be a simpler, memory-efficient, and faster alternative that

always existed to the strong baselines. We also present a scalable algorithm based on Householder sketches to solve Least-Mean-Squares problem across multiple worker nodes under a distributed setting. We perform thorough empirical analysis with large synthetic and real datasets to evaluate the performance of Householder sketch and compare with the strong baseline solver. Our results show Householder sketch speeds up existing LMS solvers in the scikit-learn library up to 100x-400x. Also, it is 10x-100x faster than the above strong baseline with similar numerical stability. Our results for distributed implementation show a near-negligible communication overhead with linear scalability.

3. We leverage the distributed summary generation and relaxed synchronization to parallelize convex machine learning optimization problem by building a memory-efficient distributed machine learning framework using proposed iterative solvers such as parallel dual ascent across multiple worker nodes. We present a novel QR decomposition framework (QRSVM) to efficiently model and solve a large scale SVM problem by capitalizing on low-rank representations of the full kernel matrix rather than solving the problem as a sequence of smaller sub-problems. The low-rank structure of the kernel matrix is leveraged to transform the dense matrix into one with a sparse and separable structure. The modified SVM problem requires significantly lesser memory and computation. We also derive an optimal step size for fast convergence of the dual ascent method for training the model.
4. We develop a communication-efficient implementation of the above machine learning framework with negligible communication overhead to scale model training on large-scale datasets and large workers. Experiments on benchmark data sets with up to five million samples demonstrate negligible communication overhead and linear scalability. Execution times are vast improvements over other widely used packages. Furthermore, the proposed algorithm has linear time complexity with respect to the number of samples making it ideal for training on decentralized environments such as smart embedded systems and edge-based IoT.
5. To enable energy-efficient machine learning for real-time applications on edge, we propose

and build a first-of-its-kind multiple FPGA software/hardware accelerator system to train distributed machine learning. Each FPGA unit has a pipelined model training IP logic core operating at 125 MHz with a power dissipation of 39 Watts for accelerating its allocated computations. We evaluate and compare the performance of the proposed system on five real benchmarks. The proposed FPGA co-designed accelerator system is around 3x to 24x faster than the embedded edge processor (ARM), and around 1.7x faster than the cloud processor (Broadwell). For large datasets, the proposed system achieves 2x to 8x lower energy consumption compared to the ARM processor, and 6.5x lower than Broadwell processor.

6. To handle streaming data batches at the edge, we build a series of rapid incremental solver schemes for machine learning by integrating incremental learning on federated setups to efficiently and accurately update the models while making it robust and fault-tolerant to straggling workers or device failures.

1.5 Organization

The rest of the dissertation is organized as follows. Chapter 2 presents the relaxed synchronization technique for parallel QP problems by communicating less frequently. Chapter 3 describes the construction of distributed Householder sketches as accurate summaries to efficiently train the global machine learning model. Chapter 4 proposes memory-efficient distributed machine learning framework with parallel dual ascent to reduce latency and effectively train model on multiple workers. Chapter 5 illustrates the improved communication-efficient framework for scalable machine learning. Chapter 6 presents an energy-efficient system of multiple FPGA accelerators for accelerating distributed training on decentralized edge. Chapter 7 presents a rapid incremental solver for federated machine learning to effectively handle streaming data and build robust models. Finally, Chapter 8 concludes the dissertation and presents future research directions.

2. RELAXED SYNCHRONIZATION FOR PARALLEL QUADRATIC PROGRAMMING ¹

In this chapter, we present a novel numerical algorithm for efficiently solving large-scale quadratic programming problems in massively parallel computing systems. The main challenge in maximizing processor utilization is to reduce idling due to synchronization across processors. Typically, synchronization is necessary after every iteration, which prevents many numerical algorithms from scaling with number of processors. We relax this requirement by synchronizing at a lower rate, which is referred to as *lazy synchronization*.

2.1 Introduction

Currently we are witnessing a rise in large-scale data analytics fueled by the availability of data and advancement in computing systems. Large-scale optimization is the backbone of almost all statistical and machine learning algorithms. These algorithms are routinely used to develop data-driven solutions for various industrial sectors including financial, energy, aerospace, biomedical, etc; targeting important problems related to resource allocation, operations research, system diagnostics and prognostics, advertising, and business intelligence. Near real-time solution of these problems is becoming increasingly important for competitive advantage and is pushing implementation in large-scale distributed machines. In addition to fast solutions, the size of the data is another factor for distributed implementations, as all the data cannot be processed in a monolithic formulation. There is a large body of literature on decomposition methods and decentralized algorithms in the optimization community, which naturally leads to parallel optimization algorithms as a mechanism for solving large-scale problems. Recently, there is a renewed interest in *proximal methods* [18] in various forms such as alternating direction method of multipliers (ADMM), Douglas-Rachford operator splitting, Spingarn’s method of partial inverses, Dykstra’s alternating projections method, Bregman iterative algorithms for l_1 problems in signal processing, and many

¹This chapter is reprinted with permission from “A Relaxed Synchronization Approach for Solving Parallel Quadratic Programming Problems with Guaranteed Convergence” by Kooktae Lee, Raktim Bhattacharya, Jyotikrishna Dass, V. N. S. Prithvi Sakuru, and Rabi N. Mahapatra, 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Copyright ©2016 IEEE.

more. These methods are simple but powerful and are well suited for distributed convex optimization. They take the form of a *decomposition-coordination* procedure, in which solutions to small local subproblems are coordinated to find a solution to a large global problem, and combine the benefits of dual decomposition and augmented Lagrangian methods for constrained optimization problems. The communication bandwidth is determined from the size of the local subproblems. While the proximal methods admit robust decomposition and can be used to parallelize optimization problems, they are based on synchronous model of computation, which will not be feasible for future computing systems.

2.1.1 Motivation

It is clear that future computing systems are expected to be exascale with large number of processors [19], providing a deep hierarchy of systems and resources, such as base multicore processors composed into symmetric multiprocessors which may in turn be composed into distributed heterogeneous systems [20, 21, 22]. Furthermore, such systems are likely to be heterogeneous using both heavily multi-threaded CPUs as well as GPUs. Among the main obstacle to scale algorithms to exascale levels, is the synchronization necessary in tightly coupled problems. Proximal operator based parallelized optimization algorithms will require synchronization across the distributed nodes in a cluster framework, which will severely impede the computational performance. The idling time when million cores need to be synchronized after each time step, to evaluate gradients as well as adjust step sizes for numerical stability reasons, is a major obstacle in achieving high processor utilization. This has resulted in new algorithms that adopt asynchronous model communication where the computing nodes do not wait for updates from other nodes, and proceed with the latest information in the buffer. There are many flavors of asynchronous algorithms reported in the literature [23, 24, 25, 26, 27, 28, 29, 30].

2.1.2 Contributions

Asynchrony is not the only way to relax the communication bottleneck across processing elements. Instead of synchronizing between processing elements after every iteration (i.e. *tightly*

synchronized), the communication bottleneck can be significantly avoided if synchronization is done at a lower rate (i.e. *lazily synchronized*). In this chapter, we explore this approach for solving distributed optimization problems, with focus on solving quadratic programming problems using dual-ascent approach. Since the updates are synchronized, the proposed algorithm is deterministic. The dual-ascent based algorithm admits a broadcast-gather architecture, where subproblems are solved in multiple nodes/cores and their results are gathered in a master node that couples them before the next update. The ADMM approach also has the same broadcast-gather architecture and the framework presented here can be extended to develop lazily synchronized ADMM algorithms as well.

Main contributions of this work are the following:

1. Analytical proofs for convergence and stability of the proposed *lazily synchronized dual-ascent* (LSDA) algorithm are presented in discrete-time dynamical systems framework.
2. Analytical derivation of the optimal synchronization rate is performed for which the algorithm converges to the optimal solution.
3. Experimental validation of theoretical results is conducted in a multi-node distributed framework. In our case study, we observe that the LSDA algorithm is significantly faster with a speedup of $\approx 160\times$ in comparison with the *tightly synchronized dual ascent* (TSDA) algorithm. For the problem considered here, the minimum number of update steps required for guaranteed convergence of LSDA is $70\times$ lower than of TSDA. Consequently, a 99.65% reduction in communication delay was achieved that led to $\approx 160\times$ speedup in solution time.

2.2 Related Work

Here we present related works to mitigate the synchronization penalty. Early work on asynchronous iterations was by Bertsekas and Tsitsiklis, who introduced a sufficient condition for the convergence of general asynchronous fixed-point iterations (see Chapter 6.2 in [25]), which is equivalent to a diagonal dominance condition. Nedic et.al. [26] studied distributed asynchronous subgradient method for minimizing convex function that is separable. In this research, the authors

proposed distributed computation to be carried out along subgradient iteration incrementally and asynchronously. In [28], Wei and Ozdaglar developed asynchronous ADMM based algorithm for a convex optimization problem that has separable objective function with linear constraints. The proposed algorithm showed almost sure convergence to an optimal solution with the rate of convergence $O(1/k)$. Liu et.al. [29] investigated asynchronous parallel stochastic coordinate descent algorithm to minimize constrained functions that can be expressed as summation of component-wise functions. In [29], the proposed method attains a linear convergence rate for the functions with strong convexity property. Recently Zhang et.al. [30] have presented asynchronous ADMM algorithm for consensus optimization. In their work, convergence with asynchronous updates in the mean value of the state variable. However, convergence in the mean can be achieved even in the case when the variance of the variable diverges. Also, their algorithm relies on a minimum number of updates from the distributed nodes. Asynchronous algorithms, while being resource optimal, have poor predictability and the results become stochastic. The algorithms in the literature do not come with rigorous analysis for convergence rate and stability. Consequently, the theoretical results are hard to verify experimentally. In this chapter, we provide a new algorithm to overcome the communication bottleneck via lazy synchronization across nodes. We provide theoretical analysis for stability, convergence rate and optimal synchronization period. These theoretical results are also experimentally validated in multi-node distributed system.

2.3 Problem Formulation

In this section, we introduce the preliminaries of quadratic programming problems and convex optimization problems in general. We first define the notation. The set of real number, positive integer, and the non-negative integer are denoted by the symbol \mathbb{R} , \mathbb{N} , and \mathbb{N}_0 , respectively. The superscript symbol T represents the transpose operator for both vectors and matrices. In addition, the symbol $\rho(\cdot)$ denotes the spectral radius of the square matrix (i.e., the largest one among the magnitude of eigenvalues for the given square matrix). Finally, for any real number a and any real matrix \mathbf{A} , the symbol $|a|$ and $|\mathbf{A}|$ represent the absolute value of a and the matrix with absolute value of all elements in \mathbf{A} , respectively.

The quadratic programming problem considered here is

$$\min_{\mathbf{x}} f(\mathbf{x}) \text{ subject to } \mathbf{Ax} = \mathbf{b}, \quad (2.1)$$

where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^m$, and $m, n \in \mathbb{N}$

$$f(\mathbf{x}) := \frac{1}{2} \mathbf{x}^T \mathbf{Q} \mathbf{x} + \mathbf{c}^T \mathbf{x},$$

$\mathbf{Q} \in \mathbb{R}^{n \times n}$ is a symmetric, positive definite matrix, and $\mathbf{c} \in \mathbb{R}^n$.

The corresponding Lagrangian is defined as

$$L(\mathbf{x}, \mathbf{y}) := f(\mathbf{x}) + \mathbf{y}^T (\mathbf{Ax} - \mathbf{b}),$$

where $\mathbf{y} \in \mathbb{R}^m$ is the vector of dual variable or the Lagrange multiplier.

We assume that the cost function $f(\mathbf{x})$ is separable, i.e.

$$f(\mathbf{x}) := \sum_{i=1}^N f_i(\mathbf{x}_i), \quad \mathbf{Ax} := \sum_{i=1}^N \mathbf{A}_i \mathbf{x}_i,$$

where scalar $N > 0$ denotes the total number of subproblems.

In the case that N number of distributed nodes exist, each subproblem can be solved in parallel by each node. For the implementation of parallel computing, we consider the parallel dual-ascent method that consists of iterating the updates

$$\mathbf{x}_i^{k+1} = \arg \min_{\mathbf{x}_i} L_i(\mathbf{x}_i, \mathbf{y}^k) = -\mathbf{Q}_i^{-1} (\mathbf{A}_i^T \mathbf{y}^k + \mathbf{c}_i) \quad (2.2)$$

$$i = 1, \dots, N,$$

$$\mathbf{y}^{k+1} = \mathbf{y}^k + \alpha^k (\mathbf{Ax}^{k+1} - \mathbf{b}), \quad (2.3)$$

where scalar $\alpha^k > 0$ is the step size, the superscript k is the iteration counter, and \mathbf{x}_i are partitions

of \mathbf{x} . The constraint $\mathbf{Ax} = \mathbf{b}$ can be partitioned with respect to \mathbf{x}_i , and the Lagrangian L_i can be appropriately defined. For simplicity, α^k is considered to be constant, and thus does not change over k .

A key observation here is that Equations (2.2) and (2.3) define a discrete-time dynamical system in terms of \mathbf{x} and \mathbf{y} . Thus, convergence of the iteration can be treated as stability of the dynamical system. This will be critical in analyzing the asynchronous version of the algorithm. The idea of treating iterative numerical algorithms as discrete-time dynamical systems has been considered by the first author in other applications [23] and others [31, 32].

In a distributed implementation, Equation (2.2) is parallelized over N nodes and the updated \mathbf{x}_i^{k+1} values are broadcasted to a master node, which updates \mathbf{y} using Equation (2.3). In a synchronous implementation, the master node waits for the N nodes to complete their computations and broadcast their results. This process occurs at every iteration, since \mathbf{y} value is updated using \mathbf{x} , which requires synchronization. Thus, this synchronization latency may cause serious bottleneck in parallel computation. As N increases, the idle time can be as large as 50% of the total computational time as reported in [33].

2.4 Lazily Synchronized Dual Ascent (LSDA) Algorithm

In this section, we present a novel lazy synchronization algorithm to address concerns of tight synchronization and reduce idling time. Specifically, we propose lazily synchronized dual-ascent (LSDA) algorithm with theoretical guarantees for stability and convergence to the optimal solution. We also present a theorem to determine optimal synchronization period that guarantees the fastest convergence of a given quadratic programming problem. Finally, we present the pseudocode to implement LSDA and discuss its theoretical speedup with respect to tightly synchronized dual-ascent (TSDA) algorithm.

2.4.1 Theory

From Equations (2.2) and (2.3), the dynamics of dual variable vector \mathbf{y} for the distributed QP problems can be written as follows:

$$\mathbf{y}^{k+1} = \mathbf{y}^k + \sum_{i=1}^N \alpha_i \left(\mathbf{A}_i \mathbf{x}_i^{k+1} - \frac{\mathbf{b}}{N} \right). \quad (2.4)$$

Instead of synchronizing the data for \mathbf{x} at every iteration, which results in computational inefficiency, we relax the synchronization penalty. In this case, the data for \mathbf{x} is synchronized at a certain scalar time period P that is greater than unit iteration step. The \mathbf{y} -update is then described as follows:

$$\mathbf{y}^{k+1} = \mathbf{y}^k + \sum_{i=1}^N \alpha_i \left(\mathbf{A}_i \mathbf{x}_i^{tP+1} - \frac{\mathbf{b}}{N} \right), \quad (2.5)$$

$$tP \leq k < (t+1)P,$$

where $t \in \mathbb{N}_0$.

Note that in above relaxed synchronization algorithm, \mathbf{x}_i values are fixed as constant during the period $tP \leq k < (t+1)P$. While the time k belongs to the given period, \mathbf{y} -update is carried out internally without communication, by using lastly updated \mathbf{x}_i values, i.e., \mathbf{x}_i^{tP+1} . Once the time k reaches the next period $(t+1)P$, the communication occurs to broadcast each \mathbf{x}_i value to the master node and thus \mathbf{y} value is updated using newly broadcasted \mathbf{x}_i . In this way, \mathbf{y} is updated while communicating intermittently.

Substituting \mathbf{x}_i^{tP+1} in Equation (2.5) with $\mathbf{x}_i^{tP+1} = -\mathbf{Q}_i^{-1}(\mathbf{A}_i^T \mathbf{y}^{tP} + \mathbf{c}_i)$ given in Equation (2.2), it follows

$$\mathbf{y}^{k+1} = \mathbf{y}^k + \sum_{i=1}^N \alpha_i \left(-\mathbf{A}_i \mathbf{Q}_i^{-1} (\mathbf{A}_i \mathbf{y}^{tP} + \mathbf{c}_i) - \frac{\mathbf{b}}{N} \right), \quad (2.6)$$

$$tP \leq k < (t+1)P.$$

At the end of the given period $tP \leq k < (t+1)P$, i.e., $k = (t+1)P - 1$, \mathbf{y} -update is derived from Equation (2.6) by

$$\begin{aligned}
\mathbf{y}^{(t+1)P} &= \mathbf{y}^{tP} + P \sum_{i=1}^N \alpha_i \left(-\mathbf{A}_i \mathbf{Q}_i^{-1} (\mathbf{A}_i \mathbf{y}^{tP} + \mathbf{c}_i) - \frac{\mathbf{b}}{N} \right) \\
&= \mathbf{y}^{tP} - P \sum_{i=1}^N \alpha_i (\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{A}_i^T) \mathbf{y}^{tP} - P \sum_{i=1}^N \alpha_i \left(\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{c}_i + \frac{\mathbf{b}}{N} \right) \\
&= \left(\mathbf{I} - P \sum_{i=1}^N \alpha_i (\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{A}_i^T) \right) \mathbf{y}^{tP} - P \sum_{i=1}^N \alpha_i \left(\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{c}_i + \frac{\mathbf{b}}{N} \right), \quad (2.7)
\end{aligned}$$

where \mathbf{I} stands for the identity matrix with a proper dimension. The above equation is valid due to the linear property with respect to the variable \mathbf{y} .

The stability of the QP problems with LSDA algorithm depends on the parameter P . As P increases, the communication for synchronization happens less frequently. However, one may not increase P as large as possible because the dual variable \mathbf{y} may diverge. Thus, the condition in the following lemma has to be satisfied for the numerical stability of LSDA algorithm.

Lemma 2.1. (Stability) *The dual variable for LSDA algorithm is stable if and only if*

$$\rho \left(\mathbf{I} - P \sum_{i=1}^N \alpha_i (\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{A}_i^T) \right) < 1. \quad (2.8)$$

The above condition guarantees the stability of LSDA algorithm described by Equation (2.7), since it is known that for the given discrete-time linear dynamics $\mathbf{y}^{k+1} = \mathbf{A} \mathbf{y}^k + \mathbf{b}$, where \mathbf{A} is a time-invariant matrix and \mathbf{b} is a constant vector, \mathbf{y} will not diverge to infinity as $k \rightarrow \infty$ if and only if $\rho(\mathbf{A}) < 1$. Thus, Condition (2.8) ensures the stability of dual variable \mathbf{y} in Equation (2.7).

Next, we prove that the solution of LSDA algorithm is convergent to that of TSDA algorithm in the following proposition.

Proposition 2.1. (Convergence) *Consider the QP problem that is separable. If the Condition (2.8) holds, then the dual variables \mathbf{y}_{LSDA} for LSDA and \mathbf{y}_{TSDA} for TSDA converge to the same fixed-point value $\mathbf{y}^* := \lim_{k \rightarrow \infty} \mathbf{y}_{TSDA}^k = \lim_{t \rightarrow \infty} \mathbf{y}_{LSDA}^{tP}$.*

Proof. We start from Equation (2.7), which corresponds to the dynamics of \mathbf{y}_{LSDA} . Particularly when $P = 1$, Equation (2.7) can be also used to denote the dynamics of \mathbf{y}_{TSDA} . Then, it follows

$$\begin{aligned} \mathbf{y}^* &= \lim_{t \rightarrow \infty} \mathbf{y}^{(t+1)P} \\ &= \left(\mathbf{I} - P \sum_{i=1}^N \alpha_i (\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{A}_i^T) \right) \mathbf{y}^* - P \sum_{i=1}^N \alpha_i \left(\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{c}_i + \frac{\mathbf{b}}{N} \right), \end{aligned}$$

resulting in

$$P \left(\sum_{i=1}^N \alpha_i (\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{A}_i^T) \right) \mathbf{y}^* = -P \sum_{i=1}^N \alpha_i \left(\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{c}_i + \frac{\mathbf{b}}{N} \right).$$

In above equation, P can be canceled out, since $P \geq 1$ as the synchronization period. Thus, it leads to

$$\mathbf{y}^* = - \left(\sum_{i=1}^N \alpha_i (\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{A}_i^T) \right)^{-1} \sum_{i=1}^N \alpha_i \left(\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{c}_i + \frac{\mathbf{b}}{N} \right),$$

where $\left(\sum_{i=1}^N \alpha_i (\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{A}_i^T) \right)$ has to be non-singular.

Consequently, both \mathbf{y}_{TSDA} and \mathbf{y}_{LSDA} converge to \mathbf{y}^* regardless of P , if the given scheme is stable, i.e., Condition (2.8) is satisfied. \square

Although the analytic solution for \mathbf{y}^* is given above, direct computation of the matrix $\left(\sum_{i=1}^N \alpha_i (\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{A}_i^T) \right)^{-1}$ may not be feasible, if the matrix $\left(\sum_{i=1}^N \alpha_i (\mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{A}_i^T) \right)$ is ill-conditioned or inverse operation is computationally expensive. Thus, we consider the cases in which \mathbf{y}^* has to be calculated iteratively.

2.4.2 Optimal Synchronization Period

The remaining issue is to design an appropriate value of synchronization period P because the convergence speed of LSDA algorithm strictly depends on P . Hence, we need to find the optimal P , which will result in the fastest convergence of dual variable. To determine the optimal P , denoted by P^* , we develop the following main result.

Theorem 2.1. (Optimality) For the given parallel QP problem with LSDA technique, the optimal synchronization period P^* is obtained by

$$P^* = \max_{P \in \mathbb{N}} \arg \min_{P \in \mathbb{N}} \max\{|1 - \underline{\lambda}(\boldsymbol{\beta})P|, |1 - \bar{\lambda}(\boldsymbol{\beta})P|\} \quad (2.9)$$

where $\boldsymbol{\beta} := \sum_{i=1}^N \alpha_i \mathbf{A}_i \mathbf{Q}_i^{-1} \mathbf{A}_i^T$, $\underline{\lambda}(\cdot)$ and $\bar{\lambda}(\cdot)$ denote the smallest and the largest eigenvalues of the square matrix, respectively.

Proof. The convergence speed of LSDA algorithm is solely determined by the spectral radius $\rho(\mathbf{I} - \boldsymbol{\beta}P)$, where $\boldsymbol{\beta}$ is defined in Equation (2.9). As this spectral radius becomes smaller, \mathbf{y}_{TSDA} converges to \mathbf{y}^* faster. Thus, we aim to find P^* that minimizes $\rho(\mathbf{I} - \boldsymbol{\beta}P)$.

By the definition of the spectral radius, we have

$$P^* = \arg \min_{P \in \mathbb{N}} \rho(\mathbf{I} - \boldsymbol{\beta}P) = \arg \min_{P \in \mathbb{N}} \max_{\mathbf{v}} \left| \frac{\mathbf{v}^T (\mathbf{I} - \boldsymbol{\beta}P) \mathbf{v}}{\|\mathbf{v}\|^2} \right| = \arg \min_{P \in \mathbb{N}} \max_{\mathbf{v}} \left| 1 - \frac{\mathbf{v}^T \boldsymbol{\beta} \mathbf{v}}{\|\mathbf{v}\|^2} P \right|,$$

where \mathbf{v} is the eigenvector of the matrix $(\mathbf{I} - \boldsymbol{\beta}P)$.

From the given structure of $\boldsymbol{\beta}$, it is known that $\boldsymbol{\beta}$ is symmetric, positive (semi)definite matrix. Therefore, it satisfies

$$\underline{\lambda}(\boldsymbol{\beta}) \leq \frac{\mathbf{v}^T \boldsymbol{\beta} \mathbf{v}}{\|\mathbf{v}\|^2} \leq \bar{\lambda}(\boldsymbol{\beta}).$$

Thus, the value $\frac{\mathbf{v}^T \boldsymbol{\beta} \mathbf{v}}{\|\mathbf{v}\|^2}$ lies between $\underline{\lambda}(\boldsymbol{\beta})$ and $\bar{\lambda}(\boldsymbol{\beta})$, resulting in

$$P^* = \arg \min_{P \in \mathbb{N}} \max\{|1 - \underline{\lambda}(\boldsymbol{\beta})P|, |1 - \bar{\lambda}(\boldsymbol{\beta})P|\}.$$

Finally, it is not guaranteed whether such P^* satisfying the above condition is unique. Hence, we may have multiple P^* from the above equation. Among these values, the largest P^* will be chosen as the optimal value owing to the fact that larger P^* implies less communication between distributed nodes and master node. As a consequence, Equation (2.9) is obtained for the optimal

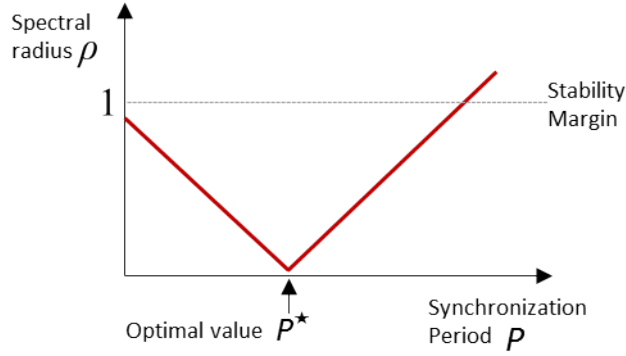


Figure 2.1: An illustration of the spectral radius for LSDA algorithm along variation of P . Reprinted with permission from [1].

synchronization period. □

Conceptually, P^* is the solution to minimize the spectral radius for the fastest convergence of y with less iterations. We illustrate this spectral radius along variation of P in Figure 2.1. As P increases, the spectral radius will decrease. However, once it becomes $P > P^*$, the spectral radius will increase, resulting in this value being greater than one, which is marginal stability bound. Such P^* can be analytically calculated by Equation (2.9) without increasing P gradually, which is computationally inefficient and expensive as well.

2.4.3 Implementation

Algorithm 2.1 presents an implementation of the proposed LSDA algorithm. The basic synchronization step used in the implementation is *Allreduce* (Step 16). This reduction operation uses the *SUM* operator to combine the `sumLocal` value (Steps 12-15) from each of the nodes to compute the `sumGlobal` value. *Allreduce* is used as a blocking operation which when invoked in any of the nodes, halts the further execution in that node until the ongoing `sumLocal` calculation is finished for remaining nodes and the newly computed `sumGlobal` value is updated for all the nodes. In the proposed LSDA algorithm, the synchronization (or communication across the nodes) occurs once in every P iterations (Steps 11 – 17) compared to once in each iteration for the TSDA algorithm. In-between the synchronization time instants, the dual variable updates itself with a

Algorithm 2.1 LSDA($\mathbf{Q}_i, \mathbf{A}_i, \mathbf{c}_i, P^*$)

Data: For each node i , local matrix \mathbf{Q}_i , local constraint matrix \mathbf{A}_i , local vector \mathbf{c}_i , optimal synchronization period, P^*

Result: A vector of dual variable, $\mathbf{y} \in \mathbb{R}^m$

```
1  $P \leftarrow P^*$ 
2  $\mathbf{b} \leftarrow \mathbf{b}_0$ 
3  $\mathbf{y}_{next} \leftarrow \mathbf{y}_0$ 
4  $\epsilon \leftarrow \epsilon_0$ 
5  $\alpha \leftarrow \alpha_0$ 
6  $k \leftarrow 0$ 
7 sumLocal  $\leftarrow 0$ 
8 sumGlobal  $\leftarrow 0$ 
9 do
10    $\mathbf{y}_{current} \leftarrow \mathbf{y}_{next}$ 
11   if  $iter \% P == 0$  then
12     for  $j \leftarrow 1 \dots (n/N)$  do
13        $\mathbf{x}_j = -\mathbf{Q}_j^{-1} (\mathbf{A}_j^T \mathbf{y}_{current} + \mathbf{c}_j)$ 
14       sumLocal  $+= \mathbf{A}_j \mathbf{x}_j$ 
15     end
16     Allreduce(sumLocal, sumGlobal)
17   end
18    $\mathbf{y}_{next} \leftarrow \mathbf{y}_{current} + \alpha(\mathbf{sumGlobal} - \mathbf{b})$ 
19    $error \leftarrow |\mathbf{y}_{next} - \mathbf{y}_{current}|$ 
20    $iter ++$ 
21 while  $error > \epsilon$ 
22 return  $\mathbf{y}$ 
```

fixed error value of $\alpha(\mathbf{sumGlobal} - \mathbf{b})$ (Step 18). This is due to **sumGlobal** value being held as constant owing to absence of synchronization among the nodes during that interval. For these iterations, the error value computed in Step 19 always satisfies the stopping threshold (ϵ) condition in Step 21 and proceeds to the next iteration. Hence, the algorithm will only terminate at some synchronization time instant $k = tP$, where $t \in \mathbb{N}_0$, when the inter-node communication results in a newly calculated **sumGlobal** value, thereby causing the error value to fall below the stopping threshold.

2.4.4 Theoretical Speedup

For the calculation of the speedup, we present the symbols with definition as follows:

- k_{TSDA}^* : the total number of iterations up to termination for TSDA algorithm
- k_{LSDA}^* : the total number of iterations up to termination for LSDA algorithm
- t_{TSDA}^p : computation time per iteration for TSDA algorithm
- t_{LSDA}^p : computation time per iteration for LSDA algorithm
- t_{TSDA}^m : pure communication time per iteration for TSDA algorithm
- t_{LSDA}^m : pure communication time per iteration for LSDA algorithm
- T_{TSDA} : total execution time for TSDA algorithm
- T_{LSDA} : total execution time for LSDA algorithm

Then, the total execution time for each case is calculated by

$$T_{\text{TSDA}} = k_{\text{TSDA}}^* (t_{\text{TSDA}}^p + t_{\text{TSDA}}^m),$$

$$T_{\text{LSDA}} = k_{\text{LSDA}}^* \left(t_{\text{LSDA}}^p + \frac{t_{\text{LSDA}}^m}{P^*} \right).$$

where, t_{LSDA}^p comprises of time taken to compute Steps 12 – 15 (once in every P^* iterations) and Steps 18 – 20 (in every iteration), i.e.

$$t_{\text{LSDA}}^p = \frac{t_{\text{LSDA}}^{14-17}}{P^*} + t_{\text{LSDA}}^{20-22}$$

The reason for dividing t_{LSDA}^m by P^* in T_{LSDA} formulation above is that the communication in LSDA occurs only $\frac{k_{\text{LSDA}}^*}{P^*}$ times during the entire convergence process, while the computation takes place at every iteration step. Thus, the expected speedup is obtained by

$$\text{SPEEDUP} := \frac{T_{\text{TSDA}}}{T_{\text{LSDA}}} = \frac{k_{\text{TSDA}}^* (t_{\text{TSDA}}^p + t_{\text{TSDA}}^m)}{k_{\text{LSDA}}^* \left(t_{\text{LSDA}}^p + \frac{t_{\text{LSDA}}^m}{P^*} \right)}. \quad (2.10)$$

With the definition of symbols $r_k := \frac{k_{\text{TSDA}}^*}{k_{\text{LSDA}}^*}$, $r_p := \frac{t_{\text{TSDA}}^p}{t_{\text{LSDA}}^p}$, $r_{\text{TSDA}} := \frac{t_{\text{TSDA}}^m}{t_{\text{TSDA}}^p}$, and $r_{\text{LSDA}} := \frac{t_{\text{LSDA}}^m}{t_{\text{LSDA}}^p}$, Equation (2.10) is written as

$$\text{SPEEDUP} = r_k r_p \left(\frac{1 + r_{\text{TSDA}}}{1 + \frac{r_{\text{LSDA}}}{P^*}} \right). \quad (2.11)$$

Cluster size is defined as the number of computing nodes in a multi-node distributed framework (cluster). As we increase the cluster size N for the given QP problem, the size of subproblem in each distributed node becomes smaller, resulting in less computation time per iteration. However, the communication overhead will increase in this case. Therefore, r_{TSDA} and r_{LSDA} will increase with the increment of N . On the other hands, P^* remains constant irrespective of N , since the matrix β in Equation (2.9), which is the sum of all subset, is invariant. Moreover, k_{TSDA}^* and k_{LSDA}^* are only problem-dependent and hence, r_k is independent of N . Finally, r_p is expected to be same regardless of N , owing to the fact that both t_{TSDA}^p and t_{LSDA}^p decrease with the same amount as N increases. Based on above, we can infer the speedup with respect to the increase of N as follows.

When the computation is dominant (i.e., r_{TSDA} and r_{LSDA} are small), a large amount of speedup is expected as N increases. This is because the term $\frac{r_{\text{LSDA}}}{P^*}$ is a small number compared to r_{TSDA} , for sufficiently large P^* . In contrast, if the communication is dominant (i.e., r_{TSDA} and r_{LSDA} are very large), Equation (2.11) is approximated by $\text{SPEEDUP} \approx r_k r_p P^* \left(\frac{r_{\text{TSDA}}}{r_{\text{LSDA}}} \right)$. In this case, both r_{TSDA} and r_{LSDA} does not change significantly as N varies. Thus, the increase of the cluster size will not benefit the speedup. Figure 2.2 depicts the speedup versus the cluster size. In the computation-dominant area, one may expect more speedup as N increases, whereas speedup remains almost constant in the communication-dominant region.

2.5 Experiment and Results

Here, we present experimental details and performance evaluation of the proposed LSDA technique. The experiments were performed on a single dataset with varying cluster sizes. We considered small cluster sizes here to validate the proposed theory in the previous section. The trend in

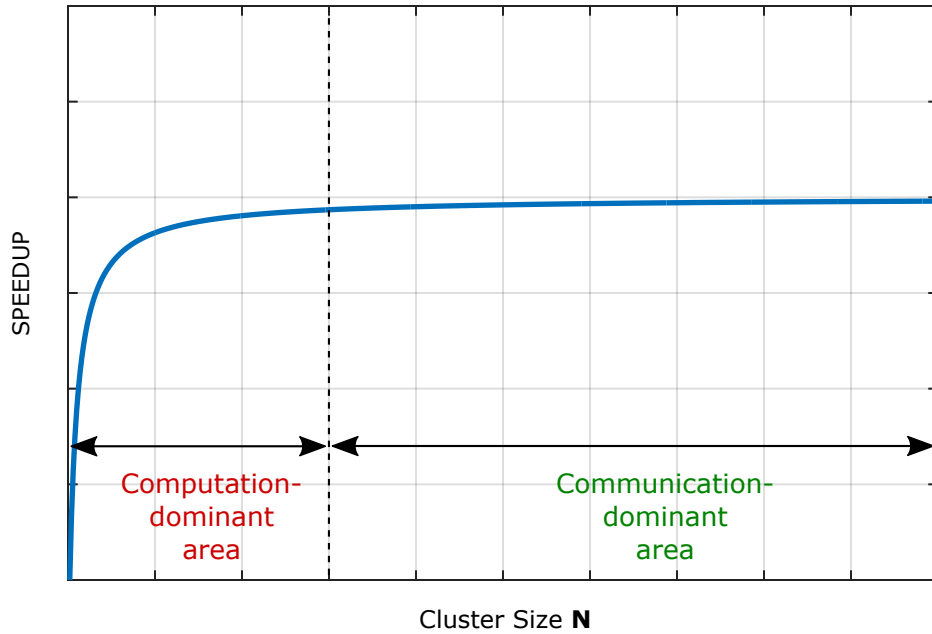


Figure 2.2: Theoretical result of speedup for LSDA algorithm with respect to TSDA algorithm for varying cluster size N . Reprinted with permission from [1].

the results is expected to follow even with large to very large sized clusters. All the experiments converged to the same value as that of the TSDA algorithm with varying execution times. Irrespective of the cluster size, the LSDA algorithm consistently outperformed TSDA with the same level of accuracy.

2.5.1 Hardware Description

We used a 40 node cluster of Amazon Web Services(AWS) Elastic Cloud Compute ² (EC2) instances. The EC2 instances originate in the same region US west (Oregon) and were spawned using the very basic *t2.micro* configuration supported by Elastic Block Storage volume of 8 GB size. Each of these instances, backed by Intel Xeon processors with clock speed up to 3.33 GHz, have one processing unit and 1 GB memory.

²<https://aws.amazon.com/documentation/ec2/>

2.5.2 Experimental Setup

The software implementation was done in C++11 supported by Armadillo(v5.400.2) [34] linear algebra library integrated with LAPACK/BLAS. We have used Message Passing Interface framework library(MPICH-v3.1.4) for the inter-node communication. The data was synthetically generated with random values uniformly distributed over $[-1, 1]$. The problem specifics are as follows:

1. Number of samples (instances) in synthetic dataset, $n = 200,000$.
2. Number of constraints, $m = 1$
3. Step size, $\alpha = 0.27$.
4. Optimal Synchronization Period, $P^* = 70$.
5. Stopping threshold, $\epsilon = 10^{-5}$.
6. Cluster Size, $N = \{10, 20, 32, 40\}$.

Cluster sizes are chosen such to distribute the samples, $n = 200,000$, equally among all the nodes. After distribution, the data set in each node has the following structure.

$$\mathbf{Q}_i \in \mathbb{R}^{\frac{n}{N} \times \frac{n}{N}}, \mathbf{A}_i \in \mathbb{R}^{1 \times \frac{n}{N}}, \mathbf{c}_i \in \mathbb{R}^{\frac{n}{N} \times 1}, \mathbf{x}_i \in \mathbb{R}^{\frac{n}{N} \times 1} \quad \forall i = 1, \dots, N$$

2.5.3 Results and Discussions

Here we present and discuss experimental results of the proposed LSDA algorithm under synchronization period, its convergence, computation and communication time, and speedup.

2.5.3.1 Synchronization Period

In LSDA, the inter-node communication, which results in data synchronization, occurs at fixed intervals of time defined by the synchronization period P .

Figure 2.3 shows the trend of number of iterations k , required for convergence with varying synchronization period P . Experimentally, it has been observed that the number of iterations k

required to converge to the optimal solution is *constant* for a particular synchronization period P , *irrespective of the cluster size*. That is, k is only dependent on P and independent of number of nodes N in a cluster. For synchronization period $P = 1$, the LSDA behaves exactly same as TSDA as expected. It is observed from Figure 2.3 that the minimum number of iterations needed for the convergence of our proposed relaxed synchronous algorithm occurs for a synchronization period $P = 70$. This synchronization period is the optimal value for which the fastest convergence of the given parallel QP problem is observed. This affirms the validity of Theorem 1. For the given data set, β in Equation (2.9), which is a scalar value in this example, is calculated as $\beta = 0.0143$. By applying the result in Equation (2.9), we obtain $P^* = 70$, which exactly coincides with the optimal value obtained from the experimental results as shown in Figure 2.3. For $P > 70$, k starts increasing implying that the dual variable \mathbf{y} is updated later than the desired time period and hence the longer convergence time.

Figure 2.4 compares the computation time for LSDA algorithm with varying synchronization period for different cluster sizes $N = \{10, 20, 32, 40\}$. Due to uncertainty in network delays, communication time is not taken into consideration here. For each of the cluster size, it is observed that the computation time also reaches a minima at the optimal synchronization period $P^* = 70$. Figure 2.5 highlights this trend for a single cluster size $N = 32$. It is interesting to note that Figure 2.5 validates our theoretical claim in Figure 2.1 for P^* as the optimal synchronization period.

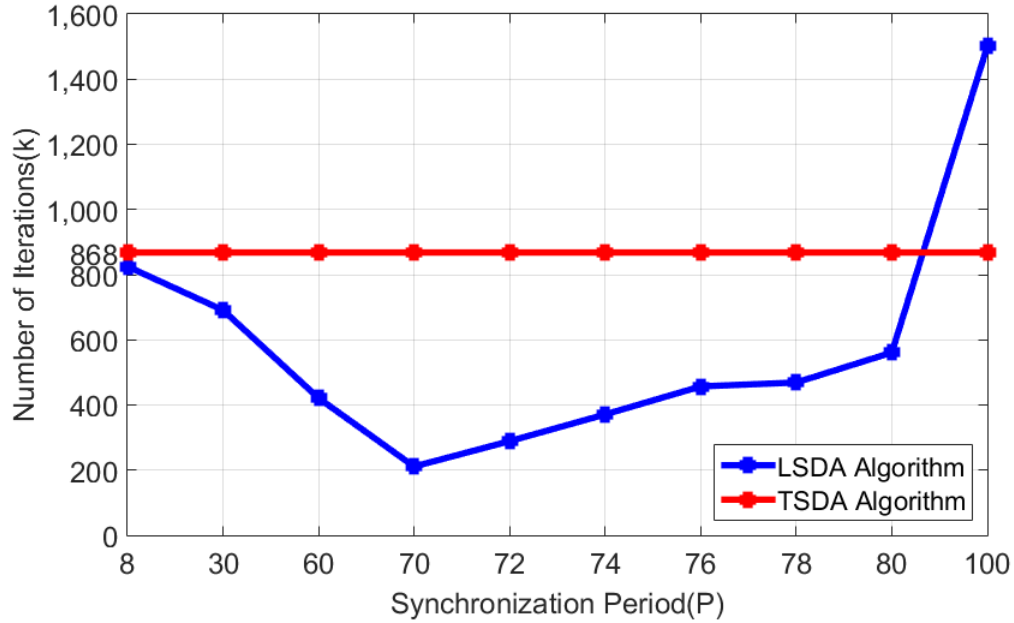


Figure 2.3: LSDA algorithm: Number of iterations (k) vs Synchronization Period (P). The number of iterations required for the TSDA algorithm to converge is constant. Reprinted with permission from [1].

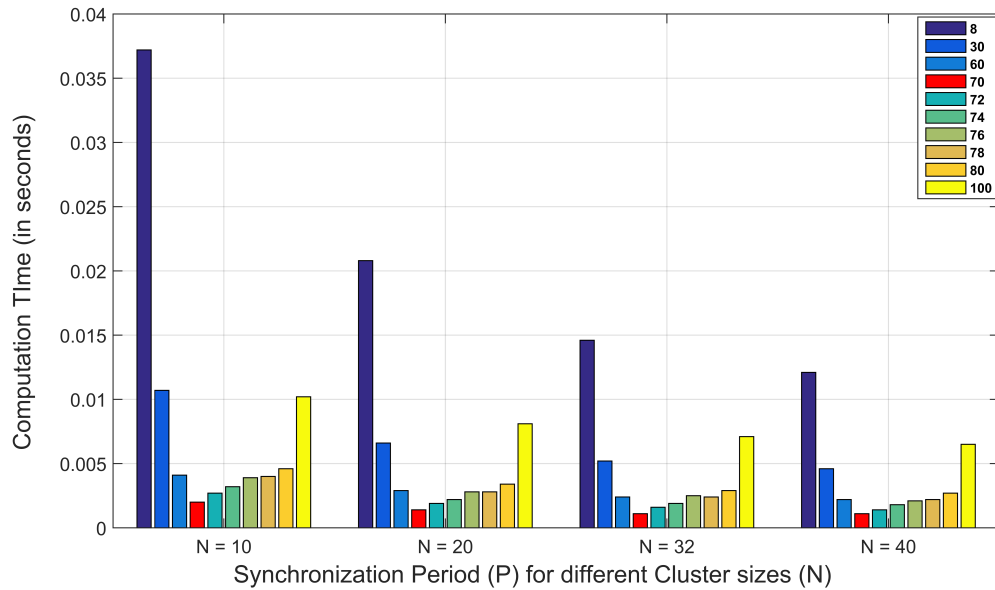


Figure 2.4: LSDA algorithm: Computation Time vs Synchronization Period, cluster size $N = \{10, 20, 32, 40\}$. Reprinted with permission from [1].

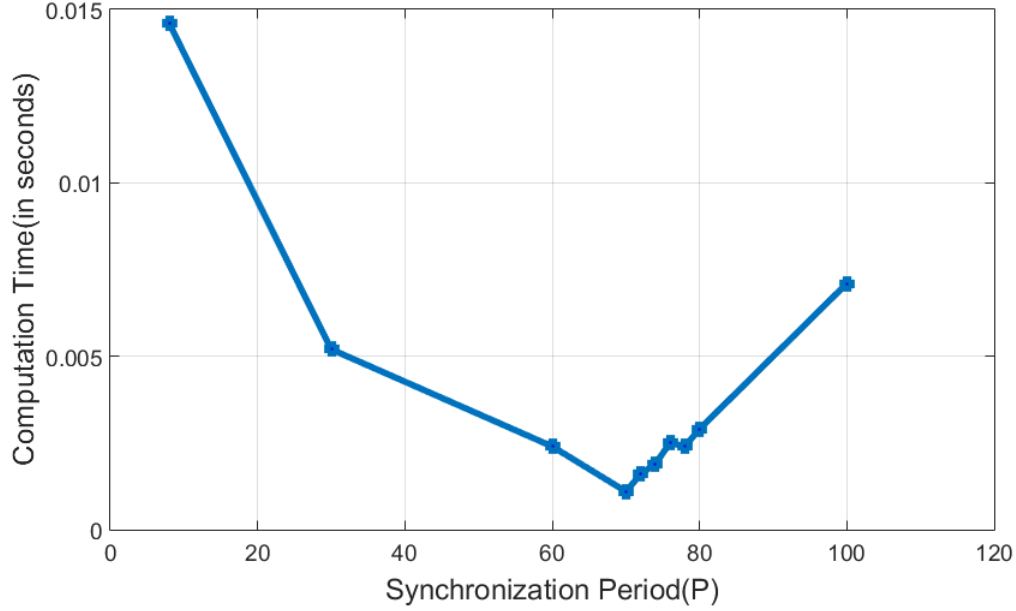


Figure 2.5: LSDA: Computation Time vs Synchronization period, cluster size $N = 32$. Reprinted with permission from [1].

2.5.3.2 Convergence

Figure 2.6 compares the convergence trend for LSDA algorithm with $P^* = 70$ with the TSDA algorithm. As guaranteed theoretically, the LSDA algorithm consistently converges to the optimal solution of the (scalar) dual variable $y^* = 188.569$ in $k = 211$ iterations, synchronizing once in every $P^* = 70$ iterations. The TSDA algorithm takes $k = 868$ iterations for convergence, synchronizing every iteration. This behavior validates that the proposed algorithm is numerically stable and accurate even with the lazy updates. Also, it is observed that the convergence trend for the relaxed algorithm is piece-wise linear because the y -update in Equation (2.5) has a constant increment between consecutive synchronizations. This linearity in convergence trend can be used to analytically determine the value of dual variable y at the synchronization time instants using point-slope form of equation of line. This will further reduce the calculation time by eliminating the need for y -update during every internal iteration. However, this improvement in computation time will not significantly improve the overall execution time as it constitutes just a small fraction

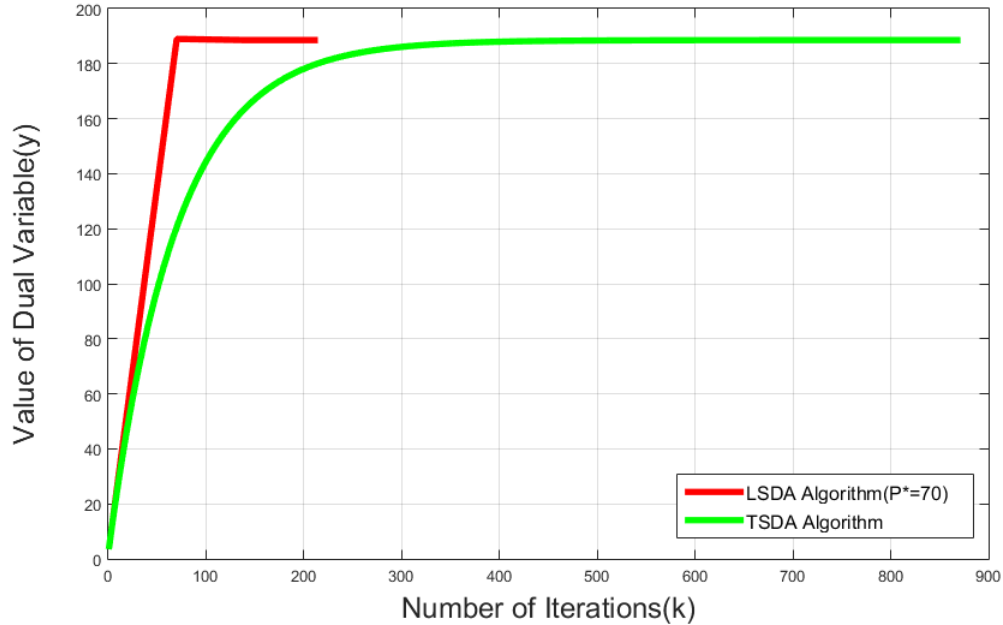


Figure 2.6: Dual variable solution (y) vs Number of iterations (k). LSDA algorithm converges to the optimal solution of the dual variable significantly faster than the TSDA algorithm. Reprinted with permission from [1].

of the overall time as observed in Figure 2.7 for TSDA and Figure 2.8 for LSDA. In the proposed LSDA algorithm, synchronization occurs roughly thrice ($\approx 211/70$) compared to 868 synchronizations in TSDA algorithm. Consequently, by using the relaxed synchronization approach the communication delay is reduced by 99.65%.

2.5.3.3 Computation Time

Computation time is defined as the time spent in various calculations in the cluster nodes during the iterations in between the synchronization time instants. Keeping the data set fixed, increasing the number of nodes in a cluster reduces the computation time as the work load per node decreases with the increase in cluster size. This trend can be observed in Figure 2.7 for TSDA algorithm and in Figure 2.8 for LSDA algorithm. On comparing the computation time of LSDA algorithm in Figure 2.8 with that of the TSDA algorithm in Figure 2.7 for varying cluster size, a significant improvement in the computation time is observed in LSDA algorithm. This significant boost in the computation time is due to lesser number of local calculations in the cluster nodes which can be

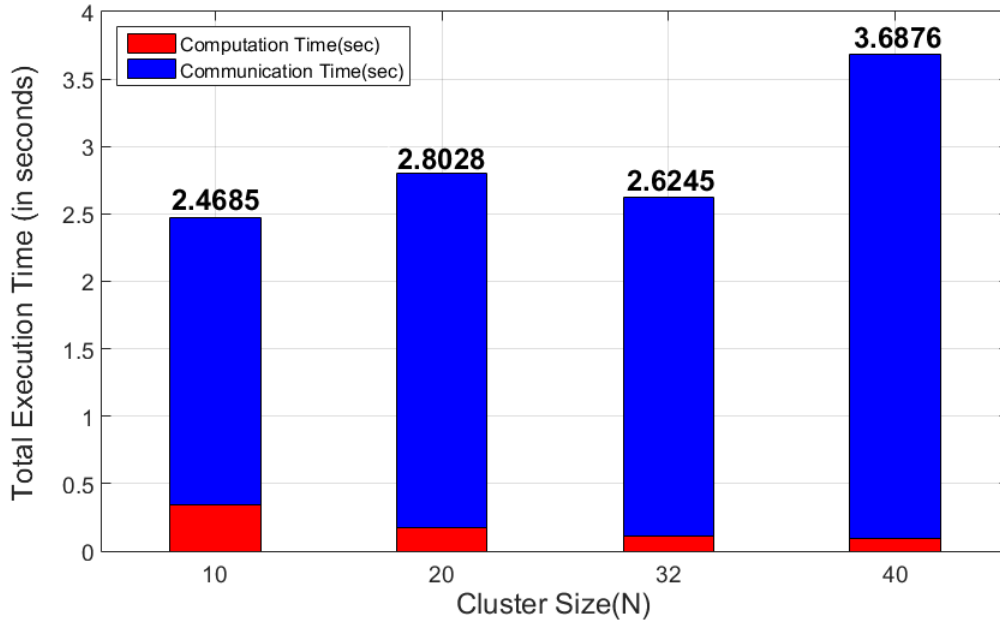


Figure 2.7: TSDA algorithm: Total execution time vs Cluster size. Across various cluster sizes, it is observed that communication time is more dominating than the actual computation time. Reprinted with permission from [1].

attributed to the reduced number of iterations k as observed in Figure 2.6.

2.5.3.4 Communication Time

Communication time is defined as the time spent for inter-node communication during a synchronization step. Figure 2.8 shows the total of communication and computation time (processor execution time) for LSDA algorithm. As discussed earlier, with increase in cluster size N , we do observe decrease in computation time as work load per node reduces. Whereas, the inter-node communication delay increases as the number of nodes in the cluster increases. In addition, it can also be observed from Figure 2.8 that out of the total execution time only a small fraction is used in computation whereas majority of it is spent on communicating the data across the nodes. These observations put forward a motivation to design communication-efficient techniques from the software perspective which we will present in Chapter 5. Also, it is worth investigating dedicated hardware architecture for distributed data processing for applications in machine learning which we discuss in Chapter 6.

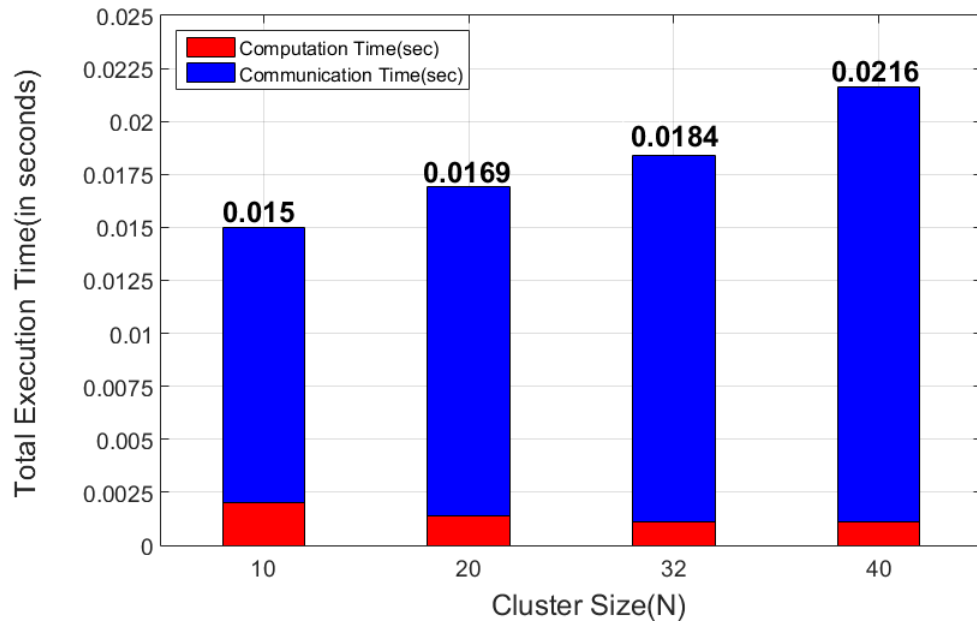


Figure 2.8: LSDA algorithm: Total execution time vs Cluster size. The total execution time taken for LSDA is significantly less than for TSDA. Reprinted with permission from [1].

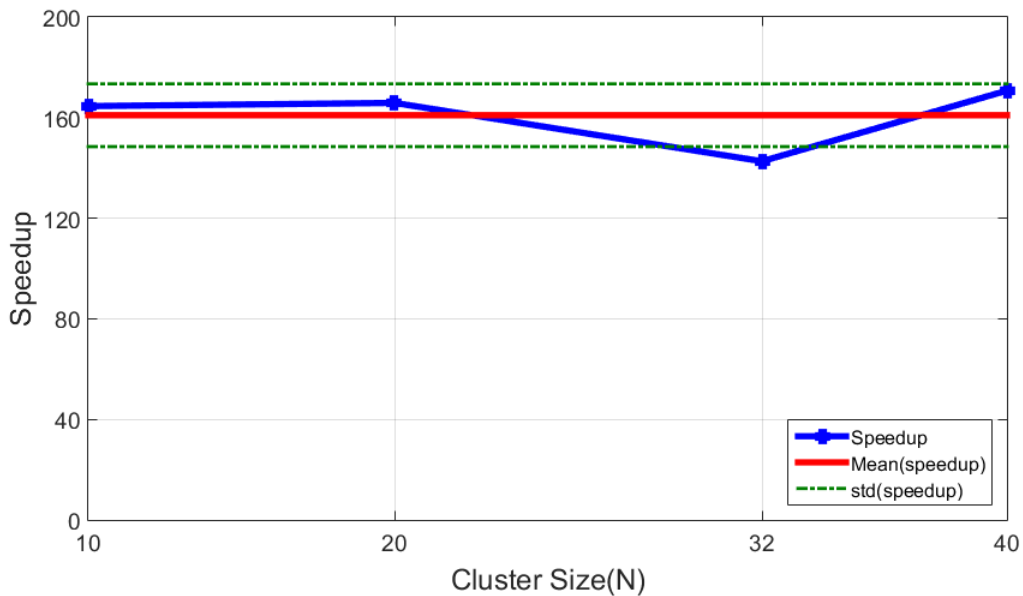


Figure 2.9: Speedup in overall execution time of LSDA algorithm with respect to TSDA algorithm for varying cluster sizes. The variation in speedup is shown with the mean value as well as the standard deviation for each cluster size. Reprinted with permission from [1].

2.5.3.5 Speedup

Figure 2.9 shows the speedup in overall execution time achieved by the proposed LSDA algorithm for different cluster sizes over the TSDA algorithm. For cluster size $N = \{10, 20, 32, 40\}$, an average speedup of around $160 \times (\pm 12.5)$ is achieved. It is worth noting that the speedup is approximately constant and flattens out with increase in cluster size. This observation aligns with our theoretical discussion in Section 2.4.4 and implies that the cluster size $N = \{10, 20, 32, 40\}$ lies in the communication-dominant region. Figures 2.7 and 2.8 support this trend, since even with cluster size $N = 10$, the communication time dominates the total execution time. In fact, this leads to a good motivation to determine the optimal cluster size that is problem specific and is expected to fall in the region where the speedup begins to flatten out. In other words, optimal cluster size is that value of N at which the communication time begins to dominate the computation time.

2.6 Summary

In this chapter, we proposed a relaxed synchronization approach to solve massively parallel large-scale Quadratic Programming (QP) problems. We analytically prove that our algorithm is numerically stable that converges to the same optimal solution as the synchronous implementation. As outlined in the chapter, communication between processing nodes is a critical bottleneck in parallel computing. The proposed algorithm alleviates this bottleneck to a large extent by employing intermediate synchronizations between iterations, which results in significant speedup of around $160 \times$ when compared to the tightly synchronized dual ascent algorithm. This intermediate synchronizations decrease the inter-node communication overhead by 99.65% and hence a speedup in solution time is observed. We also provide a theory to determine the optimal synchronization period P^* that guarantees the fastest convergence of a given parallel QP problem. In addition we prove, both analytically and experimentally, that for a given problem we can only attain a constant speedup on continuously increasing the cluster size. In subsequent chapters, we will use LSDA and optimal synchronization period to design efficient distributed training algorithms for solving optimization problems in machine learning.

3. HOUSEHOLDER SKETCH FOR MACHINE LEARNING ¹

Least-Mean-Squares (LMS) solvers comprise a class of fundamental optimization problems in machine learning such as linear regression, and regularized regressions such as Ridge, LASSO, and Elastic-Net. Data summarization techniques for big data generate summaries called coresets and sketches to speed up model learning under streaming and distributed settings. In this chapter, we explore such data summarization technique based on QR decomposition and apply it to efficiently solve large-scale distributed LMS problems across various worker nodes.

3.1 Introduction

Least-Mean-Squares (LMS) solve a fundamental slice of machine learning optimization and statistics problems that comprise ordinary least squares linear regression [35], regularized models such as ridge regression [36], LASSO [37], Elastic-net [38]. Such machine learning models are statistically robust and easily interpretable. Hence, they find applications in cancer research [39], genomics [40], cryptocurrency [41], and most recently in understanding factors of COVID-19 outbreak and its impact [42, 43, 44]. Data summarization techniques for big data generate summaries called coresets and sketches to speed up model learning under streaming and distributed settings. A *coreset* is a (weighted) subset of data points whereas a *sketch* is a linear mapping of few or all data points in the original dataset which aim to preserve or approximate the covariance matrix. Such data summaries are leveraged to speedup model learning by solving the machine learning problem approximately. For example, authors in [45] design a fast and accurate coreset-sketch fusion on input data to boost the performance of existing LMS solvers.

3.1.1 Motivation

Training machine learning models on big data requires resources with large memory, and high computational power. However, in practical applications, data is naturally decentralized, which

¹This chapter is reprinted with permission from “Householder Sketch for Accurate and Accelerated Least-Mean-Squares Solvers” by Jyotikrishna Dass, and Rabi N. Mahapatra, 2021 Proceedings of the 38th International Conference on Machine Learning (ICML), Copyright ©2021 Jyotikrishna Dass and Rabi Mahapatra.

calls for collaboratively training global machine learning model across multiple sources without sharing original data on a central server. A recent body of work aims to create such secure multi-party computation framework for training distributed regression models [46, 47, 48, 49]. In addition, efforts have to be made towards feasibility of real-time learning at these sources with relatively smaller memory and computing capabilities. Such distributed learning framework will also need to support streaming data batches to update the models. Addressing the above requirements entails reformulating the fundamental machine learning problems and using smaller and efficient representations of the original full data to accelerate the training time. Data summarization techniques for big data in [50, 51, 52, 53, 54] generate such summaries called coresets and sketches to solve the problems approximately.

More recently, an award-winning work [45] proposed a coreset and sketch fusion algorithm LMS-BOOST which accurately solves and accelerates common LMS solvers for Linear, Ridge, LASSO, Elastic-Net in scikit-learn library up to 100x. Specifically, they generate accurate coresets from their proposed faster implementation of Caratheodory set in $O(nd + d^4 \log n)$ (refer Algorithm 1, Theorem 3.1 in [45]) which is based on Caratheodory’s Theorem [55] proposed in 1907. The theorem states that every point contained in the convex hull of n datapoints in \mathbb{R}^d can be represented as a convex combination of a subset of at most $d + 1$ points, which the authors in [45] call the Caratheodory set. The fundamental idea in their novel fusion algorithm is to partition the n data points each with feature dimension d into multiple clusters optimally, then compute sketch for each cluster, followed by generating coreset for the union of sketches. Finally, they create a union of clusters corresponding to selected sketches above and recursively run the original (slower) Caratheodory algorithm [55] on this union to generate sufficiently small coreset. The authors in [45] use this coreset to summarize the data into $O(d^2) \times d$ memory and run LMS solvers with the computational time complexity of $O(nd^2 + \log n \times d^8)$ (refer Algorithms 3 – 4, Theorem 3.2 in [45]) while preserving the input covariance. In retrospect, we seek to explore classical Householder transformation [56] as a candidate for sketching while accurately solving LMS problems and being more efficient than LMS-BOOST in [45].

3.1.2 Contributions

Authors in [45] considered classical QR decomposition approach to be a stable alternative for solving LMS problems, but they naively declared it to be relatively time consuming without providing any theoretical or empirical comparisons (**Claim 1**). In addition, QR technique was referred to be unsuitable for exact factorization for streaming data (**Claim 2**). In this chapter, we set to rigorously test and check for validity of the above claims made against the classical QR decomposition.

To address **Claim 1**, we pose the following question

“Whether a recursive and clustering-based algorithm LMS-BOOST is a better alternative than classical QR as a pre-processing step to (theoretically) accurately solve and accelerate common LMS solvers ?”

To our surprise, we found **Claim 1** to be **Not True**. Hence, our first main contribution is to provide a critical analysis based on thorough comparison of both the algorithms both theoretically and via extensive empirical results which are missing in the literature.

To address **Claim 2**, we offer a decentralized QR setup for exactly factorizing the input data partitioned across multiple worker nodes based on local QR factorization without sharing the original data (privacy).

With above settings, we found **Claim 2** to be also **Not True**. Hence, our second main contribution is to justify the above decentralized QR setup as numerically stable and a suitable candidate to generate distributed sketches via exact factorization on streaming data.

This chapter provides a strong case for classical QR decomposition-based sketch by addressing the above claims and by performing extensive empirical evaluations to demonstrate its performance capabilities for solving common LMS problems. Eventually, we shed light on benefits of the Householder representations in terms of its memory, computation time to accelerate LMS solvers, its numerical stability, feasibility of being deployed on distributed network to handle large sample size and feature dimensions, and scalability across multiple worker nodes.

3.2 Related Work

The overdetermined LMS problems can be solved directly by computing the covariance matrix and its inverse in the Normal Equations. Despite its ease of implementation, this method is not recommended due to its numerical instability associated with squaring of condition number. Moreover, computing and maintaining the inverse, if it exists, for every new incoming data leads to the accumulation of numerical errors, especially with 32-bit floating-point calculations. Hence, a more numerically stable approach is to factorize X using QR decomposition [57] into matrix Q with orthonormal columns, and upper triangular matrix R , when $n \gg d$. QR decomposition can handle a much wider range of matrix by avoiding the condition-number-squaring effect [57].

A standard method of obtaining a QR Factorization is via Gram-Schmidt Orthogonalization [58] that is unstable compared to QR via Householder reflectors [57]. For applications with sparse input matrices, Givens rotation [59] allows full exploitation of the underlying sparsity. However, it is to be noted that Householder transformation uses fewer arithmetic operations compared to Givens rotations [57]. Nevertheless, it is always possible to compute efficient sketch by switching between Householder transformation and Given rotation implementation of QR decomposition based on the sparsity of the input data without much change in the structure of the framework.

Various iterative methods have been proposed to solve least squares problems such as LSQR [60], LSMR [61], Stochastic proximal accelerated gradient descent [62], and more recently randomized methods with random mixing and random sampling such as LSRN [63] and Blendepic [64]. These results are approximate solutions. However, our focus is to use a theoretically accurate sketch of the input data which could then be directly plugged into scikit-learn LMS solvers similar to LMS-BOOST algorithm in [45] but with faster execution time.

3.3 Least Mean Squares - QR (LMS-QR)

We define the Least-Mean-Squares (LMS) optimization problem for a data set (\mathbf{X}, \mathbf{y}) , $\mathbf{X} \in \mathbb{R}^{n \times d}$, and $\mathbf{y} \in \mathbb{R}^n$, where, $n \gg d$, as minimizing the sum of squared loss between the observed prediction $\mathbf{x}_j \mathbf{w}$, and true response y_i for any d - dimensional data sample $\mathbf{x}_j^T \in \mathbb{R}^d$ (row of \mathbf{X}),

$j = 1, \dots, n$, and LMS model coefficient vector $\mathbf{w} \in \mathbb{R}^d$. The generic optimization problem for LMS is defined as follows

$$\min_{\mathbf{w}} f(\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2) + g(\mathbf{w}). \quad (3.1)$$

The above formulation is modelled as a specific LMS problem based on choice of functions $f(z)$ and $g(\mathbf{w})$. For example, in LINEAR REGRESSION, $f(z) = z^2$, and $g(\mathbf{w}) = 0$.

In RIDGE REGRESSION, $f(z) = z^2$, and $g(\mathbf{w}) = \lambda\|\mathbf{w}\|_2$, where, $\lambda > 0$, is ridge regularization hyper-parameter. Such LMS solvers as those in scikit-learn library basically solve system of linear equations, i.e. $(\mathbf{X}^T\mathbf{X})\mathbf{w} = \mathbf{X}^T\mathbf{y}$ (LINEAR REGRESSION), or $(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})\mathbf{w} = \mathbf{X}^T\mathbf{y}$ (RIDGE REGRESSION).

3.3.1 Theory

The LMS objective in Equation (3.1) with $\mathbf{X} = \mathbf{QR}$ reformulates to

$$\min_{\mathbf{w}} f(\|\mathbf{QR}\mathbf{w} - \mathbf{y}\|_2) + g(\mathbf{w}). \quad (3.2)$$

where, \mathbf{Q} is an $n \times n$ orthogonal matrix and \mathbf{R} is an $n \times d$ upper trapezoidal matrix when $n > d$. We assume the data matrix \mathbf{X} is full column rank. With Householder transformation, the benefit is avoiding the explicit computation and storing of large \mathbf{Q} which otherwise becomes prohibitive for big data sets. Rather, one may simply represent \mathbf{Q} as a set of d Householder matrices $\mathcal{H} = \{\mathbf{H}(j) : j = 1, \dots, d\}$ such that $\mathbf{Q} = \mathbf{H}(1) \times \mathbf{H}(2) \dots \times \mathbf{H}(d)$. Each $\mathbf{H}(j)$ has the form $\mathbf{H}(j) = \mathbf{I} - \tau \times \mathbf{v}(j) \times \mathbf{v}(j)'$, where τ is a real scalar, and $\mathbf{v}(j)$ is a real vector called Householder reflector which we store in the reflector set $\mathcal{V} = \{\mathbf{v}(j) : j = 1, \dots, d\}$. In essence, any operations involving \mathbf{Q} are now computed using the above memory-efficient reflector set. Theorem 3.1 formally presents Householder-QR.

Theorem 3.1 (Householder-QR [57]). *Let matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ with $n > d$. Householder QR decomposition of \mathbf{X} generates set of d Householder matrices \mathcal{H} and an $n \times d$ upper trapezoidal matrix \mathbf{R} . The Householder matrices are stored as a set of d Householder reflectors \mathcal{V} . Total memory*

Algorithm 3.1 HOUSEHOLDER-SKETCH(\mathbf{X}, \mathbf{y}); see Theorem 3.2

Data: A matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, a vector $\mathbf{y} \in \mathbb{R}^n$

Result: A matrix $\mathbf{R} \in \mathbb{R}^{d \times d}$ is upper triangular such that $\mathbf{X}^T \mathbf{X} = \mathbf{R}^T \mathbf{R}$, and a vector $\bar{\mathbf{y}} \in \mathbb{R}^d$ is top d elements of the reflected vector $\mathbf{Q}^T \mathbf{y}$

```

1 ( $\mathcal{V}, \mathbf{R}$ ) := HOUSEHOLDER-QR( $\mathbf{X}$ )                                // see Theorem 3.1
2  $\bar{\mathbf{y}} := \text{MULTIPLY-QC}(\mathcal{V}, \mathbf{y}, \text{'T'})$                         // implicit  $\mathbf{Q}^T \mathbf{y}$ , see [57]
3  $\mathbf{R} \leftarrow \mathbf{R}[0 : d, :]$                                        //  $d \times d$  triangular block
4  $\bar{\mathbf{y}} \leftarrow \bar{\mathbf{y}}[0 : d]$                                        // top  $d$  elements
5 return ( $\mathbf{R}, \bar{\mathbf{y}}$ )

```

footprint of above factors is nd elements with time complexity of $O(nd^2)$ for $n \gg d$.

We now present Theorem 3.2 which uses the factors from Householder-QR to create Householder sketch as per Algorithm 3.1 for further applications in LMS problems.

Theorem 3.2 (Householder Sketch). *Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ be the original data matrix, $\mathbf{y} \in \mathbb{R}^n$ be the corresponding output label or response vector, and $n \gg d$. Let $\mathbf{X} = \mathbf{QR}$ be Householder QR decomposition. Then, $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ is a memory-efficient and theoretically accurate sketch of original data (\mathbf{X}, \mathbf{y}) such that $\mathbf{X}^T \mathbf{X} = \mathbf{R}^T \mathbf{R}$, and has memory footprint of $(\frac{d(d+3)}{2})$ elements, computed in time $O(nd^2)$.*

Proof. From Equations (3.1) and (3.2), and $\mathbf{X} = \mathbf{QR}$, where $\mathbf{Q}\mathbf{Q}^T = \mathbf{Q}^T \mathbf{Q} = \mathbf{I}$

$$\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2 = \|\mathbf{QR}\mathbf{w} - \mathbf{y}\|_2 = \|\mathbf{QR}\mathbf{w} - \mathbf{Q}\mathbf{Q}^T \mathbf{y}\|_2 = \|\mathbf{Q}\|_2 \|\mathbf{R}\mathbf{w} - \mathbf{Q}^T \mathbf{y}\|_2 = \|\mathbf{R}\mathbf{w} - \mathbf{Q}^T \mathbf{y}\|_2$$

(Accurate sketch) So, it is possible to replace the original data (\mathbf{X}, \mathbf{y}) used in existing LMS solvers with $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ which preserves the covariance $\mathbf{X}^T \mathbf{X} = \mathbf{R}^T \mathbf{Q}^T \mathbf{Q} \mathbf{R} = \mathbf{R}^T \mathbf{R}$ and solves the optimization problem accurately. For example, Ridge regression with ridge parameter λ solves $(\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})\mathbf{w} = \mathbf{X}^T \mathbf{y}$ in primal form which can be reformulated to $(\mathbf{R}^T \mathbf{R} + \lambda \mathbf{I})\mathbf{w} = \mathbf{R}^T (\mathbf{Q}^T \mathbf{y})$.

(Memory savings) \mathbf{R} is a $d \times d$ upper triangular matrix with $(\frac{d(d-1)}{2})$ elements above the diagonal and d on the diagonal resulting in $(\frac{d(d+1)}{2})$ elements compared to original data matrix \mathbf{X}

that has nd elements. $\mathbf{Q}^T \mathbf{y}$ is a reflected response vector. It is to be noted that only top d rows of \mathbf{Q}^T will be sufficient to compute $\mathbf{Q}^T \mathbf{y}$ since $n \gg d$. Hence, reflected response vector ($\mathbf{Q}^T \mathbf{y}$) is of size d compared to the original LMS formulation with response vector \mathbf{y} of size n . Hence, the total memory footprint of $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ is $O(\frac{d(d+3)}{2})$ elements which makes it memory-efficient than the original (\mathbf{X}, \mathbf{y}) occupying $n(d+1)$ space.

(Time complexity) The above sketch $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ is computed via Householder QR decomposition (HOUSEHOLDER-QR in Step 1 of Algorithm 3.1) of \mathbf{X} which generates upper triangular matrix, \mathbf{R} , and orthonormal matrix \mathbf{Q} that is internally stored as Householder reflectors. The time complexity of the above decomposition is $O(nd^2 - d^3/3)$ [57]. Calculation of $\mathbf{Q}^T \mathbf{y}$ is done implicitly by applying Householder reflectors to the response vector \mathbf{y} (MULTIPLY-QC in Step 2 of Algorithm 3.1) in time $O(nd)$ [57]. Hence, it can be seen that the total computation time for the sketch $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ is $O(nd^2 + nd - d^3/3)$ which results in $O(nd^2)$ for $n \gg d$. \square

3.3.2 Accelerating Least Mean Squares Solvers

Householder sketch $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ is precomputed and applied directly to existing LMS solvers in scikit-learn library instead of original data (\mathbf{X}, \mathbf{y}) . Reducing the memory cost from $O(nd)$ in (\mathbf{X}, \mathbf{y}) to $O(d^2)$ in Householder sketch $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ speeds up the LMS solver. Specifically, with the Householder sketch, the time for constructing $\mathbf{R}^T \mathbf{R}$ and $\mathbf{R}^T (\mathbf{Q}^T \mathbf{y})$ are $O(d^3)$ and $O(d^2)$, respectively which are significantly faster than the original time $O(nd^2)$ and $O(nd)$ spent by the solver on constructing $\mathbf{X}^T \mathbf{X}$ and $\mathbf{X}^T \mathbf{y}$, respectively. Then, LMS solves a *primal* problem with system of d equations and d unknowns (\mathbf{w} , solver coefficients). We present LMS-QR formally in Algorithm 3.2. It is to be noted that constructing HOUSEHOLDER-SKETCH takes $O(nd^2)$ for $n \gg d$ (see Theorem 3.2), and is more computationally dominant than running the LMS solver such as LINREG, RIDGECV, LASSOCV, ELASTICCV in scikit-learn. The above trend is also empirically validated in Figure 3.2 for RIDGE solver as proof of concept.

Theorem 3.3. Let $\mathbf{X} \in \mathbb{R}^{n \times d}$, $\mathbf{y} \in \mathbb{R}^n$, $(\mathbf{R}, \mathbf{Q}^T \mathbf{y}) := \text{HOUSEHOLDER-SKETCH}(\mathbf{X}, \mathbf{y})$ that ac-

Algorithm 3.2 LMS-QR($\mathbf{X}, \mathbf{y}, \text{params}$)

Data: A matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$, a vector $\mathbf{y} \in \mathbb{R}^n$, and a list of LMS parameters, params

Result: A vector of model coefficients, $\mathbf{w} \in \mathbb{R}^d$

- 1 $(\mathbf{R}, \bar{\mathbf{y}}) := \text{HOUSEHOLDER-SKETCH}(\mathbf{X}, \mathbf{y})$ // see Algorithm 3.1
 - 2 $\mathbf{w} := \text{LMS}(\mathbf{R}, \bar{\mathbf{y}}, \text{params})$ // LINREG, RIDGECV, LASSOCV, ELASTICCV in
sckit-learn
 - 3 **return** \mathbf{w}
-

celerates primal ridge solver via RIDGE-QR. Then, $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ is also the Householder sketch for the corresponding Kernel Ridge Regression problem that accelerates the dual problem via KERNELRIDGE-QR, and solves it with the same memory and time complexity, independent of data size (n) , as that of primal RIDGE-QR.

Proof. Kernel Ridge Regression with original data (\mathbf{X}, \mathbf{y}) solves $(\mathbf{K} + \lambda \mathbf{I})\boldsymbol{\beta} = \mathbf{y}$, where, $\mathbf{K} \in \mathbb{R}^{n \times n}$ is the Kernel matrix, and $\boldsymbol{\beta} \in \mathbb{R}^n$ is the vector of dual variables. For any pair of row vectors in input data, $\mathbf{x}_i, \mathbf{x}_j \in \mathbb{R}^{1 \times d}$, each element of Kernel matrix $\mathbf{K}(i, j) = \kappa(\mathbf{x}_i, \mathbf{x}_j)$, where $\kappa(\cdot)$ is a Reproducing Kernel Hilbert Space (RKHS) kernel function such that $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$ and $\phi(\cdot)$ is transformation from input space to RKHS feature space [65].

For a **linear kernel function**, $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \mathbf{x}_j^T$, $\mathbf{K} = \mathbf{X} \mathbf{X}^T$, and the objective is to solve the equation $(\mathbf{X} \mathbf{X}^T + \lambda \mathbf{I})\boldsymbol{\beta} = \mathbf{y}$ such that the model coefficient in input space, $\mathbf{w} = \mathbf{X}^T \boldsymbol{\beta}$. By applying $\mathbf{X} = \mathbf{Q} \mathbf{R}$ via HOUSEHOLDER-QR(\mathbf{X}), the above dual problem reformulates to

$$\begin{aligned} & (\mathbf{X} \mathbf{X}^T + \lambda \mathbf{I})\boldsymbol{\beta} = \mathbf{y} \\ \Rightarrow & \quad (\mathbf{Q} \mathbf{R} \mathbf{R}^T \mathbf{Q}^T + \lambda \mathbf{I})\boldsymbol{\beta} = \mathbf{y} \\ \Rightarrow & \quad (\mathbf{Q} \mathbf{R} \mathbf{R}^T \mathbf{Q}^T + \lambda \mathbf{Q} \mathbf{Q}^T)\boldsymbol{\beta} = \mathbf{y} \\ \Rightarrow & \quad \mathbf{Q}(\mathbf{R} \mathbf{R}^T \mathbf{Q}^T + \lambda \mathbf{Q}^T)\boldsymbol{\beta} = \mathbf{y} \\ \Rightarrow & \quad (\mathbf{R} \mathbf{R}^T \mathbf{Q}^T + \lambda \mathbf{Q}^T)\boldsymbol{\beta} = \mathbf{Q}^T \mathbf{y} \\ \Rightarrow & \quad (\mathbf{R} \mathbf{R}^T + \lambda \mathbf{I})\bar{\boldsymbol{\beta}} = \bar{\mathbf{y}} \end{aligned}$$

where, $\bar{\mathbf{y}} = \mathbf{Q}^T \mathbf{y}$ and $\bar{\boldsymbol{\beta}} = \mathbf{Q}^T \boldsymbol{\beta}$. The model coefficients in the input space,

$$\mathbf{w} = \mathbf{X}^T \boldsymbol{\beta} = \mathbf{R}^T \mathbf{Q}^T \boldsymbol{\beta} = \mathbf{R}^T \bar{\boldsymbol{\beta}}$$

Hence, solving $(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I})\boldsymbol{\beta} = \mathbf{y}$, system of n equations in n unknowns in KERNELRIDGE with original (\mathbf{X}, \mathbf{y}) is equivalent to solving a much smaller system of d equations in d unknowns ($n \gg d$), *accurately* and *faster*, with $(\mathbf{R}\mathbf{R}^T + \lambda\mathbf{I})\bar{\boldsymbol{\beta}} = \bar{\mathbf{y}}$ in KERNELRIDGE-QR with memory-efficient $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ sketch. It is worth noting here that once $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ sketch is available, the memory and time complexity for solving *dual* in KERNELRIDGE-QR is *independent* of data size n , and is *same* to that of solving the same problem in *primal* form via RIDGE-QR. Figure 3.4(a) demonstrates the the above similarity in solving RIDGE-QR, and KERNELRIDGE-QR (with linear kernel) based on computation time. Moreover, KERNELRIDGE-QR calculates the model coefficient \mathbf{w} using a triangular matrix in $\mathbf{w} = \mathbf{R}^T \bar{\boldsymbol{\beta}}$ in d^2 flops compared to $(2n - 1)d$ flops for $\mathbf{w} = \mathbf{X}^T \boldsymbol{\beta}$ in the original KERNELRIDGE with (\mathbf{X}, \mathbf{y}) .

For any **non-linear kernel** function such as Radial Basis Function, it is possible to represent $\mathbf{K} \approx \mathbf{A}\mathbf{A}^T$ with some low-rank matrix, $\mathbf{A} \in \mathbb{R}^{n \times k}$ via any kernel approximation techniques [66, 5]. This can be followed by constructing memory-efficient $(\mathbf{R}, \mathbf{Q}^T \mathbf{y}) := \text{HOUSEHOLDER-SKETCH}(\mathbf{A}, \mathbf{y})$ from Algorithm 3.1. Now, solving the approximated *dual* problem formulation for non-linear kernels via KERNELRIDGE-QR is equivalent in space and time complexity to solving the approximated problem in *primal* form via RIDGE-QR on $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$. Moreover, any of the above RIDGE-QR or KERNELRIDGE-QR is faster than solving the *primal* form via RIDGE with (\mathbf{A}, \mathbf{y}) .

Hence, we showed that $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ is also the Householder sketch for Kernel Ridge Regression, where, \mathbf{R} is defined based on linear or non-linear kernel, for accelerating the *dual* problem via KERNELRIDGE-QR. □

3.4 Distributed LMS-QR

In this section, we present a distributed version of LMS-QR that parallelizes the computations, and, scales Algorithm 3.2 across multiple workers (computing cores or users) to solve the

global LMS problem. We recall that HOUSEHOLDER-SKETCH (see Algorithm 3.1) is the more computational dominant step than the solver in LMS-QR (see Algorithm 3.2). HOUSEHOLDER-SKETCH calls HOUSEHOLDER-QR to perform $\mathbf{X} = \mathbf{QR}$ decomposition on the data matrix. Then, MULTIPLY-QC is invoked to eventually compute $\bar{\mathbf{y}} = \mathbf{Q}^T \mathbf{y} \in \mathbb{R}^d$.

Now, we show the feasibility of *exactly* distributing LMS-QR algorithm across multiple worker nodes without any approximations. To design a DISTRIBUTED LMS-QR in Algorithm 3.3, the more computationally expensive HOUSEHOLDER-SKETCH algorithm is parallelized across multiple *workers*. This is achieved via DISTRIBUTED HOUSEHOLDER-QR (see Theorem 3.4, and Steps 1-9 in Algorithm 3.3), and DISTRIBUTED MULTIPLY-QC (see Corollary 3.1, and Steps 10-19 in Algorithm 3.3) which create local and master Householder sketches from the local data $(\mathbf{X}_i, \mathbf{y}_i), i = 1, \dots, p$, generated *i.i.d.* at each worker node. Finally, we run the LMS solver (Steps 20 - 24 in Algorithm 3.3) at the *master* with $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$, and broadcast the global LMS model coefficients \mathbf{w} to all the workers.

Theorem 3.4 (Distributed Householder-QR [6]). *Let $\mathbf{X} = (\mathbf{X}_1^T | \dots | \mathbf{X}_p^T)^T$, where, $\mathbf{X}_i \in \mathbb{R}^{\hat{n} \times d}$ be local data matrix of parallel worker, $i = 1, \dots, p$, where $\hat{n} \gg d$, and, $n = p\hat{n}$. Let, $\mathbf{X}_i = \mathbf{Q}_i \mathbf{R}_i$ be constructed via local HOUSEHOLDER-QR (see Algorithm 3.1) for each $i = 1, \dots, p$, in parallel. Then, $\mathbf{X} = \mathbf{QR}$ for the complete data matrix can be constructed exactly, such that $\mathbf{Q} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p) \mathbf{Q}_M$, and $\mathbf{R} = \mathbf{R}_M$, where $\mathbf{R}_{stack} = \mathbf{Q}_M \mathbf{R}_M$ via another HOUSEHOLDER-QR on $\mathbf{R}_{stack} = (\mathbf{R}_1^T | \dots | \mathbf{R}_p^T)^T$ gathered from all workers. The above DISTRIBUTED HOUSEHOLDER-QR has a computational time complexity of $O(\frac{n}{p} d^2)$, with a communicated data volume of $(\frac{d(d+1)}{2})$ elements by each worker.*

Proof.

$$\mathbf{X} = \begin{pmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \\ \cdot \\ \cdot \\ \mathbf{X}_p \end{pmatrix} = \begin{pmatrix} \mathbf{Q}_1 \mathbf{R}_1 \\ \mathbf{Q}_2 \mathbf{R}_2 \\ \cdot \\ \cdot \\ \mathbf{Q}_p \mathbf{R}_p \end{pmatrix} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p) \begin{pmatrix} \mathbf{R}_1 \\ \mathbf{R}_2 \\ \cdot \\ \cdot \\ \mathbf{R}_p \end{pmatrix}$$

Let us define, $\mathbf{R}_{stack} = \begin{pmatrix} \mathbf{R}_1 \\ \mathbf{R}_2 \\ \cdot \\ \cdot \\ \mathbf{R}_p \end{pmatrix} = \mathbf{Q}_M \mathbf{R}_M,$

via HOUSEHOLDER-QR step in Algorithm 3.1 (or Theorem 3.1). Then,

$$\mathbf{X} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p) \mathbf{R}_{stack} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p) \mathbf{Q}_M \mathbf{R}_M$$

Also, it is given that $\mathbf{X} = \mathbf{Q}\mathbf{R}$ via HOUSEHOLDER-QR on complete matrix \mathbf{X} . Hence, $\mathbf{Q} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p) \mathbf{Q}_M$, is the orthogonal matrix, and, $\mathbf{R} = \mathbf{R}_M$ is the upper triangular matrix.

Time complexity and Communication volume. For a given local data $\mathbf{X}_i \in \mathbb{R}^{\hat{n} \times d}$, where, $n = \hat{n}p$, each $\mathbf{X}_i = \mathbf{Q}_i \mathbf{R}_i$ at i -th parallel worker is computed via local HOUSEHOLDER-QR (as per Algorithm 3.1, and Theorem 3.1). From Theorem 3.2, each local HOUSEHOLDER-QR takes $O(\hat{n}d^2 - d^3/3)$, in parallel for all the workers. Subsequently, $\mathbf{R}_{stack} = \mathbf{Q}_M \mathbf{R}_M$ is performed via master HOUSEHOLDER-QR in time $O(\times pd \times d^2 - d^3/3)$, where, $\mathbf{R}_{stack} \in \mathbb{R}^{pd \times d}$ is obtained by *gathering* (communicating) local upper-triangular matrices $\mathbf{R}_i \in \mathbb{R}^{d \times d}$, i.e., $\left(\frac{d(d+1)}{2}\right)$ elements from each parallel worker $i = 1, \dots, p$ to the master ($i = 1$). Hence, total computation time for DISTRIBUTED HOUSEHOLDER-QR is $O(\hat{n}d^2 + pd^3 - 2d^3/3)$ or $O\left(\frac{n}{p}d^2\right)$ for $\hat{n} \gg d$ (i.e. $n \gg pd$).

It is worth noticing that the above computational time is dominant by local HOUSEHOLDER-QR as observed in Figure 3.2 (a). \square

Relation to TSQR [67]. The construction of DISTRIBUTED HOUSEHOLDER-QR is inspired from the parallel Tall-Skinny QR (TSQR) algorithm [67]. Unlike the binary reduction tree in TSQR, we use a simple two-level organisation where the first level involves local HOUSEHOLDER-QR on each participating worker in parallel (Step 2 in Algorithm 3.3), followed by a second level which computes HOUSEHOLDER-QR at the master (Step 7 in Algorithm 3.3). Since the communication overhead is negligible as demonstrated in Figure 3.2, the choice of implementation of the DISTRIBUTED HOUSEHOLDER-QR is justified. Moreover, designing such a framework supports decentralized machine learning applications with workers (users) on the edge of distributed network. Here, every worker has same and direct access to the master compared to the neighbors in TSQR’s binary reduction scheme. Finally, we would like to highlight that established packages such as ScaLAPACK and Elemental [68] use different matrix blocking and collectives for performing Householder QR on multiple worker nodes via All-reduction scheme. In contrast, DISTRIBUTED HOUSEHOLDER-QR employs a reduction scheme similar to TSQR [67] where the global \mathbf{R} resides on only one node (master) rather than being shared across all the workers. Due to relatively lower synchronisation cost, DISTRIBUTED HOUSEHOLDER-QR similar to TSQR [67] obtains a performance benefit over ScaLAPACK [69].

Corollary 3.1 (Distributed Multiply-QC). *Let $\mathbf{c} = (\mathbf{c}_1^T | \dots | \mathbf{c}_p^T)^T \in \mathbb{R}^n$, where, $\mathbf{c}_i \in \mathbb{R}^{\hat{n}}$ be some local vector at parallel worker with local data matrix \mathbf{X}_i , $i = 1, \dots, p$, where $\hat{n} \gg d$, and, $n = p\hat{n}$. Let, orthogonal matrices \mathbf{Q}_M , and \mathbf{Q}_i , $i = 1, \dots, p$ be constructed via DISTRIBUTED HOUSEHOLDER-QR as per Theorem 3.4 such that $\mathbf{Q} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p)\mathbf{Q}_M$. Then, the reflected vector, $\mathbf{Q}^T \mathbf{c}$ (or $\mathbf{Q}\mathbf{c}$) can be constructed exactly by making $(p + 1)$ calls to MULTIPLY-QC (see Step 2 in Algorithm 3.1) such that $\mathbf{Q}^T \mathbf{c} = \mathbf{Q}_M^T ((\mathbf{Q}_1^T \mathbf{c}_1)^T | \dots | (\mathbf{Q}_p^T \mathbf{c}_p)^T)^T$ or, $\mathbf{Q}\mathbf{c} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p)\mathbf{Q}_M(\mathbf{c}_1^T | \dots | \mathbf{c}_p^T)^T$. The above DISTRIBUTED MULTIPLY-QC has a computational time complexity of $O(\frac{n}{p}d + pd^2)$, with a communicated data volume of (d) elements by each worker.*

Proof. From Theorem 3.4, for $\mathbf{X} = (\mathbf{X}_1^T | \dots | \mathbf{X}_p^T)^T$, its corresponding orthogonal matrix $\mathbf{Q} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p)\mathbf{Q}_M$. Hence,

$$\mathbf{Q}^T = \mathbf{Q}_M^T \text{diag}(\mathbf{Q}_1^T, \dots, \mathbf{Q}_p^T)$$

For a given vector $\mathbf{c} \in \mathbb{R}^n$, $\mathbf{Q}^T \mathbf{c}$ via MULTIPLY-QC (Step 2 in Algorithm 3.1) can be equivalently computed from $\mathbf{c} = (\mathbf{c}_1^T | \dots | \mathbf{c}_p^T)^T$ comprising local vector $\mathbf{c}_i \in \mathbb{R}^{\hat{n}}$, where, $i = 1, \dots, p$, as follows

$$\mathbf{Q}^T \mathbf{c} = \mathbf{Q}_M^T \text{diag}(\mathbf{Q}_1^T, \dots, \mathbf{Q}_p^T) \mathbf{c} = \mathbf{Q}_M^T \begin{pmatrix} \mathbf{Q}_1^T \mathbf{c}_1 \\ \mathbf{Q}_2^T \mathbf{c}_2 \\ \cdot \\ \cdot \\ \mathbf{Q}_p^T \mathbf{c}_p \end{pmatrix}$$

In DISTRIBUTED MULTIPLY-QC algorithm, the above is implemented as follows. Each worker, $i = 1, \dots, p$, computes its local reflected vectors $\mathbf{Q}_i^T \mathbf{c}_i \in \mathbb{R}^d$ via MULTIPLY-QC (refer Step 2 in Algorithm 3.1) in parallel with time $O(2\hat{n}d)$ as shown in Theorem 3.2. Once these local reflected vectors, each of size d elements are *gathered* (communicated) from each worker to the master, a stacked vector $\left((\mathbf{Q}_1^T \mathbf{c}_1)^T | \dots | (\mathbf{Q}_p^T \mathbf{c}_p)^T \right)^T \in \mathbb{R}^{pd \times d}$ is constructed. Then, a master MULTIPLY-QC is applied on this stacked vector using \mathbf{Q}_M^T in time $O(2 \times pd \times d)$, i.e., $O(2pd^2)$. Hence, total computation time of DISTRIBUTED MULTIPLY-QC is $O(2\hat{n}d + 2pd^2)$, i.e., $O(\frac{n}{p}d + pd^2)$, since $n = \hat{n}p$. \square

Privacy. From Theorem 3.4, and Corollary 3.1, it can be observed that the DISTRIBUTED LMS-QR in Algorithm 3.3 can locally compute $(\mathbf{R}_i, \mathbf{Q}_i^T \mathbf{y}_i)$ for each worker without the need to share its original data \mathbf{X}_i to a centralized node or any of the neighbors. By maintaining \mathbf{Q}_i privately, the algorithm avoids any other worker (or master) to reconstruct \mathbf{X}_i accurately. It is to be noted that in our experiments, *master* is designated via Message Passing Interface protocol.

Algorithm 3.3 DISTRIBUTED LMS-QR($p, \mathbf{X}, \mathbf{y}, \text{params}$)

Data: A scalar $p > 0$ parallel workers (cores or users), a matrix $\mathbf{X} = (\mathbf{X}_1^T | \dots | \mathbf{X}_p^T)^T$, $\mathbf{X}_i \in \mathbb{R}^{\frac{n}{p} \times d}$, a vector $\mathbf{y} = (\mathbf{y}_1^T | \dots | \mathbf{y}_p^T)^T$, $y_i \in \mathbb{R}^{\frac{n}{p}}$, a list of LMS parameters, params

Result: A vector of model coefficients, $\mathbf{w} \in \mathbb{R}^d$

```
// ( $\mathcal{V}, \mathbf{R}$ ) := DISTRIBUTED HOUSEHOLDER-QR( $\mathbf{X}$ ), see Theorem 3.4
1 for every worker  $i \in \{1, 2, \dots, p\}$  do
2   ( $\mathcal{V}_i, \mathbf{R}_i$ ) := HOUSEHOLDER-QR( $\mathbf{X}_i$ ) // see Theorem 3.1
3    $\mathbf{R}_i \leftarrow \mathbf{R}_i[0 : d, :]$  //  $d \times d$  triangular block
4    $\mathbf{R}_{stack} := \text{GATHER}(\mathbf{R}_i, \text{root} = 0)$  //  $\mathbf{R}_{stack} = \text{vstack}(\mathbf{R}_1, \dots, \mathbf{R}_p)$  at Master
5 end
6 if  $i == 1$  then // check for Master
7   ( $\mathcal{V}_M, \mathbf{R}_M$ ) := HOUSEHOLDER-QR( $\mathbf{R}_{stack}$ ) // see Theorem 3.1
8    $\mathbf{R}_M \leftarrow \mathbf{R}_M[0 : d, :]$  //  $d \times d$  triangular block
9 end
//  $\mathcal{V} := [\mathcal{V}_1, \dots, \mathcal{V}_p, \mathcal{V}_M]$  is never centralized or shared
//  $\mathbf{Q} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p)\mathbf{Q}_M$ , and,  $\mathbf{R} = \mathbf{R}_M$ , see Theorem 3.4
//  $\bar{\mathbf{y}} := \text{DISTRIBUTED MULTIPLY-QC}(\mathcal{V}, \mathbf{y}, \text{'T'})$ , see Corollary 3.1
10 for every worker  $i \in \{1, 2, \dots, p\}$  do
11    $\bar{\mathbf{y}}_i := \text{MULTIPLY-QC}(\mathcal{V}_i, \mathbf{y}_i, \text{'T'})$  // implicit  $\mathbf{Q}_i^T \mathbf{y}_i$ , see Algorithm 3.1
12    $\bar{\mathbf{y}}_i \leftarrow \bar{\mathbf{y}}_i[0 : d]$  // select top  $d$  elements
13    $\bar{\mathbf{y}}_{stack} := \text{GATHER}(\bar{\mathbf{y}}_i, \text{root} = 0)$  //  $\bar{\mathbf{y}}_{stack} = \text{vstack}(\bar{\mathbf{y}}_1, \dots, \bar{\mathbf{y}}_p)$  at Master
14   if  $i == 1$  then // check for Master
15      $\bar{\mathbf{y}}_M := \text{MULTIPLY-QC}(\mathcal{V}_M, \bar{\mathbf{y}}_{stack}, \text{'T'})$  // implicit  $\mathbf{Q}_M^T \bar{\mathbf{y}}_{stack}$ 
16      $\bar{\mathbf{y}}_M \leftarrow \bar{\mathbf{y}}_M[0 : d]$  // select top  $d$  elements
17   end
18 end
19  $\bar{\mathbf{y}} := \bar{\mathbf{y}}_M$  //  $\bar{\mathbf{y}} = \mathbf{Q}^T \mathbf{y} = \mathbf{Q}_M^T ((\mathbf{Q}_1^T \mathbf{y}_1)^T | \dots | (\mathbf{Q}_p^T \mathbf{y}_p)^T)^T$ 

// Solving LMS
20 if  $i == 1$  then // check for Master
21    $\mathbf{w} := \text{LMS}(\mathbf{R}, \bar{\mathbf{y}}, \text{params})$  // run LMS solver at Master
22   BROADCAST( $\mathbf{w}, \text{root} = 0$ ) // every worker receives the global model
23 end
24 return  $\mathbf{w}$ 
```

3.5 Experiment and Results

In this section we perform extensive empirical analysis for the LMS-QR (Algorithm 3.2). Recall, HOUSEHOLDER-SKETCH (Algorithm 3.1) on the input data (\mathbf{X}, \mathbf{y}) generates memory-efficient $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ to accurately solve, and accelerate common LMS solvers in scikit-learn library.

We present detailed experimental setup, and extensive discussion of results below.

3.5.1 Hardware Description

We used Google Colab to run our experiments with the above LMS-QR algorithms via Python3 Google Compute Engine running on Intel Xeon CPU @ 2.20GHz and 25 GB RAM. For *distributed* experiments, we used the Anaconda Python distribution and MPI for Python (mpi4py) package on the Texas A&M University *Ada* computing cluster of Intel Xeon CPU @ 2.5GHz.

3.5.2 Experimental Setup

We evaluated the performance of LMS-QR algorithm on regression models such as Linear Regression, Ridge, LASSO, and Elastic-Net in scikit-learn library [70]. To implement Algorithm 3.1, we use `LAPACK.dgeqrf()`, and `LAPACK.dormqr()` subroutines for HOUSEHOLDER-QR, and MULTIPLY-QC, respectively.

We used following datasets for evaluation, and fair comparison of LMS-QR performance with the default LMS solvers (with cross validation), and with Fast Caratheodory coresets based LMS-BOOST [45]. **(i)** Synthetic data (X, y) comprising uniform random entries in $[0, 100)$ for sequential experiments. **(ii)** 3D Road network[71] dataset with $n = 434, 874$ data samples. We selected $d = 2$ feature attributes (*longitude, latitude*) as per [45] to predict *height (in metres)*. **(iii)** Individual household electric power consumption [72] dataset with $n = 2, 075, 259$ data samples. We selected $d = 8$ attributes as per [45] to predict the *house price*.

We used synthetic dataset with uniform random entries in $(-100, 100)$ with zero-centering for evaluating the distributed performance of RIDGE-QR as our case study on a cluster of $p = \{2, 4, 8, 16\}$ workers (computing cores). However, it is to be noted that DISTRIBUTED LMS-QR technique is easily applicable to other solvers. Linear algebra was handled by LAPACK/BLAS, through the Intel Math Kernel Library. We ensure that each worker (core) was assigned from a different node in the cluster to ensure distributed memory with MKL threads per core limited to 1. Each test was performed 20 times, and the best result was chosen.

3.5.3 Results and Discussions

Here, we discuss the performance of the Householder sketch with respect to training time for both sequential and distributed implementations, scalability with increasing number of workers, and numerical stability.

3.5.3.1 Sequential Training Time

LMS-QR works with memory-efficient $(\mathbf{R}, \mathbf{Q}^T \mathbf{y})$ with just d rows in \mathbf{R} compared to LMS with n rows in \mathbf{X} of the original data (\mathbf{X}, \mathbf{y}) , and $(d^2 + 1)$ rows in the reduced matrix from the Fast Caratheodory coresets in LMS-BOOST [45]. Construction of \mathbf{R} via Householder transformation in LMS-QR, and the reduced matrix via Fast Caratheodory set for LMS-BOOST take time that is linear in n , and quadratic in d . Figure 3.1 (a)-(i) depicts the sequential training computation time on synthetic dataset for the LMS-QR, and compares them with respective LMSCV [70], and LMS-BOOST [45], where, $\text{LMS} = \{\text{RIDGE}, \text{LASSO}, \text{ELASTIC}\}$. From Figure 3.1 (a)-(c), we observe that for various feature dimensions $d = \{3, 5, 7\}$ and data sizes $n = \{240K, \dots, 24M\}$, LMS-QR accelerates the running time of default LMSCV by 100x for both LASSO, and ELASTIC, and by 400x for RIDGE. In comparison with LMS-BOOST, RIDGE-QR outperforms RIDGE-BOOST by 10x for $d = 7$ and various n in Figure 3.1(a). Moreover, LASSO/ELASTIC/LINREG-QR runs upto 100x faster when $n < 2.4M$ and upto 10x faster when $n > 2.4M$ than LASSO/ELASTIC/LINREG-BOOST for $d = 7$ in Figure 3.1 (b)(c)(j). In Figure 3.1 (d)-(f), we demonstrate the running time of LMS-QR for $n = 24M$ and various $d = \{3, 5, 7, 10, 25, 50\}$. We observe that LMSCV, and LMS-BOOST [45] with running time $O(nd^2 + \log n \times d^8)$, run out of memory for $d = \{25, 50\}$ while LMS-QR could handle growing feature dimension as shown. Hence, we demonstrate that LMS-QR can easily **handle big datasets** with increasing data size n and feature dimension d while enjoying the **fastest running times** compared to LMSCV and LMS-BOOST. For various size of hyper-parameter set for cross validation, $|\mathcal{A}| = \{50, 100, 200, 300\}$, for cross validation, Figure 3.1 (g)-(i) depict LMS-QR to be consistently faster than LMS-CV and LMS-BOOST. We observe similar trend for the real datasets in Figure 3.1 (k)-(l).

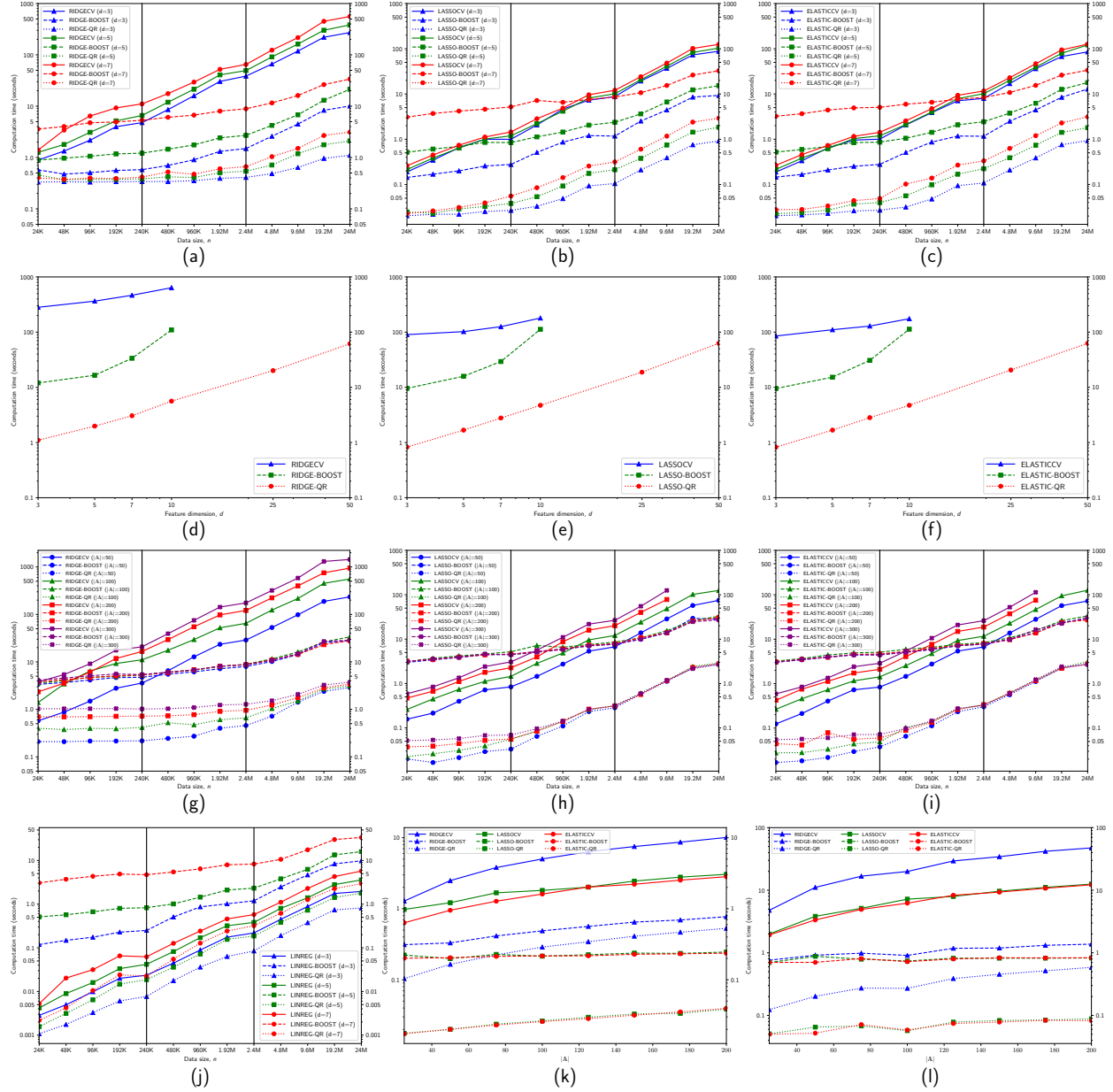


Figure 3.1: Sequential training time {RIDGE, LASSO, ELASTIC} (a)-(c): vs data size, n with feature dimension, $d = \{3, 5, 7\}$, (d)-(f): vs d with $n = 24M$, (g)-(i): vs n with hyper-parameter size $|\mathbb{A}|$, (j): LINREG vs n , (k): vs $|\mathbb{A}|$ for 3D Road Network, (l): vs $|\mathbb{A}|$ for Household Power Consumption. Cross validation folds, $|m| = 3$ for synthetic datasets (a)-(j) and $|m| = 2$ for real datasets (k)-(l). Reprinted with permission from [4].

3.5.3.2 Distributed Training Time

We evaluate the performance of DISTRIBUTED RIDGE-QR as a case study to demonstrate the effectiveness of the distributed implementation. From Algorithm 3.3, DISTRIBUTED RIDGE-QR can be split into two main stages: (**Stage 1**) DISTRIBUTED HOUSEHOLDER-QR, and (**Stage 2**) DISTRIBUTED MULTIPLY-QC, followed by solving the Ridge regression. Figure 3.2 (a)-(b) demonstrate the above two stages on a $10M \times 10$ synthetic dataset.

Stage 1 running time can be further broken into three main components: local HOUSEHOLDER-QR time, master HOUSEHOLDER-QR time, and communication (`gather`) time as illustrated in Figure 3.2 (a). Since each worker performs its local HOUSEHOLDER-QR on partitioned data of size $\frac{n}{p} \times d$ (Theorem 3.4), we observe it to be empirically more computationally dominant than the master HOUSEHOLDER-QR that works on gathered matrices of size $pd \times d$. We show that on doubling the number of parallel workers, i.e. $p = \{2, 4, 8, 16\}$ workers, the time to calculate the local HOUSEHOLDER-QR is reduced by half, i.e. $\{1.1734, 0.5405, 0.2508, 0.1233\}$ seconds respectively demonstrating it is **fully parallelized**. In comparison, master HOUSEHOLDER-QR computation time, and communication time to `gather` merely $d(d+1)/2$ elements is nearly negligible as depicted in Figure 3.2 (a).

Next, in Figure 3.2(b), we observe the timings for **Stage 2** in DISTRIBUTED RIDGE-QR. The running time of this stage can be split into following main components: DISTRIBUTED MULTIPLY-QC time to compute $\mathbf{Q}^T \mathbf{y}$, and time to solve the ridge regression via popular RIDGE solver [70]. We observe in Figure 3.2(b) that DISTRIBUTED MULTIPLY-QC time is computationally dominant as expected from Corollary 3.1, and can be **fully parallelized** with reported timings of $\{0.2782, 0.1232, 0.06012, 0.0277\}$ seconds on $p = \{2, 4, 8, 16\}$ workers, respectively. Relatively, we also observe that solving RIDGE regression with $\mathbf{R}\mathbf{R}^T \in \mathbb{R}^{d \times d}$, is nearly negligible in computation time across various choices of p . Moreover, the communication required in this stage involves a `gather` of $\mathbf{Q}_i^T \mathbf{y}_i \in \mathbb{R}^d$ at the master, and broadcasting w to all workers, which is negligible.

Finally, in Figure 3.2 (c) we depict the percentage of total running time for DISTRIBUTED RIDGE-QR that is spent on computation and communication components of **Stage 1**, and **Stage 2**

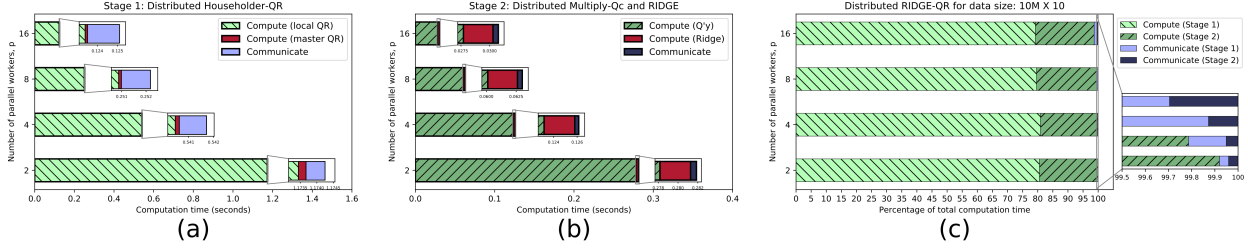


Figure 3.2: Breakdown of DISTRIBUTED RIDGE-QR training time with zoomed insets depicting communication time (a): Stage 1: DISTRIBUTED HOUSEHOLDER-QR, (b): Stage 2: DISTRIBUTED MULTIPLY-QC and RIDGE, (c): Combined percentage. Reprinted with permission from [4].

for various choices of p . We observe the **communication overhead** to be **nearly negligible**, and local HOUSEHOLDER-QR to be the most computationally dominant component. The latter can be parallelized significantly across multiple workers. We also observe that for any given dataset size and choice of p , as long as the parallelizable components, namely, local HOUSEHOLDER-QR, and DISTRIBUTED MULTIPLY-QC remain as the most computationally dominant components, the algorithm will continue to scale as per Amdahl’s Law [73]. So for big datasets, we expect good scalability on large number of p workers as discussed next.

3.5.3.3 Scalability

We discuss parallel speedup under strong scaling scenario wherein the overall problem size stays fixed but the number of parallel workers p is doubled. Parallel speedup is defined as the ratio of the running time for the sequential algorithm to that of the corresponding parallel algorithm on p workers. When speedup of any parallel algorithm is p , it exhibits ideal speedup with linear scalability. Figure 3.3 (a)-(c) demonstrates the parallel speedup of DISTRIBUTED RIDGE-QR, and compares it with popular distributed technique ADMM [74], here, RIDGE-ADMM that uses original data (\mathbf{X}, \mathbf{y}) on RIDGE [70], for \mathbf{w} -update step. In Figure 3.3 (a)-(c), we observe that with larger data sample sizes $n = \{500K, 1M, 2M\}$, DISTRIBUTED RIDGE-QR along with its dual equivalent DISTRIBUTED KERNELRIDGE-QR (linear kernel) exhibit **almost linear scalability** across p by approaching the **ideal speedup**. This is attributed to its negligible communication

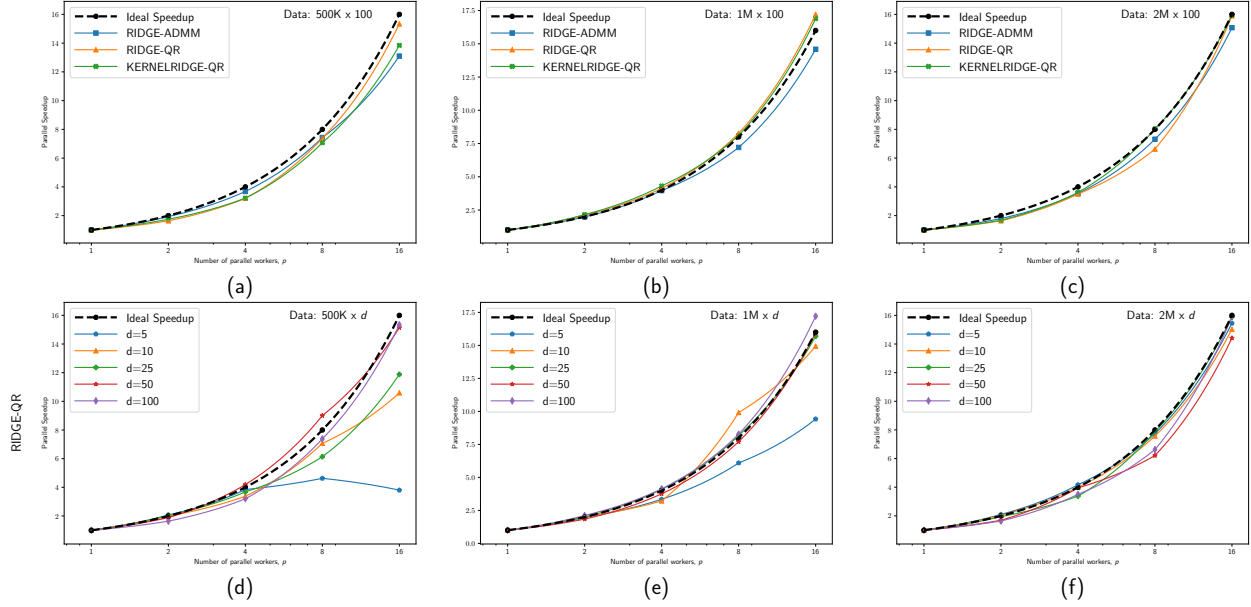


Figure 3.3: Comparing scalability of various algorithms to solve RIDGE problem on synthetic datasets of size $n \times d$ (a)-(c): Various $n = \{500K, 1M, 2M\}$, $d = 100$, (d)-(f): DISTRIBUTED RIDGE-QR with $d = \{5, 10, 25, 50, 100\}$. Reprinted with permission from [4].

overhead, and almost fully parallelizable computational components as discussed earlier. It is also worth noticing from Figure 3.3 (d)-(f) that as the feature dimension increases from $d = 5$ to $d = 10$ in each plot, DISTRIBUTED RIDGE-QR speedup tends to be more linear for high p , thereby capable of showing high scalability for datasets with large feature size on large number of parallel workers. In Figure 3.4 (a) we observe that solving RIDGE regression problem with linear kernel in *dual* form using KERNELRIDGE-QR has the same computation time as that of solving the *primal* form using RIDGE-QR for various sequential and distributed settings, $p = \{1, 2, 4, 8, 16\}$. Moreover, Householder-Sketch based sequential ($p = 1$) and distributed ($p = \{2, 4, 8, 16\}$) algorithms run much faster than the corresponding implementation of iterative RIDGE-ADMM algorithms on the original data.

3.5.3.4 Numerical Stability

Figure 3.4 (b) shows that DISTRIBUTED RIDGE-QR is numerically stable to rounding errors with increasing number of workers p , while ADMM being an iterative learning algorithm is more

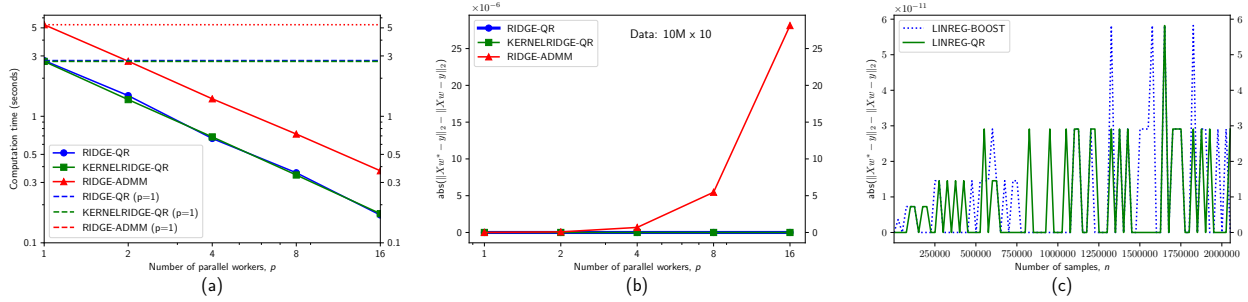


Figure 3.4: (a)-(b): Comparing DISTRIBUTED RIDGE-QR / KERNELRIDGE-QR (linear kernel), and RIDGE-ADMM for $10M \times 10$ synthetic data based on (a) Computation time (b) Accuracy, w^* is solution from scikit-learn RIDGE, (c) Accuracy of LINREG-QR and LINREG-BOOST on Household Power Consumption dataset ($\sim 2M \times 8$), w^* is solution from scikit-learn LINEAR-REGRESSION. Reprinted with permission from [4].

susceptible. For the sequential implementation on linear regression problem, [45] had demonstrated much better numerical stability of LINREG-BOOST compared to $(\mathbf{X}^T \mathbf{X})^{-1}$. Here, Figure 3.4 (c) presents the numerical stability of LINREG-QR with scaled factor of 10^{-11} to demonstrate similar trend to that of LINREG-BOOST [45].

3.6 Summary

In this chapter, we demonstrate that Householder transformation generates a theoretically accurate sketch that is relatively more memory-efficient and computationally faster than the LMS-BOOST algorithm in [45] to accurately solve LMS problems. In principle, Householder sketch accelerates common LMS solvers in scikit-learn library up to 100x-400x, and outperforms the strong baseline LMS-BOOST by 10x-100x with similar numerical stability. The distributed implementation achieves linear scalability with negligible communication overhead for large sample size and dimension across multiple worker nodes. We believe that the above results are valuable for the community to realize not to disregard classical techniques so quickly, rethink how some comparisons are done, identify common misconceptions, and reassess what the most appropriate algorithms for certain problems are. In subsequent chapters, we will incorporate Householder sketch to construct memory-efficient and communication-efficient frameworks for scalable machine learning.

4. MEMORY-EFFICIENT FRAMEWORK FOR DISTRIBUTED MACHINE LEARNING ¹

In this chapter, we use our knowledge of relaxed synchronization and distributed Householder sketches from previous chapters to perform memory-efficient modeling and training of a machine learning problem such as Support Vector Machines (SVM) across multiple workers. We will show how optimal synchronization period introduced in Chapter 2 is related to optimal step size or learning rate. The ideas presented in this chapter can be applied to any convex optimization problem or other machine learning problems with iterative solvers such as dual ascent and gradient descent. With the influx of huge amount of digital data from sensors, social media, mobile devices and online transactions, it has become increasingly challenging to store, process and analyze data for predictive analytics.

4.1 Introduction

Machine learning is at the core of solving real-world challenges in sectors like energy, transportation, finance, business analytics, health-care and manufacturing. Support Vector Machine (SVM) fall under the realm of supervised machine learning wherein a mathematical model is trained on a prior dataset and associated class labels. SVMs are commonly used for classification and regression analysis. In binary SVM classification, the goal is to optimally compute a hyperplane that maximally separates the two classes. In regression, one attempts to find a function that is an optimal fit to the data with minimum deviation in the function value. Real-world applications give rise to datasets that have a non-linear structure for which kernel SVMs are used. In linear SVM problems, the training data is used in its original feature space, thereby, enabling the adoption of coordinate gradient methods [75] to achieve the above described optimization goal. In contrast, for non-linear SVM problems the data is transformed to a higher dimensional space which

¹A part of this chapter is reprinted with permission from “Distributed QR Decomposition Framework for Training Support Vector Machines” by Jyotikrishna Dass, V. N. S. Prithvi Sakuru, Vivek Sarin, and Rabi N. Mahapatra, 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Copyright ©2017 IEEE.

A part of this chapter is reprinted with permission from “Fast and Communication-Efficient Algorithm for Distributed Support Vector Machine Training” by Jyotikrishna Dass, Vivek Sarin, and Rabi N. Mahapatra, 2019 IEEE Transactions on Parallel and Distributed Systems (TPDS), Copyright © 2019, IEEE.

represents the feature space. Subsequently the classifier can be learned by simply computing the inner products between all pairs of datapoints in the feature space without explicitly calculating their transformed coordinates. This is commonly referred to as the kernel trick [65]. Since the higher dimensional coordinates of the datapoints are not explicitly computed, applying coordinate gradient methods is infeasible for non-linear SVMs.

4.1.1 Motivation

Mathematically, SVM is a constrained convex optimization problem with a quadratic objective function. The Hessian of the quadratic function is the kernel matrix which poses major computational and scalability issues. With the growing data sizes, solving large scale machine learning problems using SVM is a tedious task both in terms of computational cost as well as memory resources. Computation and storage of the kernel matrix grows as $O(n^2)$ for n datapoints, making it impractical to compute the matrix for large sized problems.

4.1.2 Contributions

We propose a QR decomposition framework (QRSVM) to efficiently solve large-scale kernel SVM problems with guaranteed stability of dual ascent method. We use state of the art kernel approximation technique that is memory efficient unlike Singular Value Decomposition (SVD). The low-rank approximation of the kernel matrix is represented in a separable form which makes it fit to be used in the proposed QRSVM framework. Rather than proceeding with the approximated kernel matrix directly, we further decompose it using QR factorization. This leads to memory-efficient representation of the otherwise dense hessian matrix in SVM problem. The subsequent representation is sparse that it can be block-partitioned for parallel SVM training. The proposed QRSVM scales linearly with the data size which further motivates towards a distributed framework that can work on data that has been partitioned across a cluster of computing nodes. Within this, we first present a distributed QR decomposition technique and then parallelize the iterative update steps of the dual ascent method to solve the parallel kernel SVM problem. We also propose an optimal step size for fast convergence of the dual ascent method.

4.2 Related Work

Various decomposition techniques like Vapnik's chunking [76], Osuna's decomposition [77] and Platt's SMO [14] have made respectable contribution to addressing these issues. To efficiently solve large-scale classification problems, research community nowadays is more focused on modeling it as a linear SVM problem which is a simplified optimization framework [78]. In the linear SVM problems, the training data is used in its original feature space, thereby, enabling the adoption of coordinate gradient methods to solve the optimization problem. In comparison, for the non-linear SVM problems, the data is transformed to higher dimensional space by using kernel trick. Since the higher dimensional coordinates are not explicitly computed, applying coordinate gradient methods for non-linear kernel problems is infeasible. In coordinate gradient methods, a single data point contributes to the gradient shift of the objective function iteratively and hence, they have very low per-iteration computation cost. However, these methods have slow convergence rate when compared to the orthodox gradient methods like dual ascent [79]. Zhang et al. [80], Shalev-Shwartz et al. [81], Bottou [82] proposed variations of stochastic gradient descent on the primal SVM form. Joachims [83] proposed the cutting plane method while Smola et al. [84] used bundle methods on dual SVM. Hsieh et al. [75] proposed the dual coordinate descent (DCD) method and showed that it outperformed other dual methods. LIBLINEAR [85] is a state of the art library for solving linear SVM classification (SVC) problems. It is implemented using the DCD method. However, this method was really slow and not completely stable for non-document datasets especially with low dimensions [86].

For solving kernel SVM for non-linear decision boundaries, efficient sequential algorithms have been proposed. The most popular of such algorithms is SMO [14] which now finds its implementation in widely used SVM solver tool LIBSVM [87]. Due to the sequential nature of SMO algorithm, single machine kernel SVM solvers suffer from limited scalability, thereby making those unfit for training big-scale datasets. With growing data sizes, it has become utmost necessary to design parallel and distributed algorithms to train the kernel SVMs.

The state of the art parallel algorithms which have been proposed to solve kernel SVM prob-

lems are PSVM [12] and P-pack SVM [88]. PSVM, in fact, has limited scalability due to its quadratic dependence on the sample size. Moreover, PSVM works with kernel matrix approximation using Incomplete Cholesky Factorization (ICF) which lacks theoretical error bounds. Many kernel approximation techniques like Nyström [89], LLSVM [90] and random Fourier features [91] have also been explored to reduce the problem dimension. Such methods produce low-rank approximations of the kernel matrix and have been used to reduce the kernel SVM problem into a linear SVM problem. P-packSVM computes the primal form of SVM using a stochastic gradient descent method wherein the gradient is approximated at a single sample [88]. Since the primal SVM problem can be prohibitively large while its Wolfe dual problem is considerably smaller, the convergence rate of the primal solver is slower than the dual solver [92].

4.3 Support Vector Machines

In this section, we introduce the SVM problem and its formulation for both linear and non-linear classification. The goal of a typical Support Vector Machine for the binary classification problem is to determine an optimal hyperplane that maximally separates the two classes. The hyperplane rests on a set of *support points* that determine the shape of the classifier.

4.3.1 Formulation

For classifier training, one is given a training dataset $S = \{(\mathbf{x}_i, y_i), \forall i = 1, \dots, n\}$ with n number of samples. Each input data point $\mathbf{x}_i \in \mathbb{R}^d$ (\mathbb{R}^d is the d -dimensional input space) and $y_i \in \{-1, 1\}$ is the corresponding data label (or class).

An ℓ_2 -regularized primal version [65] of this problem is

$$\min_{\mathbf{w}} \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{i=1}^n \xi_i(\mathbf{w}; \mathbf{x}_i; y_i), \quad (4.1)$$

where $\mathbf{w} \in \mathbb{R}^d$ represents the normal to the hyperplane separating the data points and scalar $C > 0$ is the penalty parameter that determines the trade-off between margin maximization and training error minimization. The term $\xi_i(\mathbf{w}; \mathbf{x}_i; y_i)$ represents the squared *hinge* loss function associated with the optimization problem. A scalar *bias* term, $b \in \mathbb{R}$ is typically associated with \mathbf{w} represent-

ing the separating hyperplane, $\mathbf{w}^T \mathbf{x} + b = 0$.

The *dual* formulation of Equation (4.1) is:

$$\min_{\boldsymbol{\alpha}} \frac{1}{2} \boldsymbol{\alpha}^T \mathbf{Z} \boldsymbol{\alpha} + \mathbf{e}^T \boldsymbol{\alpha}, \quad (4.2)$$

$$\text{subject to } L \leq \alpha_i \leq U,$$

where, $\mathbf{Z} = (\mathbf{G} + \mathbf{D}) \in \mathbb{R}^{n \times n}$ is a dense and positive definite, $\mathbf{e} = -\mathbf{1}_n$, L is the scalar lower bound of each lagrangian multiplier α_i and U is its scalar upper bound. $\mathbf{G} = \{y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j)\}$, where, $\kappa()$ represents the Mercer kernel function.

For SVM, \mathbf{G} is derived from the kernel matrix, $\mathbf{K} = \{\kappa(\mathbf{x}_i, \mathbf{x}_j), \forall i, j = 1, \dots, n\}$ which is a positive semi-definite matrix. For linear-SVM, the kernel function $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$. The diagonal matrix \mathbf{D} , lower bound L and upper bound U are dependent on the type of loss function associated with the SVM problem.

For L2-SVM with

$$\ell_2\text{-loss} : \xi_i(\mathbf{w}; \mathbf{x}_i; y_i) = \max(0, 1 - y_i \mathbf{w}^T \mathbf{x}_i)^2,$$

we have $\mathbf{D} = (2C)^{-1} \mathbf{I}_n$, $L = 0$ and $U = \infty$.

There are two major types of SVMs based on the nature of the decision boundary to be learnt for any given dataset. Linear SVMs are used for finding the linearly separable hyperplane whereas Kernel (or non-linear) SVMs are used to learn complex and non-linear decision boundaries between the two classes present in the dataset. For kernel (or non-linear) SVMs, it is advantageous to solve the problem using its *dual* form to leverage the benefits of the kernel trick [93]. In addition, by using the dual formulation, the loss function vanishes from the objective function making its optimization simpler.

For a kernel (or non-linear) SVM problem, the kernel function $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$ is a similarity measure between a pair of data points, \mathbf{x}_i and \mathbf{x}_j in the dataset which has a non-linear

decision boundary. Here, $\phi()$ represents a mapping that transforms the original data point, \mathbf{x}_i from input space to the Reproducing Kernel Hilbert Space (RKHS) where it can be linearly separable. It should be noted that $\phi()$ need not be explicitly available as one has a representation of the above defined kernel function $\kappa(\mathbf{x}_i, \mathbf{x}_j)$. Some examples of non-linear kernel functions are polynomial kernel, radial basis function, etc.

In contrast, the kernel matrix \mathbf{K} for linear SVM is directly separable in terms of original data-points, i.e., $\mathbf{K} = \mathbf{X}\mathbf{X}^T$, where $\mathbf{X} = \{\mathbf{x}_i \in \mathbb{R}^d, i = 1, \dots, n\}$. Our goal is to solve the non-linear problems since most of the datasets that exist have non-linear decision boundaries inherently.

We focus on the radial basis function (RBF) as the non-linear kernel function, i.e.,

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2),$$

which transforms the data to infinite dimensional space. Unlike the linear SVM, the kernel matrix \mathbf{K} for non-linear problems is not trivially separable. Moreover, \mathbf{K} is associated with $O(n^3)$ computation for factoring the matrix and $O(n^2)$ memory to store the factors, which makes it challenging to scale to large n . A popular solution is to use a low-rank kernel approximation which speeds up kernel-based solvers while consuming limited memory.

Kernel approximation techniques attempt to find the best rank- k approximation $\mathbf{K} \approx \mathbf{A}\mathbf{A}^T$, with $\mathbf{A} \in \mathbb{R}^{n \times k}$ ($k \ll n$), which has the added benefit that the non-linear kernel matrix can now be written in a separable form just like the linear one.

The L2-SVM formulation for kernel SVM in Equation (4.2) can be written as

$$\min_{\boldsymbol{\alpha}} \frac{1}{2} \boldsymbol{\alpha}^T (\hat{\mathbf{A}}\hat{\mathbf{A}}^T) \boldsymbol{\alpha} + \frac{1}{2} \boldsymbol{\alpha}^T \left(\frac{1}{2C} \mathbf{I}_n \right) \boldsymbol{\alpha} + \mathbf{e}^T \boldsymbol{\alpha}, \quad (4.3)$$

$$\text{subject to} \quad -\mathbf{I}_n \boldsymbol{\alpha} \leq \mathbf{0}_n,$$

where, $\hat{\mathbf{A}} = \text{diag}(y) \times \mathbf{A}$, and $\mathbf{y} = \{y_i \in \{-1, 1\}, i = 1, \dots, n\}$. L2-SVM provides a simpler constraint formulation which specifies that each α_i corresponding to \mathbf{x}_i must be non-negative. The

data points corresponding to positive scalar α_i 's are the *support vectors* on which the separating hyperplane rests.

4.3.2 Challenges

The matrix $\hat{\mathbf{A}} \in \mathbb{R}^{n \times k}$ with $k \ll n$, has a tall and skinny structure. For SVM problems involving large n , $\hat{\mathbf{A}}\hat{\mathbf{A}}^T$ is a dense square matrix (Hessian component of the SVM objective function) requiring $O(n^2)$ memory to store and $O(n^3)$ computation. Furthermore, this square matrix can not be decomposed into independent and separable sub-matrices. Hence, it is difficult to parallelize the training while using this matrix. In addition, $\hat{\mathbf{A}}\hat{\mathbf{A}}^T$ is a rank deficient matrix that can lead to instability when minimizing the Lagrangian of the *dual* SVM problem in Equation (4.6).

4.4 Memory-Efficient Framework

In this section, we propose QRSVM which is based on incorporating Householder QR sketches to SVM problem. QRSVM framework addresses the above challenges by employing the QR decomposition technique to efficiently transform the dense coefficient (hessian) matrix into a sparse and memory-efficient form followed by the dual ascent method to solve the resulting optimization problem iteratively until convergence is achieved.

4.4.1 QRSVM

Here, we present a detailed formulation of the QRSVM framework and also discuss its benefits.

4.4.1.1 Formulation

The QRSVM formulation is based on the use of a rank- k approximation of the kernel matrix K for which we use MEKA, a "Memory Efficient Kernel Approximation" technique [5]. MEKA uses nearly same amount of storage as other approximation techniques like incomplete Cholesky decomposition [94], Greedy basis selection techniques [95] and Nyström [96] methods, while achieving lower approximation error. As an example, on the covtype dataset with half a million samples, MEKA takes around 70 seconds and uses less than 80 MB memory on a single machine to achieve 10% relative approximation error, while standard Nyström approximation is about $6x$

slower and uses more than 400 MB memory to achieve similar approximation [5].

QR decomposition of the matrix $\hat{\mathbf{A}}$ yields $\hat{\mathbf{A}} = \mathbf{QR}$. Here, \mathbf{Q} is an orthogonal matrix of size $n \times n$ and \mathbf{R} is an upper trapezoidal matrix of size $n \times k$. The cost function of the non-linear SVM problem in Equation (4.3) now becomes

$$\frac{1}{2}\boldsymbol{\alpha}^T(\mathbf{QRR}^T\mathbf{Q}^T)\boldsymbol{\alpha} + \frac{1}{2}\boldsymbol{\alpha}^T\left(\frac{1}{2C}\mathbf{I}_n\right)\boldsymbol{\alpha} + \mathbf{e}^T\boldsymbol{\alpha} \quad .$$

Defining $\hat{\boldsymbol{\alpha}} = \mathbf{Q}^T\boldsymbol{\alpha}$, $\hat{\mathbf{e}} = \mathbf{Q}^T\mathbf{e}$ and using $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}_n$, the L2-SVM quadratic programming problem becomes

$$\min_{\hat{\boldsymbol{\alpha}}} \frac{1}{2}\hat{\boldsymbol{\alpha}}^T\left(\mathbf{RR}^T + \frac{1}{2C}\mathbf{I}_n\right)\hat{\boldsymbol{\alpha}} + (\hat{\mathbf{e}})^T\hat{\boldsymbol{\alpha}} \quad (4.4)$$

$$\text{subject to} \quad -\mathbf{Q}\hat{\boldsymbol{\alpha}} \leq \mathbf{0}_n \quad .$$

In Equation (4.4), \mathbf{RR}^T is a symmetric sparse matrix of size n where the non-zeros are restricted to the first $k \times k$ submatrix. Thus, the Hessian of the cost function in Equation (4.4) is a block diagonal matrix comprising of two diagonal blocks: An $k \times k$ symmetric and dense submatrix, $(\mathbf{RR}^T)_k + (1/2C)\mathbf{I}_k$ and a diagonal submatrix, $(1/2C)\mathbf{I}_{n-k}$.

4.4.1.2 Benefits

The key benefits of the QRSVM formulation are as follows

1. **Sparsity:** It transforms the Hessian of the optimization problem from a dense $n \times n$ matrix $(\hat{\mathbf{A}}\hat{\mathbf{A}}^T + (1/2C)\mathbf{I}_n)$ in Equation (4.3) to an overall sparse matrix which consists of a small dense $k \times k$ block in Equation (4.4). Thus, it requires lesser storage of $O(k^2)$ compared to earlier dense representation of $O(n^2)$. We illustrate the above sparsity benefit in Figure 4.1.
2. **Separability:** As observed in Figure 4.1, QRSVM also renders the aforementioned non-separable Hessian matrix into a block separable form. Now, it is possible to further leverages

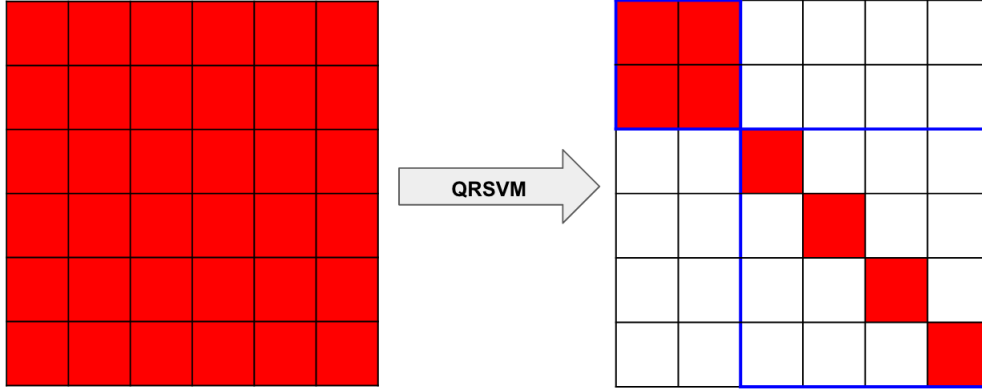


Figure 4.1: QRSVM technique transforms a 6×6 dense and non-separable Hessian (coefficient) matrix into a sparse block diagonal matrix, where, the first 2×2 block is full rank and the second 4×4 block is a diagonal submatrix. Dense regions are colored. The two blocks in the transformed matrix on the Right are outlined in blue. Here, $n = 6$ and $p = 2$. Reprinted with permission from [2].

this separability to independently solve the two sub-problems in parallel using the dual ascent algorithm as shown in the next section.

3. **Stability:** Training an SVM requires solving the quadratic programming problem defined in Equation (4.3) which is numerically stable [97]. QRSVM incorporates a stable MEKA [5] technique as a pre-processing stage followed by Householder [98] method for QR decomposition. On applying QR decomposition, the low-rank Hessian matrix, $\hat{\mathbf{A}}\hat{\mathbf{A}}^T$ becomes block-separable where the two sub-diagonal blocks are now invertible. The first block $(\mathbf{R}\mathbf{R}^T)_k + (1/2C)\mathbf{I}_k$ is full-rank and the second block $(1/2C)\mathbf{I}_{n-k}$ is trivially invertible. The invertibility of the Hessian ensures stable computation of the minimization step, Equation (4.6), in the dual ascent stage. Overall, the QRSVM formulation is numerically stable.

4.4.2 Optimization

Here, we list the update steps of the dual ascent algorithm to optimize (train) the QRSVM optimization problem with guaranteed convergence. We also propose an optimal step size formulation for faster convergence of the algorithm in fewer iterations. Finally, we analyze the time complexity of our proposed QRSVM framework.

4.4.2.1 Dual Ascent Algorithm

Dual ascent is a gradient method which involves iterating through the update steps until convergence [79]. The Lagrangian \mathcal{L} of the Quadratic Programming problem in Equation (4.4) is written as follows

$$\mathcal{L}(\hat{\alpha}, \beta) = \frac{1}{2} \hat{\alpha}^T \left(\mathbf{R}\mathbf{R}^T + \frac{1}{2C} \mathbf{I}_n \right) \hat{\alpha} + (\hat{\mathbf{e}})^T \hat{\alpha} + \beta^T (-\mathbf{Q}\hat{\alpha}), \quad (4.5)$$

where, $\beta \geq \mathbf{0}_n$ is the vector corresponding to Lagrangian dual variable.

Dual ascent update steps for QRSVM are as follows.

Step 1: Minimization of Lagrangian

$$\begin{aligned} \hat{\alpha}^{t+1} &= \arg \min_{\hat{\alpha}} \mathcal{L}(\hat{\alpha}, \beta^t) \\ &= - \left(\mathbf{R}\mathbf{R}^T + \frac{1}{2C} \mathbf{I}_n \right)^{-1} (-\mathbf{Q}^T \beta^t + \hat{\mathbf{e}}) \quad . \end{aligned} \quad (4.6)$$

Step 2: Dual variable update

$$\beta^{t+1} = \beta^t + \eta(-\mathbf{Q}\hat{\alpha}^{t+1}) \quad . \quad (4.7)$$

Here $\eta > 0$ is the scalar step size and the superscript $t = 0, 1, 2, \dots$ is the iteration counter. β^0 is initialized to $\mathbf{0}_n$. To satisfy the non-negativity constraint on each scalar β_i and ensure convergence of dual ascent, it is replaced with $\max(0, \beta_i)$ during every iteration.

4.4.2.2 Optimal Step Size

Here, we provide a formulation for computing the optimal step size, η^* that will ensure the least number of iterations for the dual update. For this, we extend Theorem 2.1 which states a result for the optimal synchronization period in a given parallel Quadratic Programming (QP) problem solved using the proposed *lazy synchronous dual ascent* (LSDA) technique in Chapter 2.

In the parallel QP problem, every worker node in a cluster of processors computes its local minimization step, analogous to Equation (4.6) above. For the dual variable update step, these local w_i 's are aggregated by lazily synchronizing the cluster nodes, once in every optimal syn-

chronization period, P^* . We observe that the above technique can also be interpreted as *tightly* synchronizing among the various nodes with an optimal step size, where, P^* can be considered as the optimal scaling factor.

We provide the following lemma (extending Theorem 2.1 in Chapter 2) for the dual ascent method to ensure a minimum number of iterations for the solution of QRSVM to converge.

Lemma 4.1. (Scaling factor for optimal step size) *To ensure the minimum number of iterations involving the dual variable update step, the scaling factor P^* for optimal step size is obtained by*

$$P^* = \max_{P \in \mathbb{N}} \arg \min \max\{|1 - \lambda_{\min}(\mathbf{M})P|, |1 - \lambda_{\max}(\mathbf{M})P|\} \quad (4.8)$$

where, $\mathbf{M} := \eta \left(\mathbf{R}\mathbf{R}^T + \frac{1}{2C} \mathbf{I}_n \right)^{-1}$, $\eta > 0$ is step size and $\lambda_{\min}(\cdot)$ and $\lambda_{\max}(\cdot)$ denote the smallest and the largest eigenvalues of the square matrix \mathbf{M} , respectively.

On solving equation (4.8), we get the following result.

Corollary 4.1. *For any $\eta > 0$, the optimal step size η^* can be computed using*

$$\eta^* = P^* \eta, \quad P^* \in \mathbb{N} \quad (4.9)$$

where,

$$P^* = \begin{cases} 1 & \text{if } 0 < \bar{\lambda}^{-1} < 2 \\ \lfloor \bar{\lambda}^{-1} \rfloor & \text{if } \bar{\lambda}^{-1} \geq 2 \end{cases}$$

, and $\bar{\lambda} = (\lambda_{\max}(\mathbf{M}) + \lambda_{\min}(\mathbf{M}))/2$

Proof. On plotting the functions $f_1(P) = |1 - \lambda_{\min}(\mathbf{M})P|$ and $f_2(P) = |1 - \lambda_{\max}(\mathbf{M})P|$, we can observe that the intersection of the above two functions occurs at point $L(\bar{\lambda}^{-1}, \lambda_{\max} \bar{\lambda}^{-1} - 1)$ as illustrated in Figure 4.2.

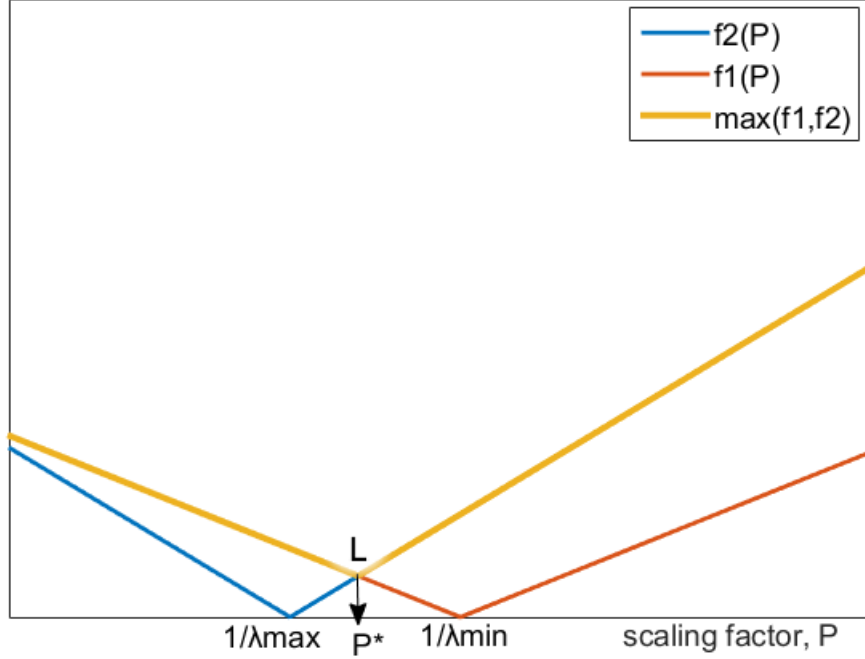


Figure 4.2: Optimal scaling factor, P^* . Reprinted with permission from [2].

$$\max\{f_1(P), f_2(P)\} = \begin{cases} 1 - \lambda_{\min}(\mathbf{M})P & \text{if } 0 < P \leq L_x \\ \lambda_{\max}(\mathbf{M})P - 1 & \text{if } P > L_x \end{cases}$$

where, $L_x = \bar{\lambda}^{-1}$

Minimum value of $\max\{f_1(P), f_2(P)\}$ occurs at point of intersection L . In other words,

$$\arg \min_{P \in \mathbb{N}} \max\{f_1(P), f_2(P)\} = L_x.$$

Since, $P^* \in \mathbb{N}$ as per Lemma 4.1, we must ensure that for $0 < L_x < 2$, optimal scaling factor $P^* = 1$. When $L_x \geq 2$, P^* is assigned the highest integral value lesser than L_x , i.e., $P^* = \lfloor \bar{\lambda}^{-1} \rfloor$. This value which is lesser than L_x ensures that the scaling factor is optimum, P^* , and will result in optimal step size, η^* , leading to stability and convergence of dual ascent method. \square

It is worth noting that given the special structure of \mathbf{M} in our case comprising of the in-

verse of a block separable sparse matrix (positive semi-definite), we get $\lambda_{min}(\mathbf{M}) = 2\eta C/(1 + 2C\lambda_{max}(\mathbf{R}\mathbf{R}^T))$ and $\lambda_{max}(\mathbf{M}) = 2\eta C$. For practical values of C in the proposed formulation, $\lambda_{max}(\mathbf{M}) \gg \lambda_{min}(\mathbf{M})$. Hence, $\bar{\lambda}^{-1} \approx 1/(\eta C)$ can be used as a good approximation for faster convergence of the dual ascent method.

4.4.2.3 Complexity Analysis

In this subsection we present the computational complexity associated with the QRSVM framework. The framework is illustrated in Figure 4.3 comprising of two major stages: QR decomposition and dual ascent.

For implementing the QR decomposition of $\hat{\mathbf{A}}$, Householder transformation [98] is chosen since it has better numerical stability than the Gram-Schmidt process [99] and requires fewer arithmetic operations compared to Givens rotations. As discussed in [98], orthogonal matrix Q is never explicitly computed. Rather, it is stored as a set of k - Householder reflectors. The computational complexity for QR factorization is $O(nk^2)$.

In the dual ascent stage, the computational complexity of a single iteration of QRSVM is the combined cost of the two update steps defined in Equations (4.6) and (4.7). Premultiplying \mathbf{Q} (or \mathbf{Q}^T) to any vector v by using Householder reflectors requires $O(nk)$ operations, where n is the size of vector v and k is the number of Householder reflectors [100]. Thus, the cost of computing $(-\mathbf{Q}^T\beta^t + \hat{\mathbf{e}})$ in Equation (4.6) and $(-\mathbf{Q}\hat{\alpha}^{t+1})$ in Equation (4.7) is $O(nk)$. The multiplication of the block diagonal structure of $(\mathbf{R}\mathbf{R}^T + \frac{1}{2C}\mathbf{I}_n)^{-1}$ with the precomputed $(-\mathbf{Q}^T\beta^t + \hat{\mathbf{e}})$ in Equation (4.6) can be split into following two subproblems.

Subproblem 1: The first k components of $\hat{\alpha}^{t+1}$ are computed by solving a system with the $k \times k$ coefficient matrix $(\mathbf{R}\mathbf{R}_k^T + \frac{1}{2C}\mathbf{I}_k)$. By computing and storing Cholesky factors of this matrix before starting the iterations, the system can be solved in $O(k^2)$ operations. Cholesky factorization of the coefficient matrix is a one time calculation that is carried out in the beginning of the dual ascent algorithm at the cost of $O(k^3)$.

Subproblem 2: Calculation of the remaining $(n - k)$ components of $\hat{\alpha}^{t+1}$ requires $O(n - k) \approx O(n)$ operations as it is reduced to scalar multiplication with $2C$.

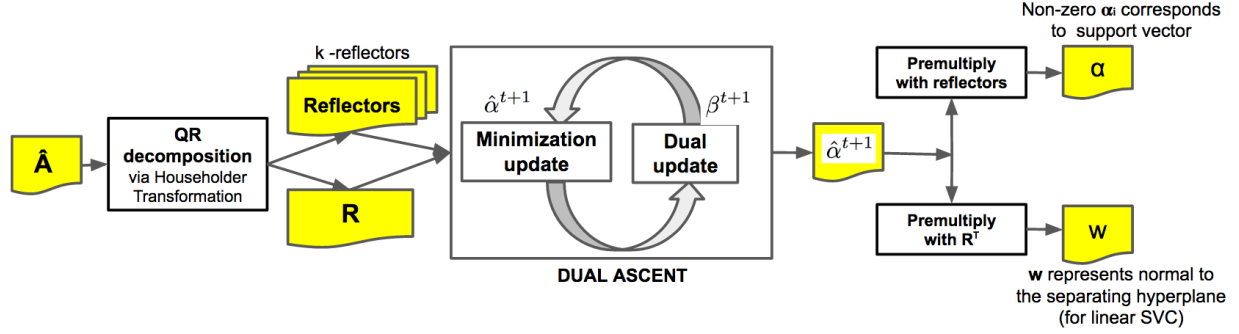


Figure 4.3: QRSVM framework comprises of two main stages, namely, (1) QR decomposition of the approximated input matrix $\hat{\mathbf{A}}$ that yields Householder reflectors and a matrix \mathbf{R} , and (2) Dual ascent method to solve the QRSVM problem for obtaining \mathbf{w} , which is the normal to the hyperplane (for linear SVM), and identifying set of support vectors. Reprinted with permission from [2].

Solving these subproblems in Equation (4.6) incurs $O(k^3 + k^2 + n)$ cost. Since, the pre-computation of $(-\mathbf{Q}^T \beta^t + \hat{\mathbf{e}})$ mentioned earlier requires $O(nk)$ cost, the overall computational cost per iteration incurred in Equation (4.6) is $O(nk + k^3 + k^2 + n) \approx O(nk)$, since $k \ll n$. Also, the computational cost of Equation (4.7) is trivially $O(nk)$ for a single iteration as mentioned earlier. Combining the cost for Equations (4.6) and (4.7), the effective computational complexity for dual ascent is $O(nk)$ per iteration.

The overall QRSVM with both the stages requires $O(nk^2 + nkt_c)$ operations, where t_c is the number of iterations required for convergence of the dual ascent. The trend is empirically illustrated in our experiments which depicts training time is linearly dependent on the number of samples, n .

To identify support vectors for prediction, the values of α are recovered from $\hat{\alpha}$ by simply pre-multiplying $\hat{\alpha}$ with \mathbf{Q} via k -Householder reflectors in $O(nk)$ operations. For kernel SVMs where k -rank kernel approximation techniques such as the one in [101] are used, QRSVM finds the prediction for a test sample in $O(k^2)$ cost using the simplification $\mathbf{R}_k^T \hat{\alpha}_k$.

4.4.3 Distributed QRSVM

With the increase in the quantity of data and challenges associated with it in terms of computation and storage, it has become evident to look for distributed algorithms that can solve parallel SVM problems efficiently. In this section, we present the formulation of parallel non-linear SVMs and a distributed version of our proposed QRSVM framework to solve the classification problems.

4.4.3.1 Distributed QR decomposition

We present a distributed version of the QRSVM framework.

In the QRSVM framework, $\hat{\mathbf{A}}$ is decomposed into factors \mathbf{Q} and \mathbf{R} . To deal with large data sizes, one partitions the data matrix $\hat{\mathbf{A}} = [\hat{\mathbf{A}}_1^T, \dots, \hat{\mathbf{A}}_p^T]^T$ across p computing cores such that each core receives $\frac{n}{p}$ rows of $\hat{\mathbf{A}}$ (assume n is a multiple of p). The number of cores p should be less than $\frac{n}{k}$ to ensure each block has more rows than columns. It is preferable to have $p \ll \frac{n}{k}$ to maintain a tall and skinny structure on each core. We now discuss how \mathbf{Q} and \mathbf{R} can be generated from the partitioned matrices $\hat{\mathbf{A}}_i, i = 1, \dots, p$ and stored in a distributed manner similar to parallel Tall-Skinny QR (TSQR) algorithm [67].

We now discuss how to distribute the decomposition of $\hat{\mathbf{A}} = \mathbf{QR}$ across cluster of computing cores such that the resulting matrices \mathbf{Q} and \mathbf{R} ; representatives of the overall data, can be generated from the partitioned matrices $\hat{\mathbf{A}}_i$.

Theorem 4.1. *For QR decomposition of a matrix $\hat{\mathbf{A}} = \mathbf{QR}$, where \mathbf{Q} is an orthogonal matrix and \mathbf{R} is an upper triangular matrix, we can generate \mathbf{Q} and \mathbf{R} from p horizontal partitions of $\hat{\mathbf{A}} = \{\hat{\mathbf{A}}_i\}, i = 1 \dots p$ as follows*

$$\mathbf{Q} = \text{diag}(\mathbf{Q}_1, \mathbf{Q}_2, \dots, \mathbf{Q}_p) \times \mathbf{Q}_g$$

$$\mathbf{R} = \mathbf{R}_g$$

where, $\hat{\mathbf{A}}_i = \mathbf{Q}_i \mathbf{R}_i$ and $vstack(\mathbf{R}_1, \dots, \mathbf{R}_p) = \mathbf{Q}_g \mathbf{R}_g$. Here, \mathbf{Q}_i and \mathbf{Q}_g are orthogonal matrices and \mathbf{R}_i and \mathbf{R}_g are upper triangular matrices.

Proof. For $\forall i \in 1 \dots p$ computing cores, $\hat{\mathbf{A}}_i = \mathbf{Q}_i \mathbf{R}_i$. We can write the overall $\hat{\mathbf{A}}$ as

$$\hat{\mathbf{A}} = \begin{bmatrix} \hat{\mathbf{A}}_1 \\ \vdots \\ \hat{\mathbf{A}}_p \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_1 & & \\ & \ddots & \\ & & \mathbf{Q}_p \end{bmatrix} \begin{bmatrix} \mathbf{R}_1 \\ \vdots \\ \mathbf{R}_p \end{bmatrix}$$

Since each \mathbf{Q}_i is an orthogonal matrix, $\text{diag}(\mathbf{Q}_1 \dots \mathbf{Q}_p)$ is an orthogonal matrix. Since each $\mathbf{R}_i \in \mathbb{R}^{\frac{n}{p} \times k}$ is an upper triangular matrix, $\mathbf{R}_{gather} = \begin{bmatrix} \mathbf{R}_1 \\ \vdots \\ \mathbf{R}_p \end{bmatrix}$ is *not* an upper triangular matrix.

In the master core, we can further decompose

$$\mathbf{R}_{gather} = \mathbf{Q}_g \mathbf{R}_g$$

such that $\mathbf{Q}_g \in \mathbb{R}^{n \times n}$ is an orthogonal matrix and $\mathbf{R}_g \in \mathbb{R}^{n \times k}$ is an upper triangular matrix.

Now,

$$\hat{\mathbf{A}} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p) \times \mathbf{Q}_g \mathbf{R}_g$$

We know that $\hat{\mathbf{A}} \in \mathbb{R}^{n \times k}$ is k -rank. In other words, $\hat{\mathbf{A}}$ has linearly independent columns which imply its QR decomposition is unique. Hence, $\mathbf{R} = \mathbf{R}_g$ and $\mathbf{Q} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p) \times \mathbf{Q}_g$ \square

The implementation of the distributed QR decomposition technique is further illustrated in Figure 4.4.

Corollary 4.2. For any vector $\mathbf{v} \in \mathbb{R}^n$, $\hat{\mathbf{v}} = \mathbf{Q}^T \mathbf{v}$; stored as partitions in the cluster cores and orthogonal matrices \mathbf{Q} and \mathbf{Q}_i , computing $\mathbf{Q} \hat{\mathbf{v}}$ and $\mathbf{Q}^T \mathbf{v}$ in the distributed QRSVM framework can be formulated as

$$\mathbf{Q} \hat{\mathbf{v}} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p) \times \{\mathbf{Q}_g \hat{\mathbf{v}}\}$$

and

$$\mathbf{Q}^T \mathbf{v} = \mathbf{Q}_g^T \times \{\text{diag}(\mathbf{Q}_1^T, \dots, \mathbf{Q}_p^T) \mathbf{v}\}$$

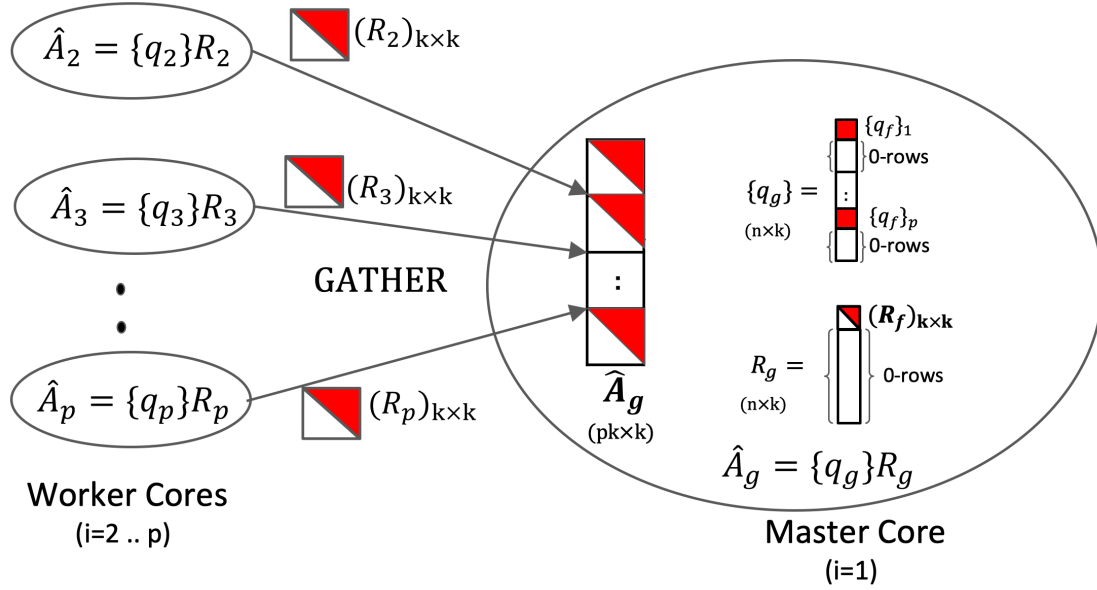


Figure 4.4: Distributed QR decomposition: The orthogonal matrices, Q , are stored as sets of their Householder reflectors, denoted as $\{q\}$. Rather than sending the entire $(n/p) \times k$ sized matrix R_i , we only communicate its informational content, denoted as $(R_i)_{k \times k}$ i.e. the upper triangular $k \times k$ block. Thereby, we reduce the size of the matrices being communicated across the distributed network by avoiding redundant zero-blocks. At the master core, instead of creating the complete matrix with redundant zero-blocks, we *gather* the $(R_i)_{k \times k}$ from all the worker nodes to generate a stacked-up representation called R_{gather} . On QR decomposition, it factorizes into $\{q_f\}$ and R_f . We retrieve the original informational content in R_g as $(R_f)_{k \times k}$ and on zero-padding the $\{q_f\}_i$ blocks, we generate the reflectors $\{q_g\}$ for Q_g . Reprinted with permission from [2].

Each of the above computation inherently requires communication (gather and scatter) across the distributed network.

For $Q\hat{v}$ computation, we first *gather* all local \hat{v}_i from the worker cores and calculate $\{Q_g\hat{v}\}$ at the master core. Then, we *scatter* its partition $\{Q_g\hat{v}\}_i$ across all the computing cores in a cluster. Each of which has its local Q_i from QR decomposition of its partitioned data \hat{A}_i . The *scattered* $\{Q_g\hat{v}\}_i$ is then pre-multiplied with local Q_i to compute v_i .

For $Q^T v$ computation, we first calculate local $\{Q_i^T v_i\}$ in each worker node i . Then, we *gather* each of the local values at the master node and pre-multiply it with Q_g^T to generate \hat{v} . Finally, we *scatter* it to all the cluster nodes as \hat{v}_i . Again, it is worth noting that these Q_i 's and Q_g are never explicitly calculated, rather, stored as respective sets of k - Householder reflectors in the

computing nodes.

4.4.3.2 Parallel Dual Ascent

With the distributed QR decomposition technique, it is possible to update the *dual ascent* steps, Equations (4.6) and (4.7), in parallel across the parallel cores (or workers or cluster nodes).

From Equation (4.6), let us define the invertible coefficient matrix (substituting, $\mathbf{R} = \mathbf{R}_g$)

$$\mathbf{F} = -\left(\mathbf{R}_g\mathbf{R}_g^T + \frac{1}{2C} \times \mathbf{I}_n\right)$$

which has a sparse and separable structure, well illustrated in Figure 4.1. As a result of its separability, we can block-partition $\mathbf{F} = \mathbf{F}_1 \oplus \mathbf{F}_2 \oplus \dots \oplus \mathbf{F}_p$ such that each diagonal block $\mathbf{F}_i \in \mathbb{R}^{\frac{n}{p} \times \frac{n}{p}}$ is allocated to each computing core i . Here, \oplus represents an operator that diagonally combines the sub-blocks to generate the entire block-diagonal matrix.

Since \mathbf{R}_g is stored at the master node and rank $k \ll \frac{n}{p}$ as to maintain the tall and skinny structure of $\hat{\mathbf{A}}_i$, the dense and symmetric $k \times k$ block i.e. $\left((\mathbf{R}_g\mathbf{R}_g^T)_k + (1/2C)\mathbf{I}_k\right)$ in coefficient matrix \mathbf{F} becomes a part of the partition \mathbf{F}_1 at the master core. It is also worth appreciating that with the distributed QRSVM formulation, there is no need to *actually* partition and store the other diagonal blocks \mathbf{F}_i in the worker cores $i = 2, \dots, p$, as these are simply constant diagonal matrices $(-0.5/C)\mathbf{I}_{n/p}$.

On parallelizing **Step 1** of the dual ascent for iteration $(t + 1)$,
at compute core, i

$$\hat{\boldsymbol{\alpha}}_i^{t+1} = \mathbf{F}_i^{-1}(-\hat{\boldsymbol{\beta}}_i^t + \hat{\mathbf{e}}_i) \quad (4.10)$$

where,

$$\mathbf{F}_i^{-1} = \begin{cases} \mathbf{F}_1^{-1} & \text{if } i = 1 \\ \text{diag}(-2C) & \text{if } i = 2, \dots, p \end{cases}$$

and, $\hat{\boldsymbol{\beta}}^t = \mathbf{Q}^T \boldsymbol{\beta}^t$

Similarly, on parallelizing **Step 2** of the dual ascent for iteration $(t + 1)$, at compute node, i

$$\hat{\beta}_i^{t+1} = \hat{\beta}_i^t + \eta^*(-\hat{\alpha}_i^{t+1}) \quad (4.11)$$

Here, we are using the optimal step size η^* defined in Equation (4.9). We also note that by changing the dual variable from β to $\hat{\beta}$ in the above update steps, we ensure that $\hat{\alpha}_i^{t+1}$ and $\hat{\beta}_i^{t+1}$ calculation during each iteration $(t + 1)$ occurs locally without requiring any communication or synchronization across the computing cluster cores. However, after every iteration, the original dual variable β has to be checked for non-negativity as discussed earlier in Section 4.4.2.1. This requires communication (*gather* and *scatter*) across the computing cores as we transform from $\hat{\beta}$ to β using $\mathbf{Q}\hat{\beta}$ and then transform back to $\hat{\beta}$ (to be used in Step 1 in the next iteration) using $\mathbf{Q}^T\beta$ after checking (and zeroing) the negative β values.

4.5 Experiment and Results

We evaluate the proposed sequential QRSVM framework in comparison to other state-of-the-art sequential solver LIBLINEAR for HIGGS dataset for relative convergence. We also validate the linear scalability of QRSVM with a number of training samples. For the distributed version of the QRSVM framework, we train non-linear classifiers for benchmarks a9a and covtype and report their computation and communication timings. Finally, we compare the proposed distributed QRSVM with PSVM and P-packSVM.

4.5.1 Hardware Description

The experiments were performed in *Ada* super-computing cluster of Texas A&M HPRC with InfiniBand interconnect. Each of the 792 general compute nodes in this super-computing cluster is a 10-core Intel Xeon E5-2670 v2 (Ivy Bridge) processor with 64 GB memory. We use Message-Passing Interface (MPI) [102] as inter-node communication platform.

4.5.2 Experimental Setup

The distributed QRSVM framework is implemented in C/C++. The linear algebra computations in the framework were handled using Armadillo library [34] integrated with LAPACK/BLAS. We use binary classification datasets freely available on LIBSVM datasets repository ²

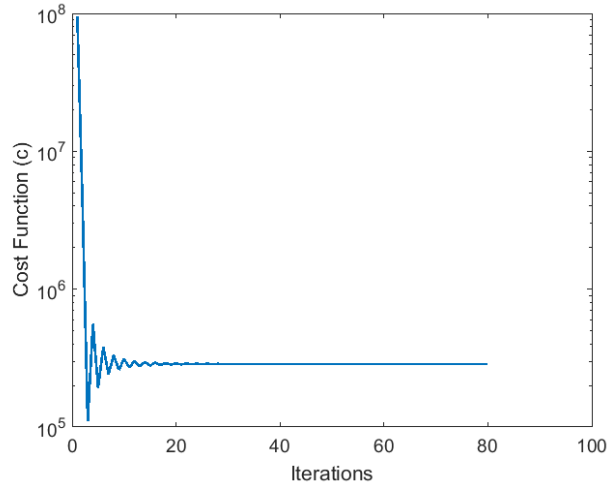
- a9a: The machine learning task here is to predict whether income exceeds \$50K/yr based on census data. Data sparsity is 11.3%. $n_{train} = 32560$, $d = 123$
- covtype.binary: One needs to predict the forest cover type (class 2 vs other 5 classes) using cartographic variables only. We use the [0,1] scaled version of the dataset having sparsity of 22%. $n_{train} = 464810$, $d = 54$

4.5.3 Results and Discussions

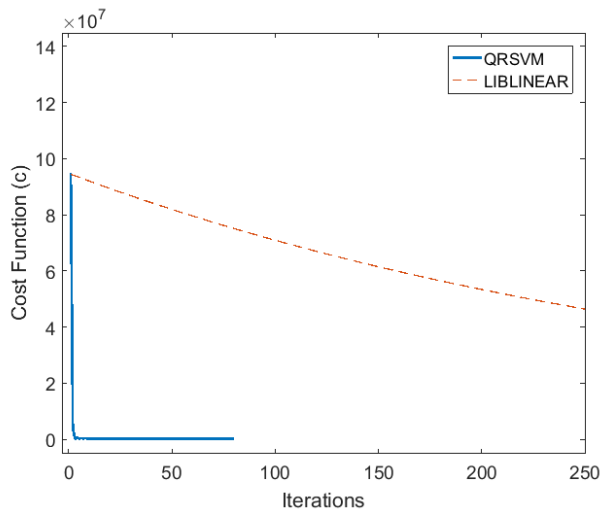
4.5.3.1 Convergence

We show the convergence of the basic QRSVM framework on single-machine and compare it with state-of-the-art linear solver library, LIBLINEAR (dual coordinate descent). We choose HIGGS² dataset ($n = 10.5M$ and $d = 28$) in linear SVM formulation here to simply demonstrate the superior convergence of our approach compared to the linear SVM solver. As shown in Figure 4.5a, it was observed that for most of the problems, QRSVM reaches a reasonable value of the objective function within 20 iterations of dual ascent method. Figure 4.5b compares the convergence rates of QRSVM with LIBLINEAR. It must be pointed out that the LIBLINEAR algorithm implementation in [85] is set to terminate at a preset maximum iteration count (default 1000). In our experimentation, we chose this preset value to 1500. The result at the end of this maximum iteration was used to compare with QRSVM. QRSVM was able to converge to the optimal cost of the objective function for HIGGS dataset within 80 iterations while LIBLINEAR couldn't converge even after 1500 iterations. For non-linear SVM, Figure 4.6 shows the convergence of distributed QRSVM as function of training error (log) over number of iterations for datasets a9a and covtype.

²<https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html>



(a) QRSVM



(b) QRSVM vs LIBLINEAR (DCD)

Figure 4.5: Convergence of QRSVM for HIGGS dataset: (a) Using QRSVM we converge to a reasonable value of the optimal cost within 20 iterations. (b) QRSVM converges relatively faster compared to LIBLINEAR. Here, we illustrate for 250 iterations. LIBLINEAR was not able to converge to the optimal cost value in 1500 iterations while QRSVM converged to the optimum in 80 iterations. Reprinted with permission from [2].

4.5.3.2 Kernel Approximation Quality

The kernel approximation results are shown in Figure 4.7. We randomly sampled 20,000 rows of kernel matrix to evaluate the relative approximation error for covtype. We use relative kernel approximation error to measure the quality [5]. Figure 4.7 shows the kernel approximation

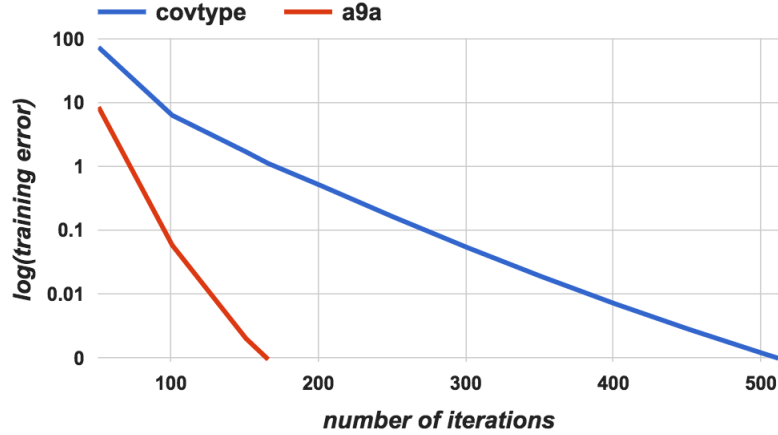


Figure 4.6: Training error trend during model learning phase for the datasets. a9a training takes 166 iterations to converge and covtype takes 512 iterations. Stopping error threshold: 10^{-3} and $p = 16$. Reprinted with permission from [2].

performance of different methods by varying k [5]. The rank k varies from 100 to 600 for covtype. Figure 4.7 shows that MEKA scheme always achieves lower error with less time and memory. This is because MEKA aims to approximate the kernel matrix by a rank- ck approximation (where, c is number of clusters) using similar amount of time and memory while all other methods are only able to form a rank- k approximation.

4.5.3.3 Scalability

We discuss scalability with both sample size (number of instances) n and rank k .

Scalability with n : The effect on training time of QRSVM was analyzed by increasing number of instances, n in a synthetic dataset. Its kernel approximation, $k = 18$ was kept fixed. It is observed that QRSVM training time increases linearly with the number of data points as shown in Figure 4.8a. As a result, our proposed framework is capable of handling growing data sizes efficiently making it suitable for big data applications.

Scalability with rank k : With increasing the rank of the approximated kernel, the training time of QRSVM is observed to grow quadratically as shown in Figure 4.8b. This trend validates our discussion in Section 4.4.2.3. However, as we are addressing the low-rank approximation of the input problem space through the proposed QRSVM framework, the benefits from the fast overall

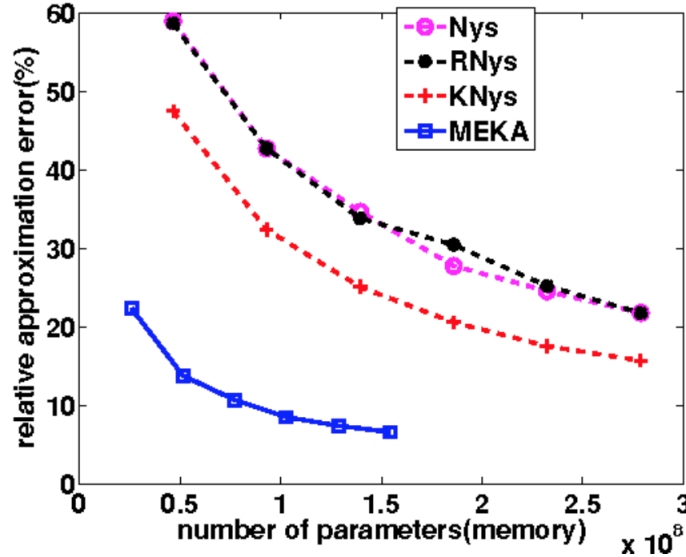


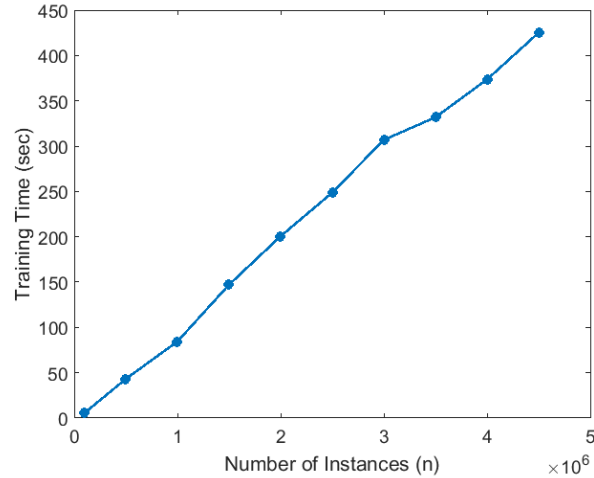
Figure 4.7: Low-rank Gaussian kernel approximation results using MEKA and other methods. Reprinted with permission from [5].

convergence rate of the algorithm outweighs the quadratic cost.

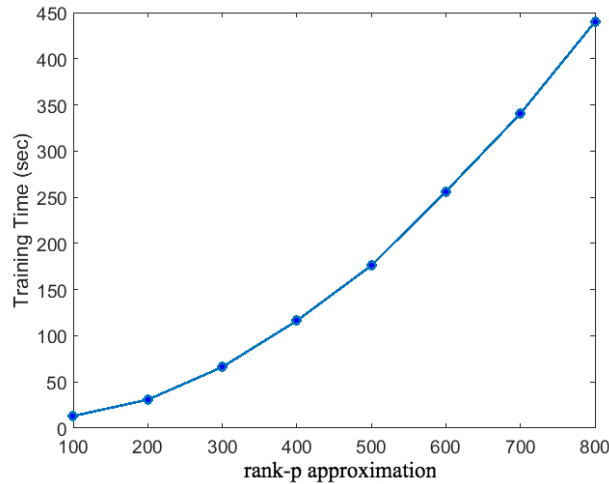
4.5.3.4 Distributed Training Time

Computation time, T^{comp} : The proposed distributed QRSVM framework for parallel SVM training has the following major computational requirements: Low-rank representation of the kernel matrix using MEKA [5] which has time complexity proportional to the memory for storing the approximated kernel matrix. We denote this time as T_{meke}^{comp} . The distributed QR decomposition stage shown in Figure 4.4, partitioned data \hat{A}_i is locally decomposed into Q_i and R_i . We denote its worst case computation time as $T_{localQR}^{comp}$ for any given cluster core. At the master core in the above stage, the *gathered* data \hat{R}_{gather} is further $Q_f R_f$ decomposed whose computation time we denote as $T_{masterQR}^{comp}$. Finally, in the parallel dual-ascent (pda) stage, we solve the update steps in Equations (4.10) and (4.11) locally in the parallel cores. The computation time for the update steps is denoted as, T_{pda}^{comp} . All the above computation timings for the chosen benchmarks are listed in Table 4.1.

Communication time, T^{comm} : The distributed QRSVM framework solves the kernel SVM



(a) Scales linearly with number of instances



(b) Scales quadratically with rank- k approximation.

Figure 4.8: Scalability of QRSVM (a) with sample size n : A synthetic dataset having a fixed approximated kernel rank- $k = 18$ and increasing n was used to test scalability with number of instances. (b) with rank k : A synthetic dataset with fixed number of instances, $n = 100,000$ and increasing rank was used to test scalability with rank- k (dimensionality). Reprinted with permission from [2].

problem in the process of which it needs to communicate data across the distributed network. It has two necessary communication requirements. The first need for communication across the distributed cluster occurs when local $(\mathbf{R}_i)_{k \times k}$ are *gathered* at the master node during the distributed QR decomposition stage. This is also depicted in Figure 4.4. Let us denote this communication time as $T_{gatherR}^{comm}$. The second requirement to communicate happens in the parallel dual-ascent (pda)

Time details	a9a (in ms)	covtype (in s)
T_{meka}^{comp}	460	2.1
$T_{localQR}^{comp}$	24	1.89
$T_{masterQR}^{comp}$	4	0.02
$T_{gatherR}^{comm}$	0.5	0.04
T_{pda}^{comp}	1628.1	120.18
T_{pda}^{comm}	17.1	0.36
T_{train}	1674.2	122.50

Table 4.1: Distributed-QRSVM: Timing details. Reprinted with permission from [2].

Parameters	a9a	covtype
rank, k	40	64
C	2^{-1}	2^{-1}
γ	2^{-3}	2^3
approx. K_{error}	0.51	0.58
#cores, p	16	16
stopping threshold	10^{-3}	10^{-3}
optimal step size, η^*	1.9	1.9
#iterations, t	166	512

Table 4.2: Distributed-QRSVM: Parameter values. Reprinted with permission from [2].

stage. As discussed in Section 4.4.3.2, bulk of the communication happens after every iteration in the form of *gather* and *scatter*. This is necessary to ensure convergence of the algorithm. Let us denote this communication time as T_{pda}^{comm} . The overall SVM training time, $T_{train} = T^{comp} + T^{comm}$ for both the datasets is also shown in Table 4.1.

4.5.3.5 Optimal Step Size

For different values of C and step size η , we derive the optimal step size using Equation 4.9 for training the models for the benchmark datasets, a9a and covtype. Figure 4.9 plots the training time for the various step sizes and we chose the $\eta^* = 1.9$ as optimal that provides the fastest time. Table 4.2 lists the above parameters, C and η^* .

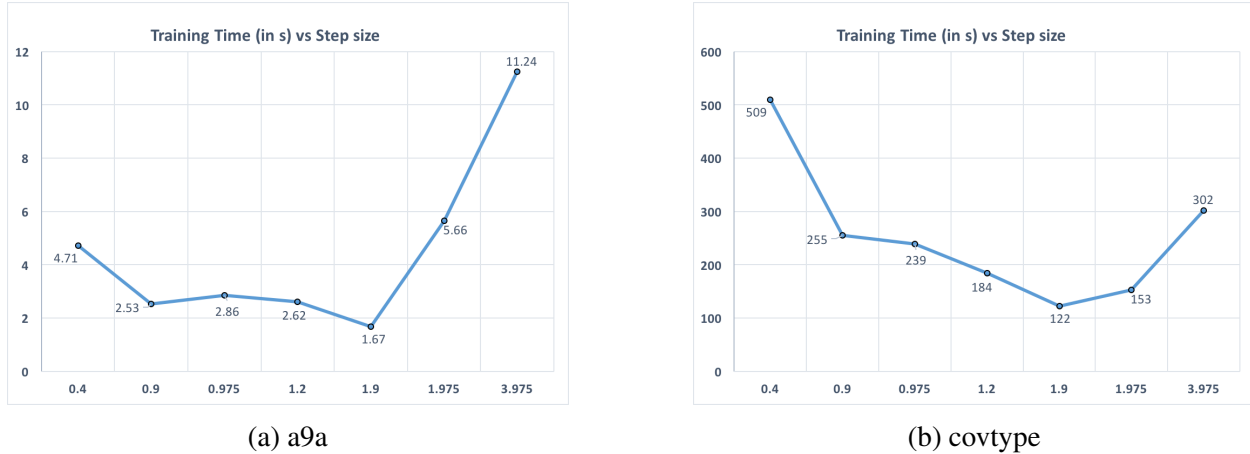


Figure 4.9: Optimal step size: For both datasets, optimal step size is observed to be 1.9. Reprinted with permission from [2].

4.5.3.6 Comparison

We compare the distributed-QRSVM framework with state of the art kernel SVM solver, PSVM [12]. PSVM uses incomplete Cholesky decomposition to approximate the kernel matrix whereas we use the memory-efficient MEKA [5]. Both PSVM and the proposed framework optimize the *dual* objective/cost function making our comparison fair. As for PSVM parameters, we used the default setting. The *dual* residual threshold for convergence was set to 0.001 which we also keep as the stopping threshold for QRSVM. The upper limit of 10,000 iterations was maintained for PSVM and the suggested $m' = m^{0.5}$ was used. As mentioned in [12], the value of m approximation was chosen to ensure a balance between accuracy and efficiency. We observe that the distributed version of the proposed QRSVM framework trains the SVM model for the covtype.binary dataset in around 2 minutes using $p = 16$ processors, whereas, PSVM under the same setup converges in around 20 minutes. Another distributed kernel solver, P-packSVM [88] solves the same dataset 1.24x faster (for P-pack, $r = 100$ model) than PSVM using 16 processors as reported in Table II in [88]. Due to unavailability of open source code for P-packSVM, we were unable to accurately report its training time for the above dataset on our experimental infrastructure. However, we can estimate that P-packSVM would have clocked around 16 minutes (1.24x

better than PSVM) for 16 processors. In contrast, our framework converged in 2 minutes.

4.6 Summary

In this chapter, we proposed a QR decomposition framework for solving kernel support vector classification problems. The framework uses Householder sketch to convert a dense and low rank matrix to a highly sparse and separable matrix, with invertible block-partitions. In addition, we provide optimal step size for solving dual ascent method for faster convergence. We empirically demonstrate that the proposed QRSVM framework scales linearly with dataset size making it suitable to handle large data problems. We further present a distributed QRSVM framework to accelerate the training of kernel SVM model by parallelizing the dual-ascent algorithm. In next chapter, we will seek to further optimize the parallel dual ascent stage involving model-update iteration and communication to further accelerate the SVM training and to make it amenable for scaling on large number of parallel workers.

5. COMMUNICATION-EFFICIENT FRAMEWORK FOR SCALABLE MACHINE LEARNING ¹

To support distributed training on large number of parallel workers in a decentralized machine learning framework, it is paramount to design scalable algorithms which do not suffer from large communication overheads during iterative model training. In this chapter, we present a communication-efficient framework for scalable machine learning.

5.1 Introduction

5.1.1 Motivation

To exploit the potential of faster convergence of dual solvers we proposed a distributed QRSVM framework in Chapter 4 that projects the problem onto a reduced subspace via QR factorization to minimize computation and storage needs. At present, however, it has significant communication overheads which limits its ability to achieve higher speedups in the training phase. In this chapter, we propose a fast and communication-efficient algorithm for distributed QRSVM training that is scalable to large data sets and number of parallel workers.

5.1.2 Contributions

We list the main contributions of this chapter as follows.

1. We leverage memory-efficient QRSVM framework introduced in Chapter 4 to design a communication-efficient implementation to train kernel SVM faster. The improved design significantly reduces the communication overhead by making it insignificant compared to the computation time. As a result, the resulting framework is feasible for efficient parallelism across multiple computing cores and trains SVM faster than the prior framework in Chapter 4. For instance, with the proposed approach and a computed optimal step size of

¹A part of this chapter is reprinted with permission from “Fast and Communication-Efficient Algorithm for Distributed Support Vector Machine Training” by Jyotikrishna Dass, Vivek Sarin, and Rabi N. Mahapatra, 2019 IEEE Transactions on Parallel and Distributed Systems (TPDS), Copyright © 2019, IEEE.

0.9, we are able to train the SVM for covtype dataset on 16 cores in 18 seconds compared to 261 seconds in a prior implementation, which represents an improvement of 14x.

2. We evaluate the performance of our algorithm on three real world benchmarks, covtype (geography), webspam (electronic) and SUSY (physics). The algorithm attains speed improvement of 45x, 29x and 136x, respectively, on these benchmarks on a 64 core multiprocessor with respect to sequential implementation.
3. We demonstrate through experiments that the proposed training algorithm outperforms state-of-the-art algorithms such as PSVM and P-packSVM. For instance, the distributed QRSVM algorithm was 71x faster than PSVM and 57x faster than P-packSVM on the covtype benchmark using 16 cores. In addition, our algorithm scales linearly with the sample size and hence can handle extremely large datasets such as SUSY (5M samples) with ease.

5.2 Related Work

It has become necessary to design parallel and distributed algorithms to train kernel SVMs for large-scale problems. A number of attempts have been made to parallelize kernel SVM training. Cascade-SVM [103] is one of the earliest works that presents a parallel SVM training algorithm in which the global SVM problem is divided into local sub-problems. The cascade framework is designed in a hierarchical reduction-tree-like structure in which each layer uses independent SVM solvers. The drawback of the framework is that for large datasets it either leads to more resource requirements (SVM solvers) or longer training time. In addition, one has to always ensure that the SVM solver at the last stage is capable of handling the output of the previous layers (called support vectors) which have trickled down through the cascade. Communication Avoiding SVM (CA-SVM) [104] is based on k -means clustering technique to partition datasets. The partitioned data are stored locally on the cluster nodes and solved independently. Since it uses the local solution from one of the nodes to predict a test sample, the methodology does not compute the global SVM solution. PSVM [12] and P-packSVM [88] are among the most popular parallel algorithms to solve global kernel SVM. PSVM exhibits limited scalability due to its quadratic dependence

on the training sample size. Moreover, PSVM works with kernel matrix approximation using Incomplete Cholesky Factorization (ICF), which is difficult to parallelize, making it unfit for large datasets. In contrast, P-packSVM computes the primal form of SVM using a stochastic gradient descent method wherein the gradient is approximated at a single sample [88]. Since the primal SVM problem can be prohibitively large while its Wolfe dual problem is considerably smaller, the convergence rate of the primal solver is slower than the dual solver [92]. Hence, it is important to find alternative distributed approaches that are highly scalable and exhibit faster convergence for the dual form of kernel SVMs.

5.3 Communication-Efficient Framework

The distributed implementation of QRSVM presented in Chapter 4 performs well for small and medium-sized problems than the competing algorithms, but like others it is infeasible for large datasets. The major limitations of prior framework affecting its scalability to large datasets and parallel workers are as follows

- Large memory needed to construct and store global factors at the master core, namely, Householder reflectors for \mathbf{Q}_g and the upper trapezoidal matrix \mathbf{R}_g .
- Significant communication overhead incurred during $\mathbf{Q}\hat{\beta}$ and $\mathbf{Q}^T\beta$ operations in each iteration of the parallel dual ascent.

In this section, we describe the efficient implementation for distributed QRSVM along with the improvements done over the one in Chapter 4 to address the above limitations. These improvements are significant in both memory savings and communication efficiency thereby ensuring faster speedup and highly scalable framework for training SVM on large datasets.

5.3.1 Implementing Distributed QR Decomposition

We propose a novel implementation of the distributed QR decomposition technique (Stage 1) wherein memory-efficient representations of the global factors at the master core are generated. Figure 4.4 illustrates the above implementation and Algorithm 5.1 presents the pseudo-code.

Firstly, local QR decompositions are done in parallel across all the cores (Step 4 in Algorithm 5.1) using Householder transformation to generate two local factors. The first local factor is an orthogonal matrix \mathbf{Q}_i in each of the computing cores. It is stored as a set of k -Householder reflectors [98], which we denote as $\{\mathbf{q}_i\}$ of size $(n/p) \times k$. Here, each column is a reflector in the set. The second local factor of the decomposition is the upper trapezoidal matrix \mathbf{R}_i of size $(n/p) \times k$. It comprises of $k \times k$ upper triangular block followed by zero-blocks. We denote the upper triangular block as $(\mathbf{R}_i)_{k \times k}$. Next, as per the formulation described in Section 4.4.3.1, the local factor, \mathbf{R}_i , from all the cores needs to be *gathered* at the master core. However, to ensure communication-efficient implementation, each of the worker cores communicates only its local upper triangular block $(\mathbf{R}_i)_{k \times k}$ to the master core (see Step 5 in Algorithm 5.1) rather than sending the complete matrix \mathbf{R}_i comprising of redundant zero-blocks. Through this strategy, we reduce the communication volume across the distributed network from $O(\frac{n}{p})$ to $O(k)$ per core. Then, at the master core, all these local upper triangular blocks $(\mathbf{R}_i)_{k \times k}$ which have been gathered from all the computing cores p are stacked to generate $\hat{\mathbf{A}}_g$ of size $pk \times k$ (see Step 7 in Algorithm 5.1). This is significantly smaller memory footprint than storing the original \mathbf{A}_g of size $n \times k$ which would have required gathering the complete \mathbf{R}_i with redundant zero-blocks. Finally in the master core, QR decomposition of $\hat{\mathbf{A}}_g$ using Householder transformation generates memory-efficient global factors; a set of Householder reflectors $\{\mathbf{q}_f\}$ and an upper trapezoidal matrix \mathbf{R}_f , as illustrated in Figure 4.4 and in Step 8 of Algorithm 5.1.

Memory improvement over Chapter 4: In the prior implementation of the distributed QRSVM as described in Chapter 4, large memory was required to construct and store these global factors (set of Householder reflectors and upper trapezoidal matrix) at the master core. Firstly, the set of k -Householder reflectors for the orthogonal matrix \mathbf{Q}_g , denoted as $\{\mathbf{q}_g\}$ of size $n \times k$, was constructed by appending $(\frac{n}{p} - k)$ rows of zeros to each $\{\mathbf{q}_f\}_i$ block. In addition, the second global factor, namely, the upper trapezoidal matrix, \mathbf{R}_g , was generated by appending $(n - k)$ rows of zeros to $(\mathbf{R}_f)_{k \times k}$. Figure 4.4 illustrates these constructions at the master core. For large sample size n , the memory consumption for $\{\mathbf{q}_g\}$ and \mathbf{R}_g increases linearly with n , posing a serious challenge

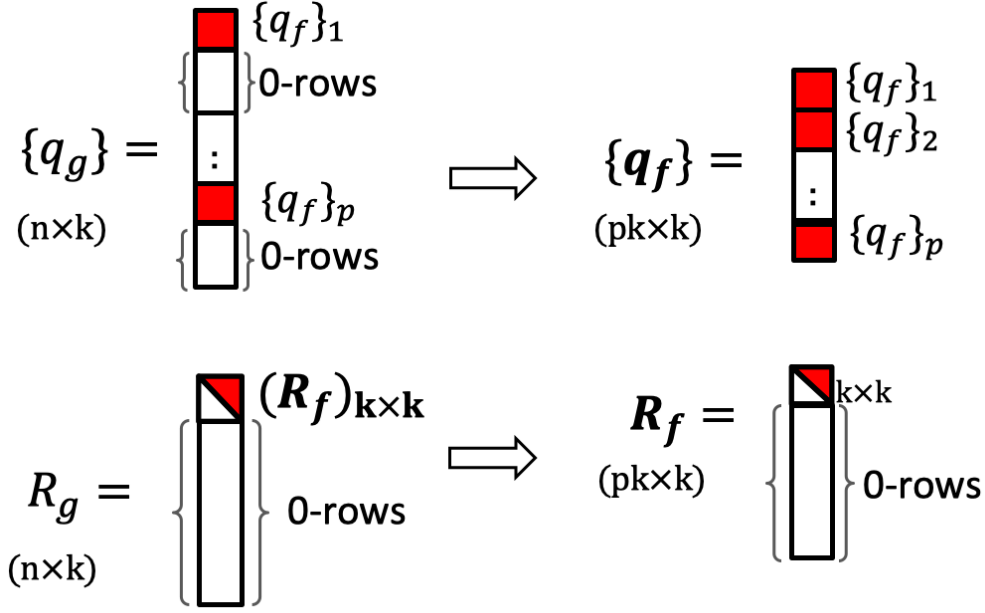


Figure 5.1: At master core: Memory improvement and representation of QR factors.

to its performance. Such large memory requirement at the master core limits the capabilities of the framework in Chapter 4 to small and medium-sized datasets only. However, in the proposed implementation we avoid constructing these global factors, $\{q_g\}$ and R_g . Rather, we retain their memory-efficient representations, denoted as $\{q_f\}$ and $(R_f)_{k \times k}$, respectively. Figure 5.1 depicts the construction of these memory-efficient representations at the master core while Algorithm 5.1 lists those as Steps 8-9. By retaining the global reflector set as $\{q_f\}$ of size $pk \times k$ rather than constructing $\{q_g\}$ of size $n \times k$ at the master core, we now achieve significant memory savings worth $(\frac{n}{pk})$ times compared to Chapter 4 framework, where, $p \ll \frac{n}{k}$. Henceforth, the distributed QRSVM framework is designed only using these memory-efficient representations, $\{q_f\}$ and $(R_f)_{k \times k}$. We illustrate the improved QRSVM framework in Figure 5.2

5.3.2 Implementing Parallel Dual Ascent

As mentioned in Section 4.4.3.2, the update Equations (4.10) and (4.11) are trivially parallelized across the cores (see Step 4 and Step 5 in pseudo-code in Algorithm 5.2). However, transformations from $\hat{\beta}$ to β and back needs to be done in every iteration to ensure the non-negativity

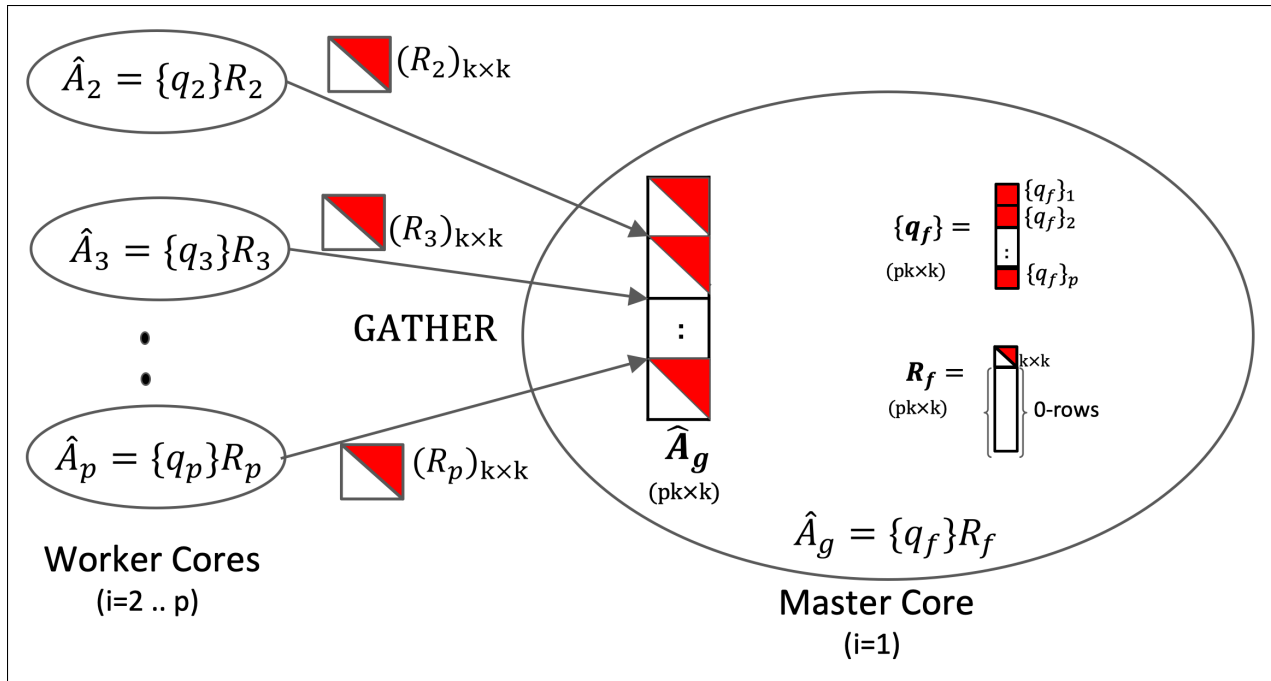


Figure 5.2: A two-level implementation of distributed QR decomposition of \hat{A} . The orthogonal matrices are stored as sets of their Householder reflectors, denoted as $\{q\}$. $(R_i)_{k \times k}$ are gathered from all the cores and \hat{A}_g is assembled at the master core. $\hat{A}_g = \{q_f\}R_f$ is computed which can be converted to original global factors, $\{q_g\}$ and R_g by appending appropriate rows of zeros as depicted in Chapter 4. However, the proposed implementation here uses memory-efficient representations of the global factors for \hat{A}_g , i.e., $\{q_f\}$ and $(R_f)_{k \times k}$. Reprinted with permission from [6].

Algorithm 5.1 Distributed QR decomposition

Data: Matrix \hat{A} , number of parallel cores p

- 1 $k \leftarrow \text{rank}$
 - 2 **for** each core i, \dots, p **do**
 - 3 $\hat{A}_i \leftarrow$ local partitioned data
 - 4 Parallel Compute $\{q_i\}, R_i \leftarrow \hat{A}_i$
 - 5 GATHER $(R_i)_{k \times k}$ at Master core
 - 6 **end**
 - 7 $\hat{A}_g \leftarrow$ gathered or stacked $(R_i)_{k \times k}$
 - 8 Compute $\{q_f\}, R_f \leftarrow \hat{A}_g$ at Master core
 - 9 Use $(R_f)_{k \times k}$
-

of β for guaranteed convergence of the algorithm. Such transformations require communication across the computing cores. So, we focus on the calculation of the following two distributed matrix-vector products that involve inter-core communication processes *gather* and *scatter*.

$$\beta = \mathbf{Q}\hat{\beta} = \text{diag}(\mathbf{Q}_1, \dots, \mathbf{Q}_p) \times (\mathbf{Q}_f\hat{\beta})$$

and

$$\hat{\beta} = \mathbf{Q}^T\beta = \mathbf{Q}_f^T \times (\text{diag}(\mathbf{Q}_1^T, \dots, \mathbf{Q}_p^T)\beta)$$

In each iteration of the Stage 2 (Steps 3-11 in Algorithm 5.2), these two multiplication modules are invoked to transform $\hat{\beta}_i$ to β_i and vice-versa, respectively (see Step 6 and Step 8 in Algorithm 5.2). This can lead to large communication overhead. Algorithms 5.3 and 5.4 describe how to transform $\hat{\beta}$ to β and β to $\hat{\beta}$, respectively, in an efficient manner. It is worth restating that \mathbf{Q}_i 's and \mathbf{Q}_f are never calculated explicitly, rather, are stored as sets of k -Householder reflectors; as $\{\mathbf{q}_i\}$'s in the worker cores and as $\{\mathbf{q}_f\}$ in the master core. The implicit matrix-vector multiplication involving these reflectors is described in [100].

To compute $\beta = \mathbf{Q}\hat{\beta}$ in Algorithm 5.3, we first gather local $\hat{\beta}_i$ from the worker cores and calculate $(\mathbf{Q}_f\hat{\beta})$ vector at the master core. Next, the partitioned vector $(\mathbf{Q}_f\hat{\beta})_i$ is scattered to the cores. Core i uses its local \mathbf{Q}_i in the form of reflector set $\{\mathbf{q}_i\}$ that was pre-computed during the QR decomposition of the partitioned data $\hat{\mathbf{Q}}_i$ to multiply \mathbf{Q}_i to $(\mathbf{Q}_f\hat{\beta})_i$ in order to obtain β_i . These multiplications with \mathbf{Q}_i proceed concurrently at each core.

To compute $\hat{\beta} = \mathbf{Q}^T\beta$ in Algorithm 5.4, however, we first calculate local $(\mathbf{Q}_i^T\beta_i)$ in each core i . This computation can be trivially parallelized. Then, these local vectors are gathered at the master core. The gathered vector $(\text{diag}(\mathbf{Q}_1^T, \dots, \mathbf{Q}_p^T)\beta)$ is multiplied with \mathbf{Q}_f^T to generate $\hat{\beta}$ at the master core. Finally, we scatter it to all the cores such that each core gets its $\hat{\beta}_i$.

Communication improvement over Chapter 4: In the prior implementation in Chapter 4, significant communication overhead was incurred during $\mathbf{Q}\hat{\beta}$ and $\mathbf{Q}^T\beta$ operations in each iteration of the parallel dual ascent. Specifically, it required communicating large volume of data of

Algorithm 5.2 Parallel Dual Ascent

Data: Matrix \mathbf{F}_i , vector $\hat{\mathbf{e}}_i$, vector $\hat{\beta}_i^0 = \mathbf{0}$, number of parallel cores p

```
1 iteration  $t \leftarrow 0$ 
2 for each core  $i, \dots, p$  do
3   while error > threshold do
4     Parallel Compute Equation 4.10:  $\hat{\alpha}_i^{t+1} = \mathbf{F}_i^{-1}(-\hat{\beta}_i^t + \hat{\mathbf{e}}_i)$ 
5     Parallel Compute Equation 4.11:  $\hat{\beta}_i^{t+1} = \hat{\beta}_i^t + \eta^*( -\hat{\alpha}_i^{t+1} )$ 
6     Compute  $\beta_i \leftarrow \mathbf{Q}\hat{\beta}$  // Algorithm 5.3
7     Parallel Compute (element-wise):  $\beta_i \leftarrow \max\{0, \beta_i\}$ 
8     Compute  $\hat{\beta}_i \leftarrow \mathbf{Q}^T \beta$  // Algorithm 5.4
9     error  $\leftarrow |\hat{\beta}_i^{t+1} - \hat{\beta}_i^t|$ 
10     $t \leftarrow t + 1$ 
11  end
12 end
```

Algorithm 5.3 Compute $\beta_i \leftarrow \mathbf{Q}\hat{\beta}$, in each core i

Data: Matrices $\mathbf{Q}_i, \mathbf{Q}_f$, vector $\hat{\beta}_i$

```
1 GATHER  $\hat{\beta}_i$  at Master core
2  $\hat{\beta} \leftarrow$  gathered  $\hat{\beta}_i$ 
3 Compute  $(\mathbf{Q}_f \hat{\beta})$  at Master core
4 SCATTER  $(\mathbf{Q}_f \hat{\beta})$  to all cores
5  $(\mathbf{Q}_f \hat{\beta})_i \leftarrow$  scattered  $(\mathbf{Q}_f \hat{\beta})$ 
6 Parallel Compute  $\beta_i \leftarrow \mathbf{Q}_i \times (\mathbf{Q}_f \hat{\beta})_i$ 
```

Algorithm 5.4 Compute $\hat{\beta}_i \leftarrow \mathbf{Q}^T \beta$, in each core i

Data: Matrices $\mathbf{Q}_i, \mathbf{Q}_f$, vector β_i

```
1 Parallel Compute  $(\mathbf{Q}_i^T \beta_i)$ 
2 GATHER  $(\mathbf{Q}_i^T \beta_i)$  at Master core
3  $(\text{diag}(\mathbf{Q}_1^T, \dots, \mathbf{Q}_p^T) \beta) \leftarrow$  gathered  $(\mathbf{Q}_i^T \beta_i)$ 
4 Compute  $\hat{\beta} \leftarrow \mathbf{Q}_f^T \times (\text{diag}(\mathbf{Q}_1^T, \dots, \mathbf{Q}_p^T) \beta)$  at Master
5 SCATTER  $\hat{\beta}$  to all cores
6  $\hat{\beta}_i \leftarrow$  scattered  $\hat{\beta}$ 
```

size $\left(\frac{n}{p}\right)$ per core via gather and scatter processes in the distributed network. Increasing the number of cores resulted in more interactions across the network. As a result, the training time became prohibitive for large datasets thereby limiting the benefits of the distributed QRSVM framework. In the proposed implementation which is based on memory-efficient representation $\{\mathbf{q}_f\}$, it is sufficient to communicate (via gather and scatter) only the first k -elements of the partitioned vector, β_i or $\hat{\beta}_i$ in each of the above computation operations (Algorithms 5.3 and 5.4). This is a key step to achieve near-negligible communication overhead. As a result, in each iteration of the parallel dual ascent, the communication volume is reduced by a factor of $\left(\frac{n}{pk}\right)$ per core compared to the one in Chapter 4, without incurring any computation error. Moreover, the proposed implementation is rendered computation-bound only which can be handled efficiently by employing more number of parallel computing cores. Due to above communication improvements, we achieve large speedup in the SVM training and can handle significantly larger workloads, thereby making the framework scalable than the one in Chapter 4.

An alternative implementation for $\mathbf{Q}\hat{\beta}$ and $\mathbf{Q}^T\beta$ in the parallel dual ascent method (Stage 2) can be used to avoid the frequent gathering and scattering of the k -sized vectors β_i and $\hat{\beta}_i$ from each of the cores. One can *scatter* the reflector set $\{\mathbf{q}_f\} \in \mathbb{R}^{pk \times k}$ present in the master core across all the worker cores as $\{\mathbf{q}_f\}_i$ on completion of distributed QR decomposition (Stage 1). Each core can now locally compute $(\mathbf{Q}_{f_i}\hat{\beta}_i)$ and $(\mathbf{Q}_{f_i}^T\beta_i)$ via implicit multiplication described in [100]. With distributed $\{\mathbf{q}_f\}_i$ in each core, it is possible to parallelize the implicit matrix-vector $(\mathbf{Q}\mathbf{v})$ multiplication while simply using *All_Reduce* on the locally computed $\{\mathbf{q}_f\}_i^T\beta_i$ (scalar) values with *SUM* operation. In every iteration of the update steps, *All_Reduce* will be invoked k -times within every core for computing $(\mathbf{Q}_{f_i}\hat{\beta}_i)$ and $(\mathbf{Q}_{f_i}^T\beta_i)$. This alternative is expected to be beneficial for sufficiently large k . We leave the experimental evaluation with this alternate design for future analysis and beyond the scope of the current work.

5.4 Experiment and Results

Here we present experimental results to demonstrate the performance of the communication-efficient QRSVM framework for scalable SVM training.

Dataset	#training samples (n)	#features (d)	<i>k</i>-rank approx.
covtype.binary	464,810	54	64
webspam(unigram)	350,000	254	128
SUSY	5,000,000	18	128

Table 5.1: Benchmark dataset description. Reprinted with permission from [6].

5.4.1 Hardware Description

We run our parallel implementation on the *Ada* supercomputing cluster at the Texas A&M High Performance Research Computing² facility. The cluster consists of 792 compute nodes each equipped with two 10-core Intel Xeon E5-2670 v2 (Ivy Bridge) processors and 64 GB memory. The interconnect used is Infiniband. We use Message-Passing Interface (MPI) [102] for inter-process communication.

5.4.2 Experimental Setup

For our experiments, we use datasets which are available in LIBSVM datasets repository³ for binary classification. Specifically, we focus on large datasets with number of training samples in hundreds of thousand and above to justify the need for a distributed and scalable SVM framework. The RBF kernel function is used for SVM training to represent the Kernel matrix. The dataset description is provided in Table 5.1. We also report the *k*-rank approximation selected for the kernel matrix computed by MEKA. The distributed QRSVM framework has been implemented in C/C++ using the Armadillo library [34] integrated with LAPACK/BLAS for linear algebra calculations.

5.4.3 Results and Discussions

Here, we present our discussion on the various performance measures, namely, convergence, scalability, training time (with computation and communication analysis), of the improved and efficient implementation of the distributed QRSVM framework.

²<http://hprc.tamu.edu/>

³<https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html>

Parameters	covtype	webspam	SUSY
C	1	1	1
γ	2^3	2^0	2^{-3}
#cores, p	16	32	64
stopping threshold	10^{-3}	10^{-3}	10^{-3}
optimal step size, η^*	0.9	0.9	0.9
#iterations, t_c	1075	569	2096

Table 5.2: Parameter details for various datasets. Reprinted with permission from [6].

5.4.3.1 Convergence

The convergence of the distributed QRSVM technique is guaranteed by the parallel dual ascent method discussed in Section 4.4.3.2. In Chapter 4, we derived an optimal step size for faster convergence, which has been adopted in our experiments here. Figure 5.3 illustrates the convergence trend of the proposed algorithm for the three benchmarks. It exhibits a sharp decrease in the training error $\|\hat{\beta}^{t+1} - \hat{\beta}^t\|_1$ as the dual ascent algorithm converges to the optimal solution. A stopping threshold of 10^{-3} has been used for convergence. Various parameters for the selected benchmarks are shown in Table 5.2.

5.4.3.2 Scalability

Table 5.3 lists the training time and speedup achieved by distributed QRSVM on p cores, where, $p \in \{2, 4, 8, 16, 32, 64\}$. Since a processing node on *Ada* consists of two 10-core processors, we use one node for $p \in \{2, 4, 8, 16\}$, two nodes for $p = 32$, and four nodes for $p = 64$. When using more than a single node, the number of cores used on each node was identical. The only exception to this rule is the execution of SUSY, where the number of cores was restricted to 2 per node in order to leverage caches effectively on each node for large dataset. It is found that SVM on covtype, webspam and SUSY can be trained in as fast as 6 seconds, 9 seconds, and 210 seconds, respectively, on 64 cores. This results in speedup of around 45x, 29x and 136x over sequential QRSVM implementation on a single core. Speedup (S_p) is computed as the ratio of training time

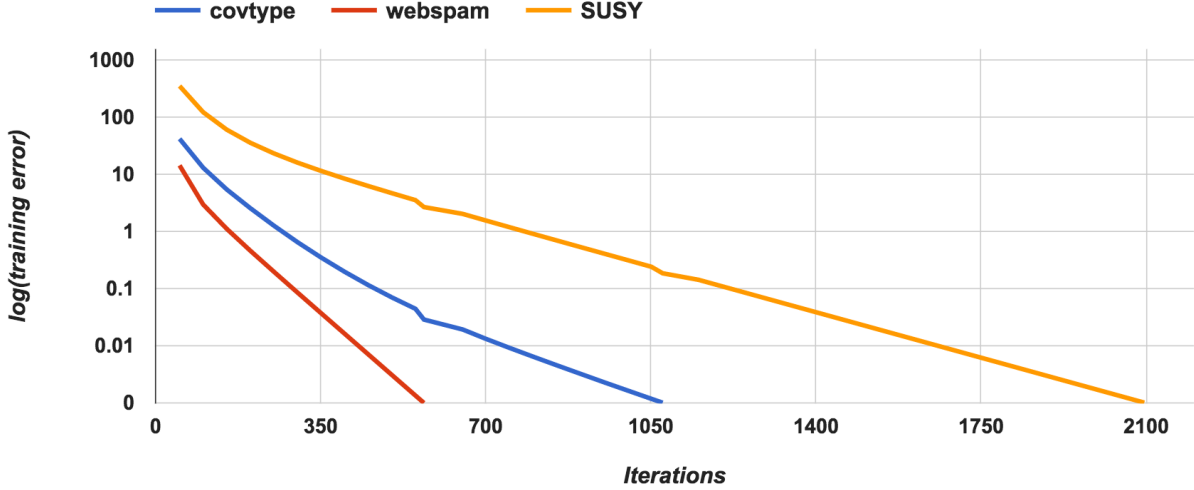


Figure 5.3: Convergence rate of *distributed QRSVM* algorithm: Training error on benchmark datasets (covtype, webspam and SUSY) approaches the stopping threshold (10^{-3}) in $t_c = 1075$, $t_c = 569$, and $t_c = 2096$ iterations, respectively, using the optimal step size $\eta^* = 0.9$. Reprinted with permission from [6].

(T_{train}) on p cores to that on a single core, i.e.,

$$S_p = \frac{T_{train|p}}{T_{train|p=1}},$$

where, $T_{train|p}$ is the QRSVM training time on p cores. Since Stage 1 (distributed QR decomposition) and Stage 2 (parallel dual ascent) of the distributed technique have been parallelized efficiently, we are able to achieve very high speedups with respect to the sequential execution. The two smaller benchmarks show near-linear speedup on up to 16 cores. The speedup observed for SUSY is highly dependent on cache utilization by the cores due to large problem size. SUSY demonstrates near linear speedup for $p = \{16, 32, 64\}$ when $p = 8$ is used as the base case since the subproblem on each core fits within the local cache. However, when $p < 8$ cores are used, the subproblem size on each core is too large to fit the local cache which results in relatively larger training time. As a result, one observes super-linear speedup as we increase the number of cores when $p = 1$ (sequential implementation) is used as the base case.

Dataset	sequential	p=2	p=4	p=8
covtype	268 (1x)	132 (2x)	64 (4x)	33 (8x)
webspam	258 (1x)	120 (2x)	58 (4x)	32 (8x)
SUSY	28,614 (1x)	11,284 (2x)	4,405 (7x)	1,686 (17x)

Dataset	sequential	p=16	p=32	p=64
covtype	268 (1x)	18 (15x)	10 (27x)	6 (45x)
webspam	258 (1x)	19 (14x)	11 (23x)	9 (29x)
SUSY	28,614 (1x)	804 (36x)	380 (75x)	210 (136x)

Table 5.3: Scalability of distributed-QRSVM. Training time (in seconds) and Speedup, S_p wrt sequential-QRSVM (S_p is shown in parenthesis). Reprinted with permission from [6].

5.4.3.3 Distributed Training Time

Here we demonstrate and discuss the results for various components of the distributed training time with respect to computation and communication analysis.

Computation time, T_{comp} : The distributed QRSVM framework for parallel training has three major computational requirements:

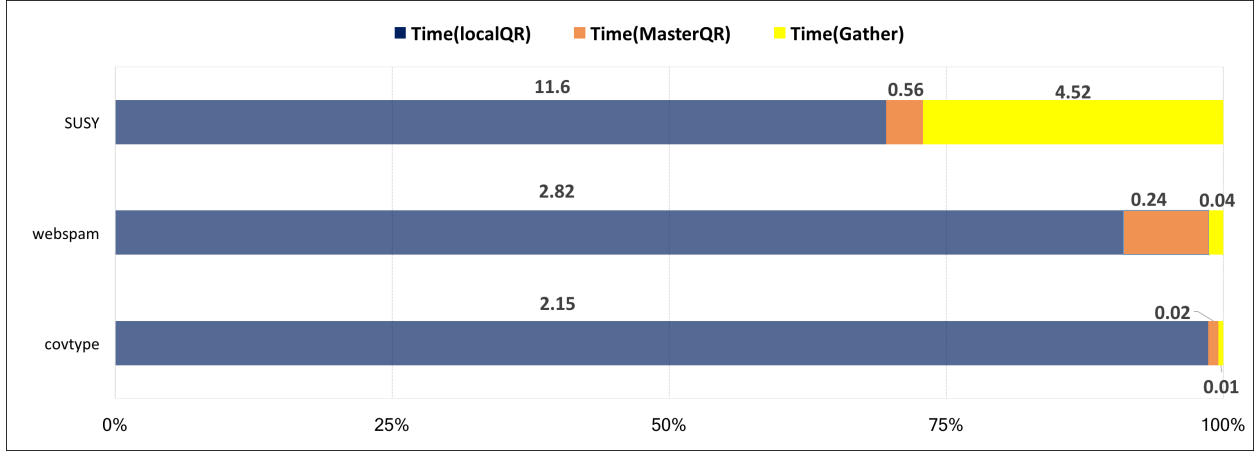
1. Low-rank kernel approximation
2. Distributed QR decomposition (Stage 1)
3. Parallel Dual ascent (Stage 2)

For computing the low-rank kernel approximation we use MEKA [5] which is a sequential code. The time to compute these approximations, denoted as T_{LRA} , has values 2.1 seconds, 5.93 seconds and 29.66 seconds for the benchmarks covtype, webspam and SUSY, respectively. Kernel approximation is a one-time pre-processing step prior to the other two more computationally dominant stages of the proposed framework. Therefore, we do not include T_{LRA} in the overall training time, and instead focus on the time spent on the other two stages. As a side note, it is also possible to use alternative kernel approximation techniques like [94], [95], [96], etc., but we recommend MEKA for its superior properties of memory-efficiency and low approximation error [5].

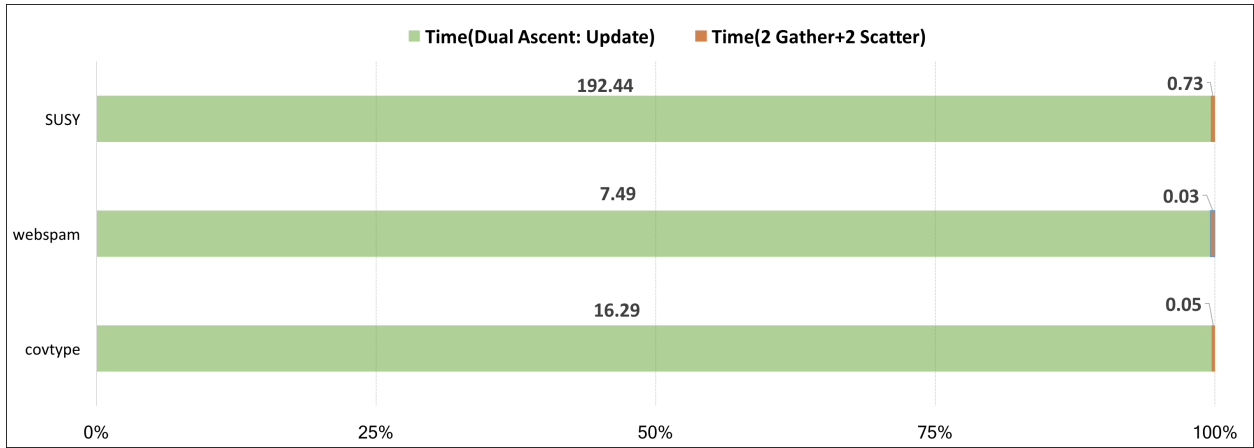
In the distributed QR decomposition stage (Algorithm 5.1), partitioned data $\hat{\mathbf{A}}_i$ is decomposed locally in each core into \mathbf{Q}_i and \mathbf{R}_i concurrently (Step 4 in Algorithm 5.1). We denote its worst-case computation time as $T_{localQR}$. Then, at the master core the gathered data $\hat{\mathbf{A}}_g$ is further factorized (see Step 8 in Algorithm 5.1), and its computation time is denoted as $T_{masterQR}$. Figure 5.4(a) depicts the timing distribution of the distributed QR decomposition, where $T_{localQR} = \{2.15, 2.82, 11.6\}$ seconds and $T_{masterQR} = \{0.02, 0.24, 0.56\}$ seconds for benchmarks covtype ($p = 16$), webspam ($p = 32$) and SUSY ($p = 64$), respectively. It can be seen that most of the total execution time for Stage 1 is spent on computing the local QR decomposition, $T_{localQR}$.

Finally, for each iteration in the parallel dual ascent stage, we locally compute the update steps given in Equation (4.10) and Equation (4.11) in parallel across the cores. They are denoted as Step 4 and Step 5 of Algorithm 5.2, respectively. Later, we compute the transformation from $\hat{\beta}$ to β and back to $\hat{\beta}$ in Step 6 and Step 8 of Algorithm 5.2 which are separately tagged as Algorithm 5.3 and Algorithm 5.4, respectively. Note that, Step 3 and Step 6 in Algorithm 5.3 and Step 1 and Step 4 in Algorithm 5.4 are the computation steps in the above transformations. We combine the above mentioned computation times for all the iterations of the parallel dual ascent until convergence and denote as T_{update} . Figure 5.4(b) presents the total execution time for the Stage 2 which includes both the computation and the communication time. Here, the computation time is $T_{update} = \{16.29, 7.49, 192.44\}$ seconds for benchmarks covtype ($p = 16$), webspam ($p = 32$) and SUSY ($p = 64$), respectively. It is observed that more than 99% of the execution time of parallel dual ascent (Stage 2) is spent on computation. Hence, the proposed algorithm is amenable for full parallelism by incorporating more cores.

Communication time, T_{comm} : The distributed QRSVM framework has two necessary communication requirements. The first communication occurs during Stage 1, i.e., distributed QR decomposition, when local $(R_i)_{k \times k}$ are gathered at the master core (Step 5 in Algorithm 5.1). This is also depicted in Figure 4.4. Let us denote this communication time as $T_{gatherR}$. As seen in Figure 5.4(a) the communication overhead, $T_{gatherR} = \{0.01, 0.04\}$ seconds hardly impacts the Stage 1 execution time for benchmarks covtype and webspam on $p = 16$ and $p = 32$ cores, respectively.



(a) Stage 1: Distributed QR decomposition



(b) Stage 2: Parallel Dual ascent

Figure 5.4: Timing (in seconds) analysis for computation and communication in different stages of distributed QRSVM. (a) Stage 1: $T_{localQR}$ and $T_{masterQR}$ are the computation time, whereas T_{gather} is the communication time. (b) Stage 2: T_{update} denotes the computation time for all iterations of parallel dual ascent, while T_{2g+2s} refers to the communication time spent in transformations $\hat{\beta}$ to β . Datasets: covtype ($p = 16$), webspam ($p = 32$), SUSY ($p = 64$). Reprinted with permission from [6].

It is due to minimal inter-node communication where the number of *Ada* nodes required are 1 and 2, respectively based on our experimental setup. In the case of larger benchmark SUSY, the inter-node communication is higher as 32 *Ada* nodes are used. Here, the communication requirement in Stage 1 is $T_{gatherR} = 4.52$ seconds on $p = 64$ cores, which is around 27% of the execution time of Stage 1. However, for such large datasets requiring more number of nodes, the increase

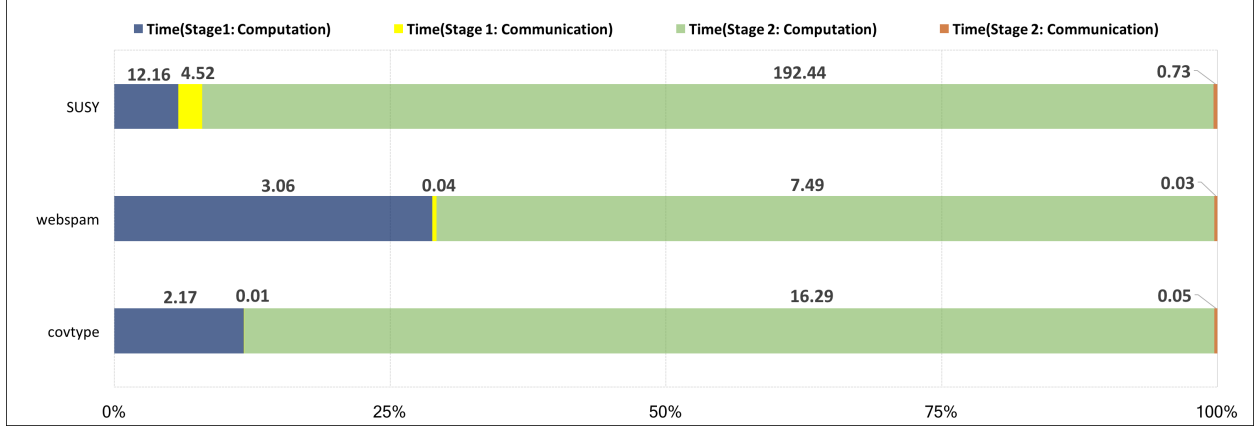


Figure 5.5: Overall training time, T_{train} analysis (in seconds) for benchmarks covtype ($p = 16$), webspam ($p = 32$) and SUSY ($p = 64$). A majority of the training time is spent in iterative computations in Stage 2 (parallel dual ascent), while communication overhead is negligible. Reprinted with permission from [6].

in inter-node communication time for Stage 1 is largely compensated by the computation time for Stage 2, as seen in Figure 5.5.

The second communication occurs during each iteration of the parallel dual ascent method (Step 6 and Step 8 in Algorithm 5.2). There are two gather and two scatter processes in each iteration as discussed in Section 5.3. These communication processes, gather and scatter, first occur in Step 1 and Step 4 of Algorithm 5.3, respectively and again in Step 2 and Step 5 of Algorithm 5.4, respectively. Let T_{2g+2s} denote the time involved for gathering (g) and scattering (s) twice during all the iterations of the parallel dual ascent until convergence. From Figure 5.4(b), we observe that $T_{2g+2s} = \{0.05, 0.03, 0.73\}$ seconds for benchmarks covtype ($p = 16$), webspam ($p = 32$) and SUSY ($p = 64$), respectively, has insignificant impact ($< 1\%$) on execution time of Stage 2 in comparison to its computation time, T_{update} .

Training time, T_{train} : The overall training time, denoted as $T_{train} = T_{comp} + T_{comm}$, for the distributed QRSVM algorithm is observed to be $T_{train} = \{18.52, 10.62, 209.85\}$ seconds on benchmarks covtype ($p = 16$), webspam ($p = 32$) and SUSY ($p = 64$), respectively. In the proposed framework, T_{train} comprises of computation time, i.e., $T_{comp} = T_{localQR} + T_{masterQR} + T_{update} = \{18.46, 10.55, 204.60\}$ seconds, and communication time, i.e., $T_{comm} = T_{gatherR} + T_{2g+2s} =$

$\{0.06, 0.07, 5.25\}$ seconds. Figure 5.5 depicts the analysis of T_{train} where the time for communication is around 0.3%, 0.6% and 2.5% of the total training time for benchmarks covtype, webspam and SUSY respectively. Moreover, in Figure 5.5, it is also observed that parallel dual ascent (Stage 2) in the QRSVM algorithm is more computationally dominant stage than the distributed QR decomposition (Stage 1). On Ivybridge processor, parallel dual ascent achieves around 700 MFLOP per second per core for the chosen benchmarks. These results validate our communication-efficient implementation of distributed QRSVM where negligible amount of SVM training time is spent in communicating data across the network. Hence, our framework is a step towards distributed training on the edge for applications in IoT without transferring the data to a central server for training. In essence, our framework offers decentralization of SVM training with guaranteed convergence.

5.4.3.4 Comparison

In Table 5.4, we compare the training time of the proposed framework with state-of-the-art parallel solvers, namely, PSVM [12] and P-packSVM [88] on covtype benchmark. Moreover, we also evaluate the performance improvement over the prior implementation of distributed QRSVM framework in Chapter 4. It is observed that for a given benchmark, the proposed implementation of the distributed QRSVM trains much faster than PSVM, P-packSVM and prior implementation in Chapter 4. Both PSVM and the distributed QRSVM optimize the dual form of the SVM cost function, making our comparison fair. We have used the default setting for PSVM parameters as reported in [12]. In our experiments, the dual residual threshold for convergence was set to 10^{-3} .

PSVM is the foremost parallel SVM solver available as open source. It employs parallel incomplete cholesky factorization (PICF) to approximate $n \times n$ kernel matrix with a k -rank representation. PSVM then performs parallel Interior-Point Method to solve the quadratic optimization problem. As per [12], the computational complexity of PSVM is $O(nk^2/p)$. To ensure good accuracy, the authors in PSVM [12] recommend the rank of the ICF of kernel matrix as $k = \sqrt{n}$, where n represents the number of training samples. As a result, the cost of executing PSVM becomes quadratic in n which results in large training time and hinders its scalability to large datasets.

In contrast to PSVM, the proposed distributed QRSVM framework builds on the current state-

Algorithm	p=2	p=4	p=8	p=16	p=32	p=64
PSVM [12]	8,562	4,396	2,352	1,270	635	341
P-packSVM [88]	-	-	2,019	1,022	295	110
<i>dis</i> -QRSVM (Chapter 4)	390	309	271	261	256	454
<i>dis</i> -QRSVM (improved)	132	64	33	18	10	6

Table 5.4: Comparing *dis*-QRSVM with PSVM, P-packSVM and Chapter 4 framework on T_{train} (in seconds) for *covtype* dataset. Here, - represents data is unavailable. Reprinted with permission from [6].

of-the-art kernel approximation, MEKA [5], which is both memory-efficient and has the least kernel approximation error amongst various approximation methods such as ICF. Since the accuracy of our framework is directly related to the quality of kernel approximation, we can safely argue that accuracy of our MEKA-based distributed QRSVM will be at least at par or better than the traditional ICF-based PSVM. In the light of above, the k -rank approximation of the kernel matrix is chosen empirically through MEKA [5]. We ensure rank $k \ll n$ as discussed in Section 4.3. Since the dominant computations in both stages of the distributed QRSVM framework can be fully parallelized, the time complexity of the proposed framework is $O(nk^2/p + nkt_c/p)$. With the above choice of $k \ll n$, the computational cost for the distributed QRSVM is linear in number of samples n unlike PSVM which shows quadratic behavior. Therefore, the proposed framework converges faster and is more scalable than PSVM. For instance, it can be observed from Table 5.4 that on $p = \{2, 4, 8, 16, 32, 64\}$ cores for *covtype* benchmark, PSVM takes $T_{train} = \{8562, 4396, 2352, 1270, 635, 341\}$ seconds while the proposed distributed QRSVM framework trains in $T_{train} = \{132, 64, 33, 18, 10, 6\}$ seconds with performance improvement of $\{65x, 69x, 71x, 71x, 63x, 57x\}$.

Due to lack of availability of P-packSVM code as open source, we estimate its results based on its training time ratio with respect to PSVM obtained from [88]. We report the training performance of P-packSVM on $p = \{8, 16, 32, 64\}$ cores since the base number of cores used in [88] is 8. As observed in Table 5.4, our implementation performs $\{61x, 57x, 30x, 18x\}$ faster than P-packSVM for training *covtype* on $p = \{8, 16, 32, 64\}$ cores, respectively.

Time	p=2	p=4	p=8	p=16	p=32	p=64
T_{update} (Chapter 4)	379	304	269	259	252	448
T_{update} (improved)	122	59	30	16	8	5
T_{2g+2s} (Chapter 4)	1.63	1.81	0.34	0.05	1.19	3.02
T_{2g+2s} (improved)	0.02	0.02	0.14	0.05	0.03	0.11

Table 5.5: Comparing the improved *dis*-QRSVM with prior implementation in Chapter 4 on Stage 2 computation time, T_{update} and communication time, T_{2g+2s} (in seconds) for *covtype* dataset. Reprinted with permission from [6].

We also compare and evaluate the performance of the proposed implementation with our prior framework in Chapter 4. Section 5.3 discusses the communication improvement over Chapter 4 framework during the parallel implementation of the dominant stage of iterative dual ascent (Stage 2). Hence, we compare Stage 2 computation time, T_{update} and communication time, T_{2g+2s} of both the implementations in Table 5.5. It can be empirically observed that the proposed implementation significantly improves over Chapter 4 implementation in computation of Stage 2 with relatively lower communication overhead, leading to faster overall training time. From Table 5.4, we achieve performance improvement of $\{3x, 5x, 8x, 14x, 25x, 75x\}$ in training time compared to Chapter 4 implementation on $p = \{2, 4, 8, 16, 32, 64\}$ cores, respectively. While the prior framework in Chapter 4 could train datasets just as large as *covtype* with $n = 464, 810$ samples, the proposed implementation is demonstrated to work for same and even larger datasets such as *SUSY* with $n = 5M$ samples. Moreover, the results in Table 5.5 also validate that prior framework was unable to scale on large number of cores while the proposed approach exhibits better parallel speedup, owing to both memory and communication-efficient implementation.

5.5 Summary

In this chapter, we proposed a fast and efficient implementation to improve the distributed QR decomposition framework in Chapter 4 for scaling the kernel SVM training. The framework comprises of two stages: distributed QR decomposition and parallel dual ascent, to accelerate the training. We efficiently implement these two stages of the framework to achieve significant memory and communication benefits over the prior framework. We empirically demonstrate that the

proposed distributed implementation of the improved framework achieves considerable speedup over its sequential counterpart. In addition, it substantially reduces the communication overhead to a negligible fraction of the total training time. These characteristics make the framework suitable to handle large data problems while being scalable across large number of computing cores, unlike the one in Chapter 4. We also achieve performance benefit on training time compared to state-of-the-art parallel SVM solvers. In the next chapter, we will design an efficient hardware accelerator based on distributed QRSVM for applications in embedded edge computing.

6. MULTIPLE FPGA-BASED SYSTEM FOR ENERGY-EFFICIENT TRAINING ¹

Training machine learning models on a large number of data samples is challenging due to the high computational cost and memory requirement. Hence, model training is supported on a high-performance server which typically runs a sequential training algorithm on centralized data. However, as we move towards massive workloads, it will be impossible to store all the data in a centralized manner and expect such sequential training algorithms to scale on traditional processors. Moreover, with the growing demands of real-time machine learning for edge analytics, it is imperative to devise an efficient training framework with relatively cheaper computations and limited memory. In this chapter, we propose and implement a first-of-its-kind system of multiple FPGAs to accelerate distributed QRSVM using FPGA as hardware accelerator with a focus on energy efficiency.

6.1 Introduction

Support Vector Machine (SVM) is a supervised machine learning technique with strong geometrical and statistical properties to solve classification and regression tasks. Kernel SVM classifier is used for non-linear classification by transforming the data from input space to high-dimensional feature space such that the transformed data is separable by a hyperplane. As a machine learning model, SVM comprises two phases; a *training* phase to learn a classifier model for a given input dataset, and an *inference* phase, which uses the trained model for classifying or predicting an unseen test sample. For high-dimensional data that have an inherent well-defined data pattern, although deep neural networks (DNNs) have become popular with large availability of training data and powerful computing platforms, they are not the best solution in all domains compared to SVM as discussed below. For example, a tuned SVM performs similar to convolutional neural networks with significantly faster training while utilizing far fewer compute resources

¹This chapter is reprinted with permission from “Distributed Training of Support Vector Machine on a Multiple-FPGA System” by Jyotikrishna Dass, Yashwardhan Narawane, Rabi N. Mahapatra, and Vivek Sarin. 2020 IEEE Transactions on Computers (TC), Copyright © 2020, IEEE.

[105]. When classifying remote sensing images, it was shown that SVM can in fact perform better than a deep learning counterpart [106]. Another major concern with DNNs is that they are widely considered as black-box models suffering from the interpretability of their decisions. In contrast, SVM has strong geometrical and statistical properties that allow the decisions of the SVM model to be easily interpreted from well-determined decision boundaries [65]. Moreover, training DNN requires a careful and time-consuming process of optimally engineering the meta-parameters such as learning rate, momentum, weight-cost, number of hidden units, etc failing which the model suffers from over-fitting and poor generalization. As a more efficient alternative, authors in [107] proposed stacking multiple SVM (a single SVM can be modeled as a two-layer neural network) to extract high-order discriminative features using support vectors thereby guaranteeing generalization with far fewer user-determined parameters. Other studies have incorporated SVM as a classifier with features extracted from traditional DNN for performance gain [108, 109, 110]. In light of the above popularity and benefits, we consider SVM as our machine learning model. Having said so, the proposed techniques can be used for linear regression, linear SVM, any kernel-based machine learning problems such as ridge regression, and extended to novel hybrid learning models incorporating SVM.

6.1.1 Motivation

Training SVM for large datasets is challenging due to the high memory and computational cost associated with storing and computing with the dense kernel matrix [65]. Moreover, most modern SVM solvers are either inherently sequential [14] or are inefficiently parallel [12] which limit their scalability and make them infeasible for supporting massive workloads in cloud server. To design highly efficient cloud computing solutions for such massive workloads of future, it will be necessary to decentralize the data across multiple hardware units and provide a highly scalable training framework. In addition, with the explosion of powerful smart devices where each device is collecting its own data, there is a growing demand for real-time machine learning to train and update the model with low latency and data privacy at the edge. However, each edge device is relatively limited in computing and memory resources to independently support the demands of

SVM model training. Hence, to enable efficient and scalable model training for future workloads on cloud server and for providing real-time analytics on edge, we envision a distributed computing framework to fully parallelize and scale the model training on decentralized data. This will require pooling the resources of multiple computing units in a distributed environment while minimizing network communication overhead. Each computing unit in such distributed framework will comprise energy-efficient hardware design synthesized for accelerating its component of the overall training task on its local data thereby ensuring data privacy. Hence, it is imperative to devise a distributed training algorithm and to codesign a system of connected hardware accelerators such as FPGAs for fast and memory-efficient training of machine learning models such as SVM that can be used either in cloud servers or on connected edge devices.

6.1.2 Contributions

To satisfy the requirements for future cloud and edge-based services would require a distributed model training with decentralized data. In addition, a huge gap exists in research to provide a scalable co-designed network of hardware solutions for accelerating the training of SVM under such a distributed setting. In light of the above, we propose a first-of-its-kind *distributed SVM training framework comprising a multiple-FPGA system* to accelerate computationally challenging training phase on decentralized data. In this work, we use a cloud-based testbed as a proof-of-concept for the proposed framework due to the easy availability of multiple FPGAs. However, we do ensure effective software-hardware codesign architecture for cheap computations, less memory usage, and near negligible communication overhead. This chapter makes the following contributions:

1. We present a novel multiple-FGPA system for accelerating distributed SVM training in Section 6.3 with negligible communication overhead. In particular, we synthesize a pipelined SVM training logic core in Section 6.4 to accelerate QR-decomposition and dual ascent stages of the memory-efficient training algorithm on each FPGA. To the best of our knowledge, this is the first-of-its-kind hardware implementation of any distributed SVM training algorithm across multiple FPGAs.

2. We perform an extensive performance evaluation based on training time, parallel speedup, scalability, and energy efficiency of our proposed architecture in Section 6.5 for five real SVM benchmarks and varying number of FPGA units. We observe near-linear scalability on increasing number of FPGA units both in terms of training time and energy consumption.
3. We compare the training time and the energy consumption of the proposed FPGA solution in Section 6.5.4.4 with a commercially available embedded CPU platform for edge and widely available cloud processor. On SVM benchmark datasets, our hardware design in a multiple-FPGA system trains SVM computationally *faster* and is *more energy-efficient* compared to the corresponding cluster of these other platforms.

6.2 Related Work

Many FPGA-based architectures for SVM have been designed for accelerating the inference phase that uses pre-trained classifier models [111, 112, 113, 114]. However, a little attempt has been made in the literature on designing any hardware accelerator for the computationally challenging training phase [115, 116, 117, 118, 119]. Initial work [115] explored the nearest point approach for SVM training using the Gilbert algorithm and only mapped the datapath to FPGA. Moreover, no provision was made for datasets that do not fit the available block rams. Authors in [116] improved upon the above work to handle large datasets by using batches of data samples and applying the ensemble SVM training approach. However, [116] uses a single FPGA to train independent SVM models on these batches sequentially, which are later aggregated. The authors in [117] designed an FPGA-based co-processor for a popular SVM solver called Sequential Minimal Optimization [14]. However, due to the sequential nature of SMO, their design is neither feasible to leverage parallelism and achieve higher training speedups for large datasets nor it is amenable for training with decentralized data. There has also been some work on devising new SVM training algorithms solely for the purpose of designing dedicated FPGA implementations [118, 119]. However, these algorithms are limited to single FPGA implementation and are infeasible for distributed SVM training on a system of multiple computing units.

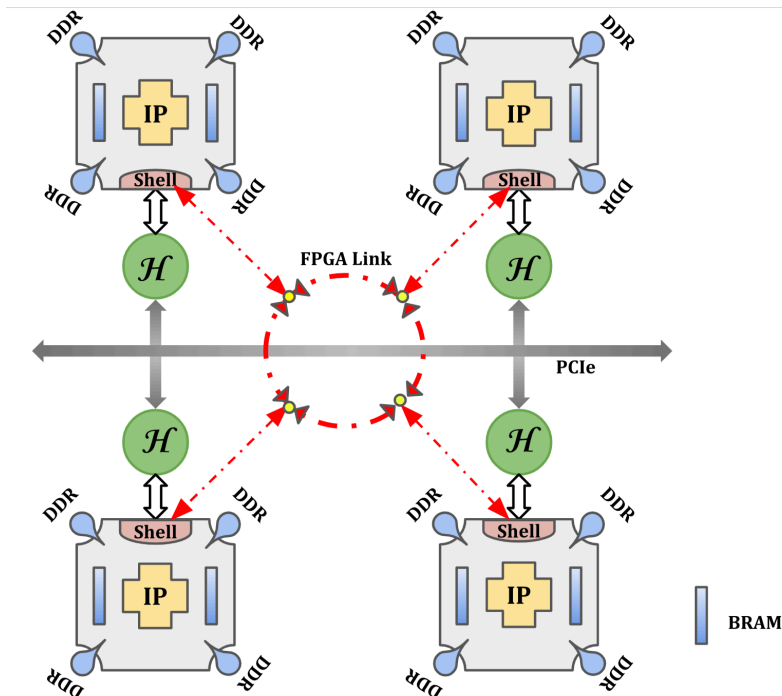


Figure 6.1: Illustration of a multiple-FPGA system with $p = 4$ compute units: Here, each unit comprises an FPGA that has IP logic along with a host. Each FPGA has internal memory (BRAM) and external DRAM (DDR) memory. The communication among compute units is either through a bi-directional ring (FPGA Link) or via PCIe bus interconnection. Reprinted with permission from [3].

6.3 System Overview

We envision a distributed network where each computing unit comprises an FPGA along with a local host processor. The host handles the flow of control and communication with other computing units in the network. Figure 6.1 illustrates a scenario with four such FPGA units. In the multiple-FPGA system for distributed SVM training, each computing unit stores its private data \hat{A}_i in its external DRAM (DDR). Then, all these units collaborate with each other as per the QRSVM algorithm [6] to train the complete SVM model. The algorithmic computations involved in training is implemented as a QRSVM IP logic core on an FPGA. This QRSVM IP enables the FPGA unit to accelerate all the training computations locally. As depicted in Figure 6.1, each FPGA has an interface (Shell) which enables communication with other FPGAs in the network, whenever

required. Such communication among the compute units is either *direct* through a bi-directional ring (FPGA link), or *indirect* through a host device connected over a PCIe bus. In indirect communication via the host, it is to be noted that the host processor will not be participating in any compute acceleration or storage of data. In fact, our goal is to build an architectural design fully capable of handling all computations and memory requirements by itself. Since the communication overhead of the algorithm is less than 1% as shown later in Section 6.5, the host can be any low-power commercial off the shelf embedded microprocessor in practice.

6.4 Accelerator Design

In this section, we describe the FPGA hardware design for computational kernels in the QRSVM IP logic core. Subsequently, we discuss the interfacing of the IP with the host and the memory.

6.4.1 Microarchitecture

To design an efficient architecture, we aim to synthesize specific hardware kernels for computations involved in the distributed QR decomposition and parallel dual ascent stages. As noted before, our implementation assigns all the computation to the FPGA fabric. It is to be reiterated that our focus is not on a heterogeneous computing solution where both the FPGA and the host share the computation workload via load balancing techniques for peak performance. The optimality of such systems is highly subjective and dependent on the heterogeneity, resource availability, and computing power of different types of hardware platforms (FPGA, CPU, GPU, etc), and how well one can extract acceleration and efficiency via load balancing. Rather, our aim is to design a homogeneous multiple FPGA framework by pitching FPGA as a complete hardware candidate for training SVM model that is sufficient enough to handle computationally intensive tasks all by itself, which typically is done on general-purpose CPUs. We will now describe the architectural design for accelerating the different stages of the QRSVM algorithm, namely, distributed QR decomposition and parallel dual ascent.

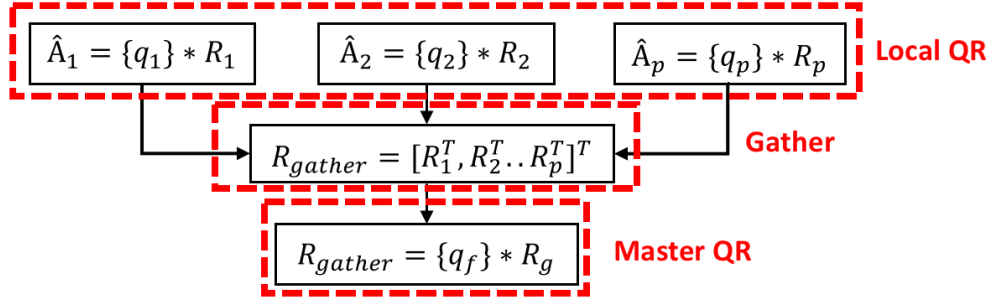


Figure 6.2: Computational flow graph for Distributed QR Decomposition. Reprinted with permission from [3].

6.4.1.1 Architecture for Distributed QR Decomposition

Here we describe the architectural design for enabling distributed QR decomposition stage. It is worth noting that the same architectural principles are easily extended to QR-based equivalent machine learning problem formulations such as linear regression, linear SVM, ridge regression, and other quadratic programming based numerical optimization problems. Figure 6.2 depicts the computational flow graph associated with distributed QR decomposition for kernel SVM. Firstly, local QR decomposition of \hat{A}_i is carried out in parallel across all the p FPGA units. These FPGA units are designated as *workers* for the purpose of algorithmic computation. Next, the locally generated upper triangular matrices, R_i 's are gathered at one of the FPGA units designated as the *master* to form R_{gather} . Finally, QR decomposition of R_{gather} is performed at the master FPGA.

It is to be noted that all these p FPGA units perform the same computation, i.e., QR decomposition via Householder reflectors, as detailed in Algorithm 6.1. Hence, the hardware design for this stage can be replicated at all FPGA units. From Algorithm 6.1, candidates for hardware acceleration are:

- Computing $\|\mathbf{q}_{ij}\|_2$: The ℓ_2 norm can be computed as $\|\mathbf{q}_{ij}\|_2 = \sqrt{\langle \mathbf{q}_{ij}, \mathbf{q}_{ij} \rangle}$.
- Updating \hat{A}_i : This can be modeled as a vector-matrix product ($\mathbf{q}_{ij}^T \hat{A}_i$), followed by a rank-1 update: $\hat{A}_i \leftarrow \hat{A}_i - 2\mathbf{q}_{ij}(\mathbf{q}_{ij}^T \hat{A}_i)$ detailed in Algorithm 6.2.

Algorithm 6.1 $\{\mathbf{q}_i\}, \mathbf{R}_i \leftarrow \hat{\mathbf{A}}_i$, via Householder algorithm

Data: Matrix $\hat{\mathbf{A}}_i$

1 $\mathbf{Q}_{\hat{n} \times k}, (\hat{\mathbf{A}}_i)_{\hat{n} \times k}$ $\triangleright \hat{n}$: samples per compute unit

2 **for** $j \leftarrow 1$ **to** k **do**

3 $\mathbf{q}_{ij} \leftarrow \hat{\mathbf{A}}_i(j : \hat{n}, j)$

4 $\mathbf{q}_{ij}(1) \leftarrow \mathbf{q}_{ij}(1) + \text{sign}(\mathbf{q}_{ij}(1)) \times \|\mathbf{q}_{ij}\|_2$ \triangleright scalar update

5 $\mathbf{q}_{ij} \leftarrow \frac{\mathbf{q}_{ij}}{\|\mathbf{q}_{ij}\|_2}$ \triangleright vector normalization

6 $\hat{\mathbf{A}}_i(j : \hat{n}, j : k) \leftarrow \hat{\mathbf{A}}_i(j : \hat{n}, j : k) - 2\mathbf{q}_{ij} \langle \mathbf{q}_{ij}, \hat{\mathbf{A}}_i(j : \hat{n}, j : k) \rangle$ \triangleright Algorithm 6.2

7 $\mathbf{R}_i = \hat{\mathbf{A}}_i(j : \hat{n}, j : k)$

8 **end**

9 $\{\mathbf{q}_i\} \leftarrow [\mathbf{q}_{i1}, \mathbf{q}_{i2}, \dots, \mathbf{q}_{ik}]$ \triangleright set of k-reflectors

Computation of the ℓ_2 -norm and rank-1 update involves two BLAS Level-1 functions: (a) Inner Product $sum = \langle \vec{x}, \vec{y} \rangle$ and (b) Scaled vector addition (saxpy), $\vec{x} = \vec{x} + \alpha\vec{y}$. The high degree of data parallelism inherent to these operations is leveraged to develop vectorized hardware implementations. FPGAs are amenable to such Single Instruction Multiple Data (SIMD) implementations, given their reconfigurable nature and high internal memory bandwidths. As an illustration, Figure 6.3 shows the architectures for computing inner product and saxpy. The proposed implementation is a pipelined design to increase throughput, where all arithmetic units at the same depth of the binary reduction tree are in one pipeline stage. This allows a given stage to process the next samples without waiting for the completion of all succeeding stages.

Pipelining: In the inner product kernel depicted in Figure 6.3, entries from \vec{x} and \vec{y} are multiplied at the leaf nodes of the binary reduction tree. The resulting products are pairwise summed along the tree branches and finally added to the previously computed partial product (denoted here by sum). For the inner product kernel, the number of pipeline stages can be determined as follows: Let us denote N as the data bus width and B as the bit width of a single element (32 bits for single-precision floating-point, 64 bits for double precision). The maximum number of leaf nodes, $W = \lfloor \frac{N}{B} \rfloor$, and number of pipeline stages, $D = \log_2 W$. For the saxpy operation, the pipeline depth is $D = 1$ since it is an element-wise operation. Hence, we can deploy a maximum of W adders and multipliers in parallel. Figure 6.3 shows the proposed designs for $W = 4$ elements in

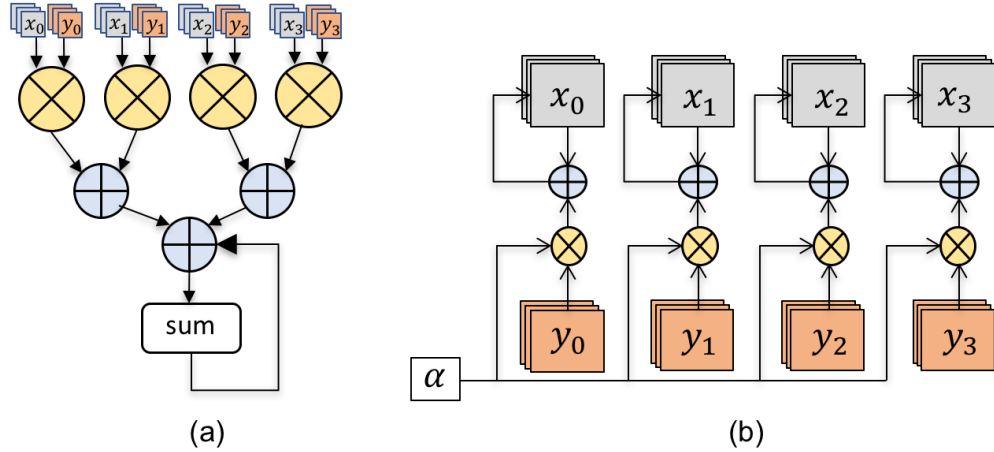


Figure 6.3: Hardware kernels for (a) inner product, $\langle \vec{x}, \vec{y} \rangle$ (b) saxpy, $\vec{x} = \vec{x} + \alpha\vec{y}$. The vectorized kernels process $W = 4$ elements in each pass. Reprinted with permission from [3].

each clock cycle (pass).

Data Layout: The BLAS operations in Algorithm 6.1 operate on columns of the partitioned data \hat{A}_i . Hence, we store the data elements in a column-major order which results in the contiguous memory access pattern, thereby, reducing the memory access time. As seen in Figure 6.4, the kernels in IP module access W column-elements of the data matrix from off-chip memory (DDR) during each clock cycle (or batch). Consequently, while storing data on chip (BRAM), we ensure that the column length is an integer multiple of W . This is to ensure that data elements from two different columns do not end up getting processed in the same computation batch. Accordingly, we pad columns with the requisite number of zeros wherever necessary.

Memory: The hardware IP interfaces with two types of memories: off-chip Random Access Memory (DDR), and on-chip Block RAM (BRAM). BRAM offers faster access time than the DDR. BRAM is generally a few Megabytes (MB) of configurable memory. Moreover, BRAM is configured in full-duplex mode with concurrent reads and writes in the same clock cycle, thereby avoiding pipeline stalls. In contrast, the memory interface of DDR with the IP is made half-duplex as illustrated in Figure 6.4. Based on the specific kernel operations, we allocate the operands to the appropriate memory type. For instance, in the inner product kernel illustrated in Figure 6.3(a), all

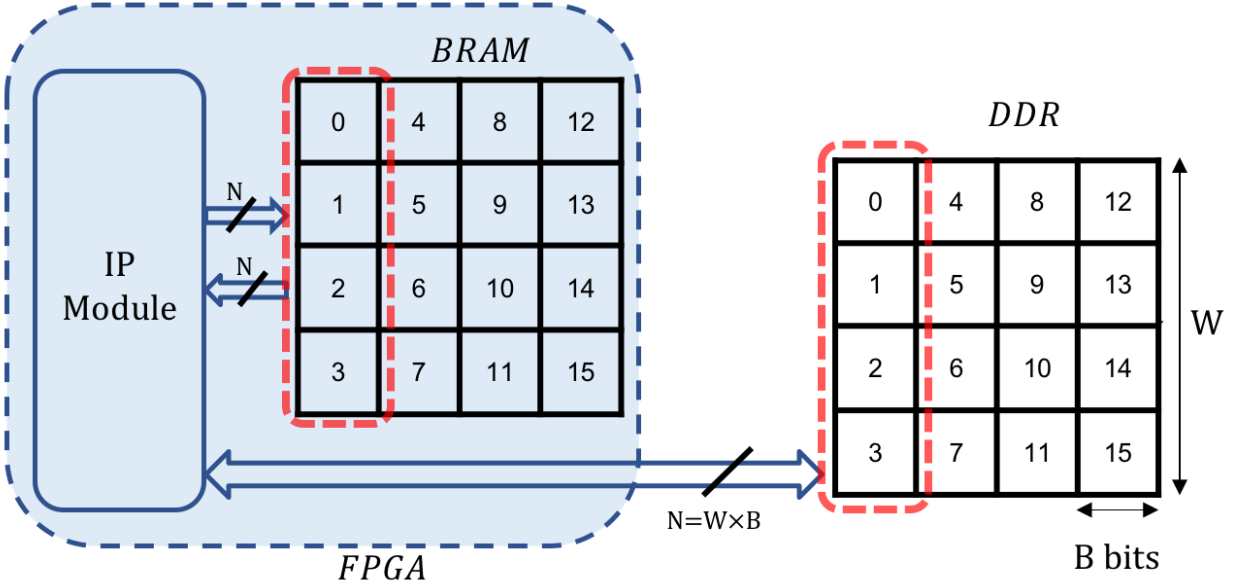


Figure 6.4: Data layout in column-major order and memory interface for on-chip Block RAM (Full-duplex) and off-chip DDR (Half-duplex) with the IP. Reprinted with permission from [3].

vector operations with \vec{x} and \vec{y} are read-only while the output variable sum is both read and write. Hence, for an efficient inner product module design, the vectors can be stored in either BRAM or DDR while the output is allocated to full-duplex BRAM. The saxpy kernel shown in Figure 6.3(b) requires \vec{x} to be both read and written simultaneously while \vec{y} is read-only. Hence, \vec{x} is stored in full-duplex BRAM while \vec{y} is allocated to half-duplex DDR in saxpy. In the context of the rank-1 update of \hat{A}_i which is a saxpy operation presented in Algorithm 6.2, we denote the vector's subscript to represent its allocated memory.

6.4.1.2 Architecture for Parallel Dual Ascent

Now, we discuss architecture design for accelerating the iterative computations in the parallel dual ascent stage. Recall, each iteration of the parallel dual ascent stage comprises updating variables $\hat{\alpha}$ and $\hat{\beta}$, transforming $\hat{\beta}$ to β , ensuring non-negativity on β (ReLU function), converting β back to $\hat{\beta}$ and finally estimating the difference in consecutive dual updates to decide if to iterate. Figure 6.5 illustrates the computational flow for this stage in detail.

Algorithm 6.2 Computing Step 6 in Algorithm 6.1

Data: Matrix $\hat{\mathbf{A}}_i$

```

1                                     ▷  $\hat{\mathbf{A}}_i(j : \hat{n}, j : k) \leftarrow \hat{\mathbf{A}}_i(j : \hat{n}, j : k) - 2\mathbf{q}_{ij} \langle \mathbf{q}_{ij}, \hat{\mathbf{A}}_i(j : \hat{n}, j : k) \rangle$ 
2 for  $m \leftarrow j$  to  $k$  do
3    $\mathbf{a}_{BRAM} \leftarrow \mathbf{A}(j : \hat{n}, m)$                                      ▷ Load into BRAM
4   Compute  $sum = \langle \mathbf{q}_{ij}, \mathbf{a}_{BRAM} \rangle$                                ▷ Inner Product
5    $\mathbf{a}_{BRAM} \leftarrow \mathbf{a}_{BRAM} - 2 \times sum \times \mathbf{q}_{ij}$                  ▷ saxpy
6    $\mathbf{A}(j : \hat{n}, m) \leftarrow \mathbf{a}_{BRAM}$                                    ▷ Write to DDR
7 end

```

Updating $\hat{\alpha}$ requires a vector subtraction, followed by pre-multiplication of \mathbf{F}^{-1} . The structure of \mathbf{F}^{-1} is given as the initial setup in Figure 6.5. At the master unit, \mathbf{F}_1^{-1} a dense $k \times k$ block followed by a diagonal sub-matrix block. On the worker units $i = \{2, \dots, p\}$, \mathbf{F}_i^{-1} is a diagonal matrix. Hence, at the master unit, the top k elements of $\hat{\alpha}_1$ are obtained by solving

$$\mathbf{L}\mathbf{L}^T \times \hat{\alpha}_{1[1:k]} = (\hat{\mathbf{e}}_1 - \hat{\beta}_1)_{[1:k]}$$

where \mathbf{L} is the Cholesky factor of \mathbf{F}_1 . i.e., $\mathbf{F}_1 = \mathbf{L}\mathbf{L}^T$ [57]. In contrast, remaining elements of $\hat{\alpha}_1$ and $\hat{\alpha}_i$, $i > 1$ can be computed by multiplying corresponding entries of $(\hat{\mathbf{e}} - \hat{\beta})$ with the scalar $(-2C)$. As illustrated in Figure 6.5, updating $\hat{\beta}$ is a saxpy operation and it must be stored in full-duplex BRAM.

By using Algorithms 5.3 and 5.4, $\hat{\beta}$ is transformed to β by computing

$$\beta = \mathbf{Q}\hat{\beta} = \text{diag}(\mathbf{Q}_1, \mathbf{Q}_2, \dots, \mathbf{Q}_p) \times \mathbf{Q}_g \times \hat{\beta}$$

while

$$\hat{\beta} = \mathbf{Q}^T\beta = \mathbf{Q}_g^T \times \text{diag}(\mathbf{Q}_1^T, \mathbf{Q}_2^T, \dots, \mathbf{Q}_p^T) \times \beta$$

is used for re-transformation. On each FPGA unit i , the matrix \mathbf{Q}_i is stored as a set of Householder reflectors, $\{\mathbf{q}_i\} = [\mathbf{q}_{i1}, \dots, \mathbf{q}_{ij}, \dots, \mathbf{q}_{ik}]$, where, $\mathbf{q}_{ij} \in \mathbb{R}^{\hat{n} \times 1}$ represents j^{th} reflector in the set. Hence, the most basic arithmetic operation in $\beta = \mathbf{Q}\hat{\beta}$ is the product of a reflector set $\{\mathbf{q}_i\}$ and a vector

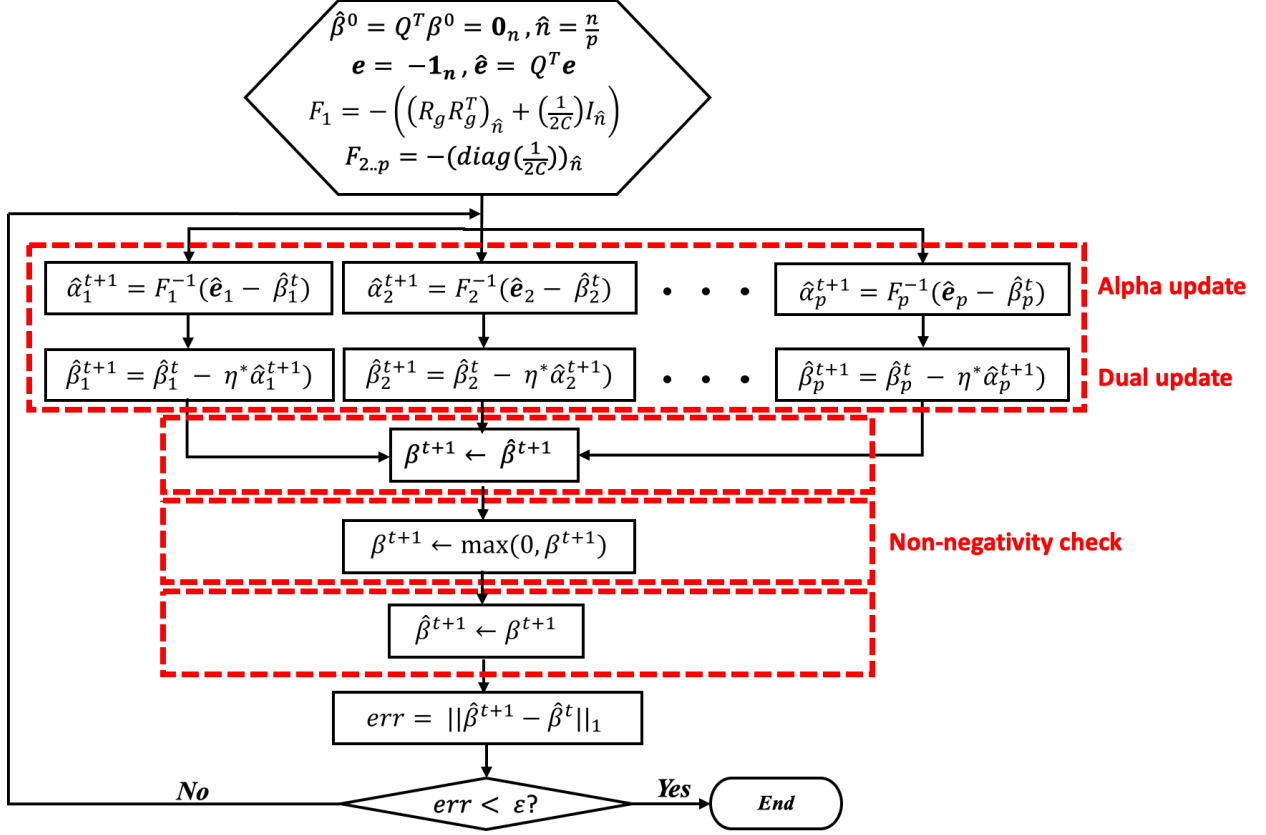


Figure 6.5: Computational flow graph for Parallel Dual Ascent. Reprinted with permission from [3].

which involves the following iterations:

$$\hat{\beta}_{i[j:\hat{n}]} \leftarrow \hat{\beta}_{i[j:\hat{n}]} - 2\mathbf{q}_{ij} < \mathbf{q}_{ij}, \hat{\beta}_{i[j:\hat{n}]} > \quad j \leftarrow k \text{ to } 1$$

By reversing $j \leftarrow 1$ to k and using β as the vector in the above iteration, one can similarly compute $\hat{\beta} = Q^T \beta$. It can be seen that each iteration of updating $\hat{\beta}$ or β comprises an inner product followed by a saxpy operation.

We observe that the computations in the dual ascent stage invoke the previously described BLAS Level-1 hardware kernels, i.e., inner product and saxpy, which are reused with an appropriate memory interface. This is achieved through column-major storage of $\hat{\alpha}$, $\hat{\beta}$ and $\hat{\mathbf{e}}$.

Imposing the non-negativity constraint on β , using a ReLU function, is an element-wise oper-

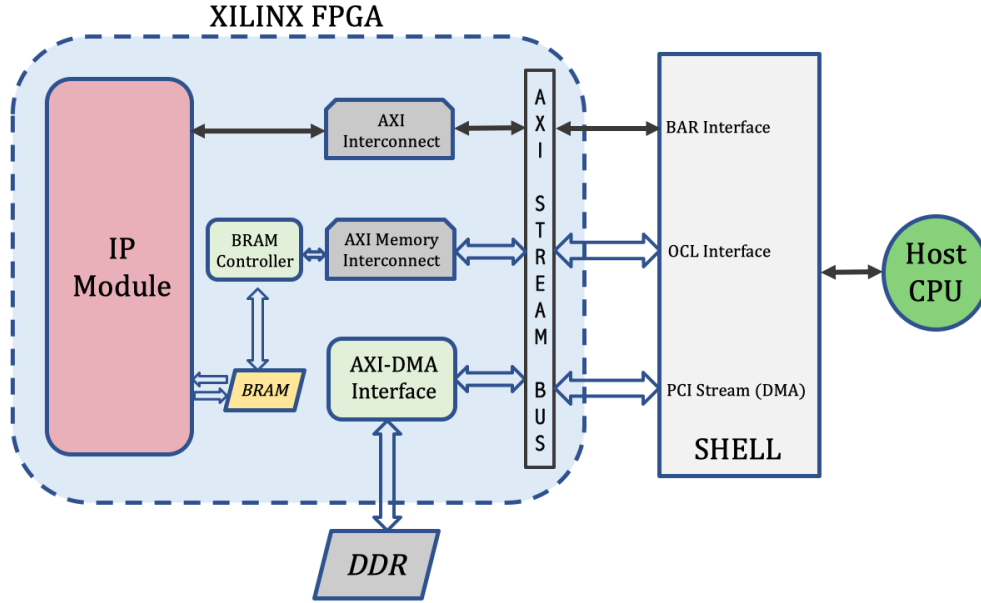


Figure 6.6: FPGA block diagram and memory interface for a single compute unit in a multiple FPGA based system. Reprinted with permission from [3].

ation. A hardware kernel for ReLU is synthesized by replacing the adder/multiplier pair in Figure 6.3(b) by a zero-comparator. Similarly, error per iteration, $\left\| \left(\hat{\beta}^{k+1} - \hat{\beta}^k \right) \right\|_1$ is computed through a binary reduction tree similar to Figure 6.3(a), with the leaf nodes configured to compute the difference of absolute values.

6.4.2 Interface Design

Upon synthesis, the hardware kernels are interfaced with the host, DDR and Block RAMs. The Xilinx Vivado design suite supports the Advanced eXtensible Interface (AXI²) protocol for Intellectual Property (IP) cores. Figure 6.6 depicts the block-diagram representation of the synthesized FPGA design for a single compute unit. In our framework, multiple such FPGA-based computing units are connected via respective hosts via PCIe bus. The synthesized hardware kernels for computation are packaged into a single IP logic core, denoted as QRSVM IP Module. The IP module is interfaced with the host over an AXI^{lite} interface. In this configuration, the host is master while

²https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf

the IP runs in servant mode. Control and status information is monitored over this interface port. Additionally, the port allows the host to read/write parameters to/from the IP (such as pointers to input data, return values, etc.). The on-chip BRAMs are synthesized with a *bram* interface. The maximum bus width supported by this interface is $N = 1024$ bits. Since we provide support for double-precision floating point numbers without any loss of accuracy when compared to software implementation, the maximum parallel compute units, $W = \lfloor \frac{N}{B} \rfloor = 16$ (refer Section 6.4.1). The IP connects to off-chip DDR over an *AXI Master* port. With this interface, the IP assumes the role of the bus controller and can directly issue memory references without host mediation.

Throughput: The throughput of each hardware kernel is determined by N (data bus width) and $W = \lfloor \frac{N}{B} \rfloor$. For the saxpy kernel (Figure 6.3(b)), doubling N would double W , which in turn would double the throughput, with data available at every clock cycle. The inner product kernel follows the same trend. However, there are limitations to the maximum bus width for a given interface. For example, the *bram* interface supports a maximum data bus width of 1024 bits, while the *AXI* interface supports up to 2048 bits. Instead, we could create multiple independent interfaces to increase memory bandwidth. In Figure 6.7, we illustrate the computation of the inner product of vectors $\langle \vec{x}, \vec{y} \rangle$ of length 16. Rather than computing with the entire vector at once, \vec{x} and \vec{y} are split equally, each of length 8 and stored as column-major order in the respective memory. This allows parallel execution of the two sub-problems, $\langle \vec{x}_1, \vec{y}_1 \rangle$ and $\langle \vec{x}_2, \vec{y}_2 \rangle$ by employing two copies of the inner product kernel, described in Figure 6.3(a). Here, each kernel copy operates on its respective sub-problem and the outputs are summed to obtain the final solution. We ensure that the read/write requests are satisfied simultaneously, i.e., no two memory references are directed to the same interface to double the memory bandwidth or throughput. In our design, we achieve it by creating two independent *AXI Master* and *bram* interfaces for accessing memory.

6.5 Experiment and Results

In this section, we first describe the hardware platform, and the various datasets used to conduct our experiments. Then we evaluate and discuss the results.

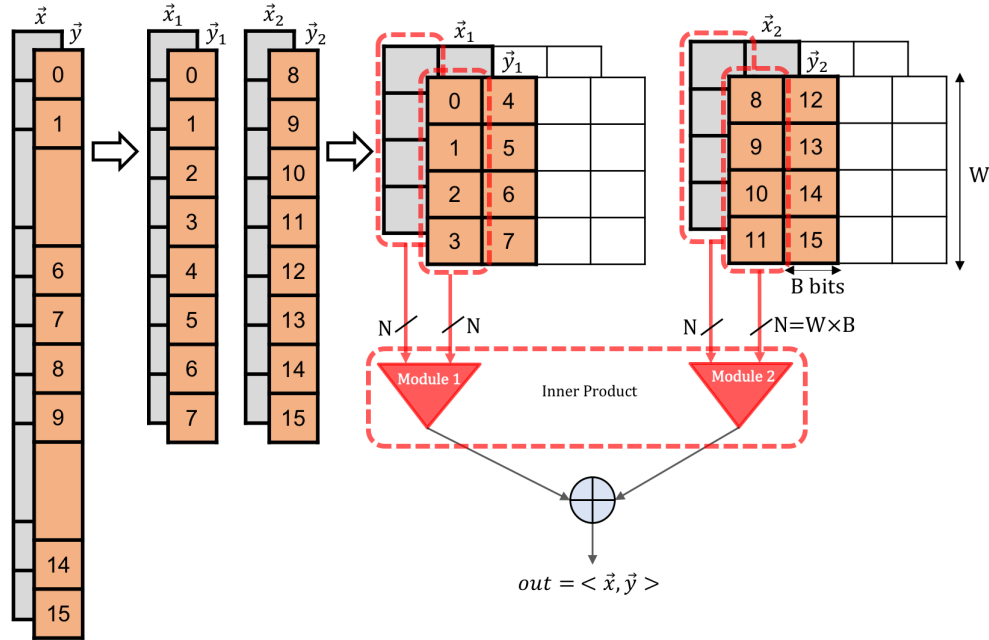


Figure 6.7: Increasing throughput of $\langle \vec{x}, \vec{y} \rangle$ by working with two copies of the inner product kernel, in parallel. Reprinted with permission from [3].

6.5.1 Hardware Description

With the ease of availability and access to multiple FPGA units, we conduct experiments on the AWS F1 platform ³ to create the proposed multiple-FPGA system for distributed SVM training. However, it must be noted that this setup is for the sake of convenience, and implements a proof of concept for the proposed design. An F1 instance *fl.16xlarge* features up to eight 16nm Xilinx Virtex UltraScale+ VU9P FPGAs which we consider as compute units ($p = 8$) as described in Section 6.3. These FPGA units communicate with each other via host processors configured over PCIe interface. It is to be recalled that the host does not share any computational workload with the corresponding FPGA. Moreover, the algorithm has communication overhead of less than 1% as discussed later. Hence, any low-power embedded microprocessor can be used as the host.

³<https://aws.amazon.com/ec2/instance-types/f1>

Benchmark	Application	#samples (n)	#features (d)	<i>k</i>-rank
MNIST	Image	60,000	780	128
Skin	Health	200,000	3	64
Webspam	Email	350,000	254	128
Covtype	Geography	464,810	54	64
SUSY	Physics	2,000,000	18	128

Table 6.1: Benchmark dataset description. Reprinted with permission from [3].

6.5.2 Experimental Setup

To evaluate the performance of our FPGA design, we use various SVM benchmark datasets from the LIBSVM binary classification repository ⁴. These are described in Table 6.1. In the initialization stage of the QRSVM algorithm, the data is preprocessed using Memory Efficient Kernel Approximation (MEKA) [5] technique to obtain the k -rank approximation of the kernel matrix. It is worth noting that the algorithmic contributions begin after the preprocessing stage. Hence, one can choose to replace MEKA in the preprocessing stage (initialization) with any competing kernel approximation technique performed on the host processor. Since the preprocessing stage using MEKA takes a small fraction of overall training time across all the benchmarks used, we exclude it from overall latency [6].

6.5.3 IP Synthesis

We synthesize the QRSVM IP as a single logic core on every FPGA unit in our system to demonstrate its capability to completely handle all the computational operations in the training by itself. Xilinx Vivado High-Level Synthesis ⁵ (HLS) tool is used for RTL synthesis. First, we create an MPI-based C++ implementation of QRSVM algorithm using the Armadillo linear algebra library [34] with LAPACK/BLAS integration. HLS enables such C++ and System C specifications to be directly targeted into Xilinx FPGAs along with the directives for loop unrolling, pipelining and creating wider memory interfaces for further optimization. The resulting FPGA image is

⁴<https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html>

⁵<https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html>

Resource	BRAM	DSP	FF	LUT
Used	1405	1221	545248	449113
Available	2160	6840	2363536	1181768
Utilization	65%	18%	23%	38%

Table 6.2: Utilization for FPGA *Xilinx Virtex xcvu9p-flgb2104-2-i*. Reprinted with permission from [3].

loaded onto all the FPGA units as the QRSVM IP. From our HLS synthesis, we estimate that the QRSVM IP can operate up to clock speeds of ~ 200 MHz. However, to ensure that the final synthesized circuit is free from setup and hold time violations, we adopt a 125 MHz clock for our proposed design. Moreover, we use AWS F1 Software Development Kit consisting of PCIe library to build the runtime environment for the host program to utilize and communicate with the FPGA. Table 6.2 lists the area utilization of FPGA post-synthesis. It can be observed that the utilization percentage of BRAM is highest among FPGA resources since it is extensively used to create full-duplex and high bandwidth memory. We use 4 MB of available BRAM to synthesize two memory blocks with a capacity of 2 MB each. These blocks are designated as cache memory for dual variables $\hat{\alpha}$ and $\hat{\beta}$ on each FPGA unit. With this memory specification, the proposed system processes a maximum of $256K$ samples on each FPGA unit.

6.5.4 Results and Discussions

Here, we discuss the performance of the proposed multiple-FPGA system for distributed training of SVM. Specifically, we measure and analyze the training time, parallel speedup, scalability, and energy consumption of our system. Then, we provide a brief quantitative comparison in training time and energy consumption with a commercial embedded processor for edge and widely available cloud-based processor.

6.5.4.1 Training Time

Table 6.3 shows the FPGA computation time involved during both the stages, denoted as T_{QR} in the distributed QR decomposition, and as T_{DA} in the iterative parallel dual ascent. Here, we

#units	#samples	MNIST ($C = 1, \gamma = 2^{-6}, \eta^* = 0.9, \tau = 181$)				
p	n	$T_{QR}(T_{local} + T_{master})$	T_{DA}	T_p^{FPGA}	<i>comp</i>	<i>comm</i>
1	60K	3.42 (3.38 + 0.04)	7.49	10.92	99.9%	0.1%
2	60K	1.76 (1.72 + 0.04)	4.07	5.84	99.8%	0.2%
4	60K	0.93 (0.87 + 0.06)	2.51	3.58	96%	4%
8	60K	0.46 (0.40 + 0.06)	2.14	2.61	99.6%	0.4%

#units	#samples	Skin ($C = 1, \gamma = 2^{-8}, \eta^* = 0.9, \tau = 54, 441$)				
p	n	$T_{QR}(T_{local} + T_{master})$	T_{DA}	T_p^{FPGA}	<i>comp</i>	<i>comm</i>
1	200K	2.80 (2.79 + 0.01)	4533	4536	99.9%	0.1%
2	200K	1.46 (1.45 + 0.01)	2226	2228	99.9%	0.1%
4	200K	0.74 (0.72 + 0.02))	1107	1108	99.9%	0.1%
8	200K	0.38 (0.36 + 0.02)	625	626	99.9%	0.1%

#units	#samples	Webspam ($C = 1, \gamma = 1, \eta^* = 0.9, \tau = 566$)				
p	n	$T_{QR}(T_{local} + T_{master})$	T_{DA}	T_p^{FPGA}	<i>comp</i>	<i>comm</i>
2	350K	9.80 (9.74 + 0.06)	66.20	76.14	99.8%	0.2%
4	350K	4.88 (4.83 + 0.05)	34.40	39.36	99.8%	0.2%
8	350K	2.60 (2.38 + 0.06)	17.92	20.59	99.7%	0.3%

#units	#samples	Covtype ($C = 1, \gamma = 2^3, \eta^* = 0.9, \tau = 1, 076$)				
p	n	$T_{QR}(T_{local} + T_{master})$	T_{DA}	T_p^{FPGA}	<i>comp</i>	<i>comm</i>
2	464,810	3.35 (3.34 + 0.01)	88.02	91.45	99.9%	0.1%
4	464,810	1.70 (1.69 + 0.01)	43.58	45.36	99.8%	0.2%
8	464,810	0.80 (0.78 + 0.02)	23.80	24.75	99.4%	0.6%

#units	#samples	SUSY ($C = 1, \gamma = 2^{-3}, \eta^* = 0.9$)				
p	n	$T_{QR}(T_{local} + T_{master})$	T_{DA}	T_p^{FPGA}	<i>comp</i>	<i>comm</i>
1	250K	14.01 (13.97 + 0.04)	94.04	108.08	99.9%	0.1%
2	500K	14.04 (13.98 + 0.06)	116.8	131.02	99.8%	0.2%
4	1M	14.07 (13.98 + 0.09)	162.1	176.18	99.9%	0.1%
8	2M	14.14 (13.98 + 0.16)	285.47	299.63	99.9%	0.1%

Table 6.3: Training time, T_p^{FPGA} (in seconds) using p QRSVM IP cores or FPGA units. T_{QR}, T_{DA} : Compute time for QR decomposition and Dual Ascent on FPGA. Percentage of T_{FPGA} spent in computation, *comp* and communication, *comm*. SUSY is used for Weak scaling analysis while rest are used to demonstrate strong scaling analysis. Reprinted with permission from [3].

also provide the execution time of each compute step in Figure 6.2 of the non-iterative stage of distributed QR decomposition, namely, T_{local} for local QR decomposition and T_{master} for master

QR decomposition. The overall training time, T_p^{FPGA} , on the multiple-FPGA system with $p = \{1, 2, 4, 8\}$ FPGA units is also reported. Each FPGA unit has a single QRSVM IP logic core. In addition, Table 6.3 reports the hyper parameters applied to train SVM models for different datasets, stopping threshold, $\epsilon = 10^{-3}$, an optimal step size η^* to ensure faster convergence [6], and total number of iterations, $t = \tau$. As discussed earlier, our proposed FPGA implementation allows datasets with sample size $n \leq 256K$ to be stored on a single FPGA unit. Hence, for larger datasets such as Webspam and Covtype exceeding the above capacity per unit, we omit the entry pertaining to $p = 1$ from the table and report the training time on $p = 2$ FPGA units and above.

It can be observed in Table 6.3 that parallel dual ascent is the more dominant computation stage than the distributed QR decomposition with $T_{DA} \gg T_{QR}$ due to iterative steps involved in the dual ascent. Moreover, it is also observed that for the QR process, T_{local} is more dominant than the T_{master} due to relatively larger data matrices handled during the local stage. In addition, it is to be noted that the percentage of time spent during communication, denoted as *comm*, across the multiple FPGA units in the network is negligible, i.e., 0.1 % to 4 % compared to that of computation, *comp*, across all the five benchmarks. Hence, we have designed the FPGA-based framework to accelerate the compute operations in both the stages owing to which execution time of the dual ascent stage is significantly lowered thereby reducing the overall SVM training time. With negligible communication overhead, the proposed FPGA-based implementation can scale to more number of FPGA units in a distributed network.

6.5.4.2 Scalability

Here, we discuss both the parallel speedup under strong scaling scenario and the weak scaling efficiency for the proposed FPGA-based implementation of the QRSVM algorithm.

Strong Scaling: First, we compute the parallel speedup by doubling the number of FPGA units (QRSVM IP cores) and comparing the training time to that of sequential implementation on a single FPGA unit. Figure 6.8 illustrates the parallel speedup under a strong scaling scenario wherein the overall problem size stays fixed but the number of FPGA units (or QRSVM IP cores) is doubled. For small dataset such as MNIST, training the SVM model on $p > 4$ FPGA units is overkill

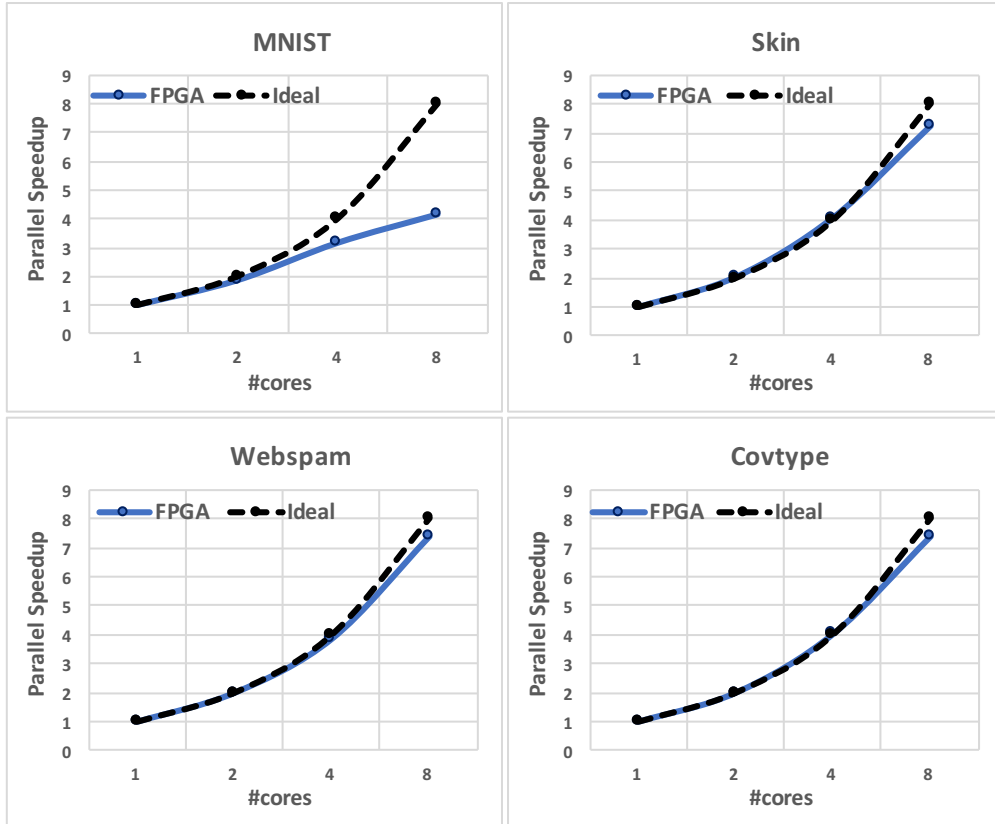


Figure 6.8: Strong scaling analysis: FPGA implementation achieves near linear parallel speedup for larger datasets such as Skin, Webspam and Covtype, as we double the #FPGA units (or QRSVM IP cores). For small dataset MNIST, going beyond $p = 4$ seems to be overkill. Reprinted with permission from [3].

as non-parallel tasks such as data loading and MEKA approximation time become relatively significant in comparison to the parallel tasks. From Table 6.3, we wish to recall that the training time for Webspam and Covtype benchmarks are computed with $p = 2$ FPGA units and above. Hence, the parallel speedup plots for these datasets take the baseline as $p = 2$. It is observed from Figure 6.8 that for relatively larger datasets, namely, Skin, Webspam, and Covtype, the parallel speedup for the proposed distributed FPGA-based design is nearly linear. In other words, training gets twice faster as the number of FPGA units (or QRSVM IP cores) doubles. Such a trend is attributed to the communication-efficient implementation of the algorithm on the FPGA.

Weak Scaling: Next, we conduct weak scaling analysis on memory-intensive benchmark such

as SUSY wherein we fix the workload per FPGA unit. Since, each FPGA unit can handle a maximum workload of $\hat{n} = 250K$ samples in its DDR memory, we randomly sample the original SUSY dataset and choose 2 million samples. Figure 6.9(a) demonstrates the weak scaling efficiency of the proposed FPGA framework as the number of FPGA units (QRSVM IP cores), p , are doubled. Weak scaling efficiency is defined as $\frac{T_1}{T_p}$, where, T_1 is training time on a single compute unit and T_p is training time on p compute units. We observe that the efficiency decreases with increasing p . This trend is expected and can be attributed to longer training time for SUSY as shown in Table 6.3, specifically arising from increase in the number of iterations until convergence. This increase in iteration count is a result of the continuous addition of a new batch of data samples to the existing training set which requires re-training and adjustment of the SVM model. Hence, T_{DA} increases for the iterative parallel dual ascent stage resulting in an increase in overall T_p^{FPGA} as shown in Table 6.3. Hence, we shift our focus to the per-iteration execution time for a deeper assessment of the weak scaling efficiency of our FPGA-based design. Figure 6.9(b) illustrates that when the SUSY sample size grows linearly and is supported with linear increase in number of FPGA units in the multiple-FPGA system, per-iteration training time remains constant. Moreover, the constant execution time of the non-iterative stage of QR decomposition, T_{QR} observed for SUSY in Table 6.3 also reinforces the desired weak scaling characteristics for fixed workload per FPGA unit. Hence, the proposed design achieves linear scaling (per iteration) under weak scaling analysis implying that it is scalable to handle growing datasets. This is attributed to the communication volume which is relatively constant, $O(k)$, regardless of the sample size (or the number of FPGA units in the system).

6.5.4.3 Energy Efficiency

As mentioned earlier in the experimental setup, we have created our multiple-FPGA system on the AWS F1 instance. Due to lack of physical access to these units, it is not possible to accurately measure the dynamic power consumption. Hence, we define the maximum energy consumed for the algorithm to be run on p FPGA units as $E_p^{FPGA} = P \times T_p^{FPGA} \times p$. Here, P is defined as the worst-case power consumption of a single FPGA rated at $P = 39$ Watts based on the post-

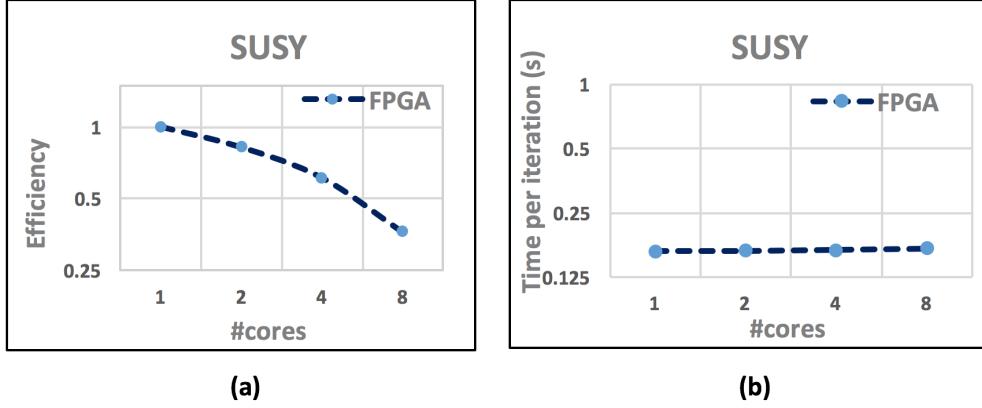


Figure 6.9: Weak scaling analysis on SUSY: (a) Weak scaling efficiency decreases as T_{QR} is constant while the dominant T_{DA} is expected to increase due to more #iterations associated with growing sample size (b) Training time per iteration is constant as desired. Reprinted with permission from [3].

synthesis of QRSVM IP under the maximum workload. Since the communication time constitutes less than 1% of the total training time as observed in Table 6.3, there is hardly any significant energy dissipation by the host processor. Table 6.4 shows the energy consumption (in kiloJoules, kJ) for the proposed FPGA-based implementation, E_p^{FPGA} , for various datasets.

Figure 6.10 illustrates the energy consumption E_p^{FPGA} trend under strong scaling scenario for representative benchmarks Skin (for $p \geq 1$) and Covtype (for $p \geq 2$). For reference, the figure also depicts the energy trend for both ideal and no scalability cases. The ideal scalability scenario refers to the serial fraction of the application being zero, hence the energy is constant with respect to the number of cores [120]. On the other hand, no scalability occurs when computation fraction that can be parallelized is zero. This is depicted by a linear trend for energy across the number of FPGA units (IP cores). It is observed that the proposed FPGA implementation consumes almost constant energy, thereby showing almost ideal behavior with respect to the number of FPGA units (IP cores). In other words, it demonstrates that the implementation is nearly fully parallel.

Figure 6.11(a) depicts the energy consumption, E_p^{FPGA} , for SUSY benchmark under weak scaling scenario. Here, the energy plots for ideal scalability and the no scalability are linear and quadratic in the number of FPGA units (IP cores), respectively [120]. We observe that the energy

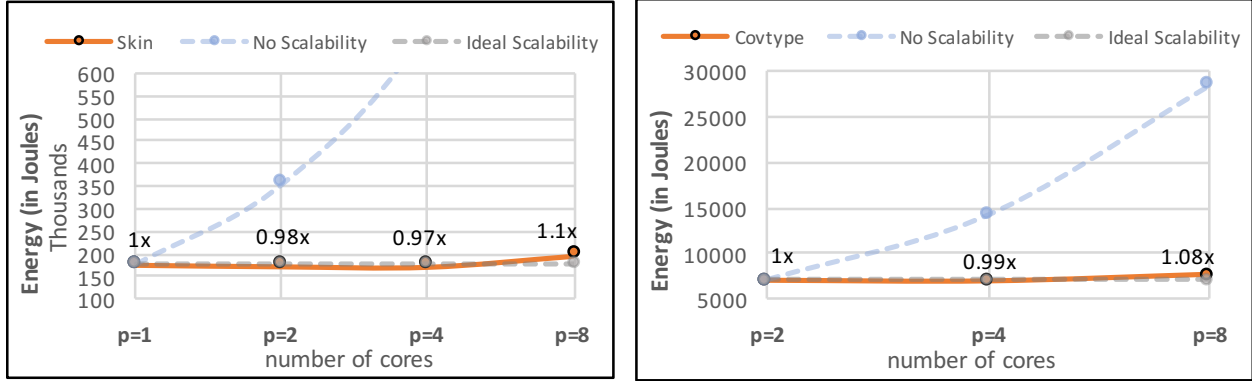


Figure 6.10: Energy consumption in FPGA under strong scaling scenario along with theoretical curves for ideal and no scalability cases. Benchmarks: Skin and Covtype. Reprinted with permission from [3].

requirement of the proposed implementation is efficient and within appreciable limits. Moreover, Figure 6.11(b) shows that the energy consumption per FPGA unit, $\left(\frac{E_p^{FPGA}}{p}\right)$, is nearly constant for the proposed implementation on SUSY benchmark, which is expected with each unit under the uniform workload. However, $p = 8$ shows a slight aberration due to relatively longer training time $T_8^{FPGA} = 299.63s$ and an increase in number of iterations associated with finer adjustment of the SVM model, as explained earlier. From the above energy analysis, the proposed FPGA-based implementation for QRSVM presents an energy-efficient platform for distributed training of SVM.

6.5.4.4 Comparison

We compare our FPGA-based hardware implementation to the C++ software implementation of QRSVM algorithm on a commercially available embedded CPU platform for edge, and a data center grade cloud processor. These two distinct hardware platforms have different architecture (FPGA vs CPU), configuration, and memory bandwidth. Hence, a comparison of these platforms is not absolute and is merely done to demonstrate the feasibility of our proposed software-hardware codesigned approach to handle computationally challenging tasks such as training of SVM. The embedded platform is an HPE ProLiant m800 cartridge ⁶ comprising of four TI Key-

⁶https://support.hpe.com/hpsc/doc/public/display?docId=emr_na-c04500667

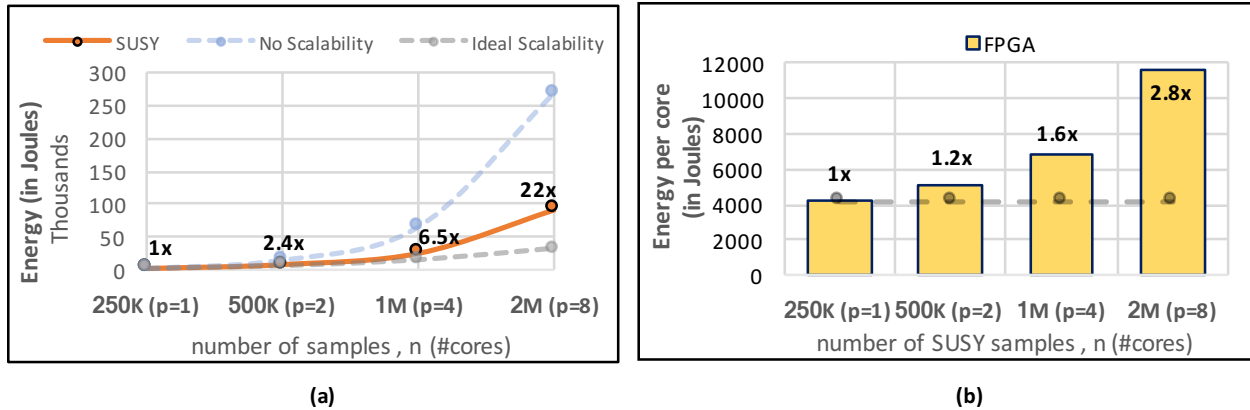


Figure 6.11: (a) Energy consumption in FPGA under weak scaling scenario along with theoretical curves for ideal and no scalability cases. (b) Energy consumption per FPGA unit while maintaining equal workload on each FPGA unit. Benchmark: SUSY. Reprinted with permission from [3].

Stone II 66AK2H SoCs running at 1 GHz connected over PCIe2. Each SoC unit features a quad ARM Cortex-A15 processors resulting in a total count of 16 processors in the cartridge. For the cloud-based processor platform, we use the Intel Xeon E5-2686 v4 (Broadwell) high-performance processors operating at 2.3 GHz available on Amazon F1 instance. For a comparative study, we run the C++ software implementation of the QRSVM algorithm for all the five SVM datasets on $p = \{1, 2, 4\}$ ARM Cortex-A15 processors (edge platform) and Broadwell processors (cloud platform) separately. In the proposed FPGA-based system, p , corresponds to the number of FPGA units, where, each unit houses a single QRSVM IP logic synthesized at 125 MHz with a post-synthesis power rating of $P = 39W$. The embedded CPU platform is low-power and measured at 14 Watts for each processor p . The thermal design power of each Broadwell processor p is rated at $P = 145W$. Table 6.4 reports the comparative training time and energy consumption results. Across the various benchmarks and number of computing units, it can be observed that in comparison to pure software implementation, the SVM training on proposed FPGA-algorithm codesign implementation is around 3x to 24x faster than the embedded edge processor (ARM), and around 1.7x faster than the cloud processor (Broadwell). For small sized datasets such as MNIST and Skin, the energy consumption for both the FPGA and ARM are almost similar. This

is because the benefit in training time from FPGA implementation gets balanced out with relatively lower power consumption on the ARM platform (14 Watts) for smaller datasets compared to FPGA (39W). However, for large datasets such as Webspam, Covtype, and SUSY, the FPGA implementation achieves 2x to 8x lower energy consumption compared to the ARM processor, and around 6.5x lower than the Broadwell processor. This is due to relatively much longer training time spent on ARM compared to the FPGA. Thus, it can be concluded that for growing data sizes, the proposed FPGA-algorithm codesign is a suitable candidate for scalable hardware solution for enabling distributed training of SVM in a multiple-FPGA system.

6.6 Summary

In this chapter, we propose a first-of-its-kind multiple-FPGA system in a distributed computing framework with a hardware-software co-design approach to tackle the more challenging training phase of machine learning models such as SVM. In particular, we synthesize algorithm IP core by mapping hardware kernels on each FPGA to accelerate the expensive training computations. The synthesized logic core operates at 125 MHz with a power rating of 39W. Here, each FPGA unit also has a dedicated low-power host which communicates with other compute units but does not share any computational workload with the FPGA. The proposed design is linearly scalable with the number of FPGA units and can accommodate growing data sizes. For various benchmarks used, it delivers faster training and is more energy-efficient than both the commercial embedded CPU (ARM Cortex-A15 processor) platform for the edge and widely available cloud processor (Broadwell) in datacenters. Thus, it can be concluded that the proposed FPGA-based design is a worthy hardware platform for accelerating distributed training for other similar machine learning models with quadratic optimization formulation. The final chapter will explore incremental learning under streaming data across distributed workers introducing fault tolerance capabilities.

#units	#samples	MNIST					
p	n	T_p^{FPGA}	T_p^{ARM}	T_p^{Broad}	E_p^{FPGA}	E_p^{ARM}	E_p^{Broad}
1	60K	10.92	31.06	18.78	0.43	0.44	2.72
2	60K	5.84	20.12	8.25	0.45	0.56	2.40
4	60K	3.58	12.9	4.35	0.56	0.72	2.52

#units	#samples	Skin					
p	n	T_p^{FPGA}	T_p^{ARM}	T_p^{Broad}	E_p^{FPGA}	E_p^{ARM}	E_p^{Broad}
1	200K	4536	21773	7167	177	305	1039
2	200K	2228	6121	3093	174	172	897
4	200K	1108	3044	1607	173	170	932

#units	#samples	Webspam					
p	n	T_p^{FPGA}	T_p^{ARM}	T_p^{Broad}	E_p^{FPGA}	E_p^{ARM}	E_p^{Broad}
1	350K	-	895	236.54	-	12.5	34.30
2	350K	76.14	477	133.99	5.9	13.4	38.86
4	350K	39.36	254	65.92	6.1	14.2	38.23

#units	#samples	Covtype					
p	n	T_p^{FPGA}	T_p^{ARM}	T_p^{Broad}	E_p^{FPGA}	E_p^{ARM}	E_p^{Broad}
1	464,810	-	1079	292.63	-	15	42.43
2	464,810	91.45	520	160.08	7.1	14.5	46.42
4	464,810	45.36	251	77.19	7	14	44.77

#units	#samples	SUSY					
p	n	T_p^{FPGA}	T_p^{ARM}	T_p^{Broad}	E_p^{FPGA}	E_p^{ARM}	E_p^{Broad}
1	250K	108.08	2452	171.29	4.2	34.3	24.84
2	500K	131.02	3131	232.01	10.2	87.7	67.28
4	1M	176.18	4189	319.03	27.5	234	185

Table 6.4: Comparison with embedded edge processor (ARM Cortex A15) and cloud processor (Broadwell) platforms. In a multiple compute system, #units, p , corresponds to #FPGA units (QRSVM IP cores), #ARM processors, and #Broadwell processors. Training time (in s), T_p^{FPGA} , T_p^{ARM} , and T_p^{Broad} . Energy consumption (in kJ), E_p^{FPGA} , E_p^{ARM} , and E_p^{Broad} . Reprinted with permission from [3].

7. RAPID INCREMENTAL SOLVER FOR FEDERATED LEARNING

Federated machine learning is an emerging field to train machine learning models across multiple workers while keeping data and compute local at the distributed edge. Empowering federated learning on devices with low memory and limited compute capabilities requires efficient processing of the streaming data while reducing communication overhead. To address the above challenges, this final chapter proposes a Rapid Incremental solVER, called RIVER, and applies it to solve federated regression problems on three distinct streaming setups; data streaming at a single worker (*Stream*), data streaming across a fixed number of workers (*Tributary*), and data streaming from varying active workers set (*Basin*). The proposed modular design is designed for scalability across batch size, feature dimension, and number of workers, negligible communication overhead, fault tolerance, and robustness of the model. We perform extensive performance evaluations on simulated federated setups under various workloads and workers to validate the above characteristics.

7.1 Introduction

The rise of modern edge devices, with their data acquisition and storage capabilities, has made distributed computing more ubiquitous than ever before. Within the constraints of their limited storage and processing capabilities, devices such as mobile phones, sensor systems in antennas, autonomous cars, smart homes, and wearable technology can constantly collect data and perform simple calculations. With the growing need for real-time analytics, it will be nearly impossible for these smart devices to transmit all data collected to a centralized server for efficient processing and training of centralized machine learning models due to data privacy concerns and limits on network bandwidth. The performance of machine learning models is heavily dependent on lots of data. Hence, the continuous collection of data streams at the decentralized sources must be quickly incorporated to incrementally update the global model and improve the quality of future predictions.

7.1.1 Motivation

Federated learning is an emerging paradigm to collaboratively train a global machine learning model across multiple devices (workers) without sharing the underlying data [121, 122, 123, 124, 125, 126, 127]. It seeks to address the above concerns on data privacy. These workers collect their data samples independently and the number of samples is highly unbalanced. Each worker can have constraints related to its processing and power capability. Since the workers are relatively simple devices, they have access to a small number of data samples and are limited in their computation abilities. During model training, many workers participate in multiple rounds of collected data streams, but it is possible for some of these workers to lose connection to the network and cease to operate or encounter device failure. Occasionally, some of the workers may straggle or may have no new data stream to report. Under these uncertain dynamics, it becomes imperative to design fault-tolerant and communication-efficient solvers to learn robust models.

7.1.2 Contributions

In light of the challenges of a typical federated learning setup (discussed in next section) and to ensure continuous model adaptation based on a constantly arriving data stream, we study the problem of federated regression on streaming data. We develop a fast and accurate solver that incrementally updates the global model through one-shot communication between the workers without storing the samples from previous streams. To the best of our knowledge, this is the first work incorporating incremental model learning in a federated setup with continuous data streams. We avoid expensive computations and frequent communication involved with local model learning and parameter exchange in a typical federated setup for rapid federated incremental learning.

The streaming data becomes available gradually over time and only a limited amount can be stored on the workers due to memory constraints. With the limited computing capabilities and demands of streaming data, it is impractical to learn a personal model and later coordinate with other workers on a communication budget to incrementally update the global model. Moreover, sharing the collected data and centralizing it on a central server is not possible due to privacy

concerns. We propose to summarize the streaming data collected and stored on each worker in a federated setup. Once the summary is generated, the data can be discarded to collect the next sequence of streaming samples. In the meantime, the *local summary* is sent to a *master* appointed from the pool of participating workers where it is merged with summaries of other workers. Instead of learning a local model at each worker, a merged summary from the master is in turn later used to recursively update the *global summary*. Essentially, in every round, all participating workers coordinate with one another to incrementally update the *global summary*, which captures all data streams used in prior rounds and adapts it with the current data streams. Using the updated *global summary*, a global model parameter is learnt at the *master*. Contrary to a typical federated setup where each worker learns its local model and collaborates to form a global model at a central server, the workers in the proposed scheme are only generating *local summaries* at low computation cost while the global model is learnt on the *master*. Unlike typical federated setups, the *master* in our case is chosen amongst the resource-constrained participating workers and compute accurate model updates without the need to approximately solve the problem.

We list the main contributions of this work as follows.

- Design accurate data summaries which entails computing a *local summary* of the current batch and updating the *global summary* of all previous batches, without having to continuously store data in between model updates.
- Devise a rapid incremental solver, namely, RIVER for 3 distinct streaming setups using the above summaries; RIVER-STREAM for a single worker, RIVER-TRIBUTARY for federated setup with data streamed from fixed number of workers, and RIVER-BASIN for fault-tolerant federated setup with varying number of workers joining/dropping-off the network.
- Apply RIVER directly on common regression solvers such as ridge regression in the popular scikit-learn library to incrementally update the model for the above streaming setups.
- Provide performance evaluation of RIVER with data of varying batch size and feature dimensions on multiple workers in simulated federated setups.

- Demonstrate the proposed incremental solver can yield accurate model parameters with a constant execution time per round and thus can efficiently scale with larger data batches compared to the baseline solvers.

The key characteristics of the proposed RIVER schemes are as follows.

- C1.** Computational cost for generating local summary is *linear* in the number of samples in the current batch, whereas it is *independent* of batch size for updating the global summary.
- C2.** Memory required to store both the local and the global data summaries is *independent* of the batch size.
- C3.** Communication overhead during coordination across multiple workers in our federated schemes, RIVER-TRIBUTARY and RIVER-BASIN is *independent* of the batch size and thus is *negligible* with respect to computation time.
- C4.** Our federated schemes employ *simple* computations of data summaries *in parallel* at the worker level while calculating the updated model at the master thereby enabling *edge analytics* on devices with low computational power.
- C5.** Our federated schemes encourage data *privacy* by not sharing data across the network and requiring no storage of data once its summary is generated.
- C6.** RIVER-BASIN is a *fault-tolerant* scheme and is naturally robust to worker failure and straggling workers.
- C7.** RIVER is *modular* in design where each scheme is built on top of the other, STREAM \rightarrow TRIBUTARY \rightarrow BASIN.
- C8.** RIVER is *universal* and applicable to any quadratic programming problem where the objective function involves a Gram matrix of the input data.

7.2 Related Work

In a typical federated learning setup based on Stochastic Gradient Descent, a selected set of workers download the initial global model from a central parameter server and retrain it locally with their own data. Then, each worker sends its local model characteristics such as parameters, gradients, etc. to the central server where the models are aggregated to form the global model updates. Next, the model updates are sent back to the workers following which the next iteration of federated learning begins. The process continues until the global model converges. However, the major challenges with the above framework are as follows: (1) Each worker is solving a local optimization problem by learning a local model which requires a large number of computations and may not be suitable for workers with limited computing capabilities. (2) The global model in the central server is usually large and needs to be compressed before being sent over a low bandwidth communication network to memory-constrained workers [122, 126, 128, 129] during each iteration. (3) It requires frequent communication between workers during training on a data batch and is hence infeasible to support fast incremental learning on continuous data streams.

Incremental learning has been implemented for many traditional machine learning algorithms such as SVM [130, 131], decision trees [132], artificial neural networks [133, 134], clustering [135, 136], and dimensionality reduction [137, 138, 139]. To solve incremental dense least squares problems, QR-decomposition has been used for multicore computing based on ScaLAPACK [140] routines with different matrix blocking and collectives via *all-reduce* scheme. However, these techniques suffer from relatively higher synchronization cost than TSQR [141]. Hence, we take inspiration from TSQR to design incremental regression specific to federated requirements $\mathcal{R}1$ – $\mathcal{R}8$ in Section 7.3.2 with decentralized workers over a dynamic communication network. It is to be noted that the over-determined LMS problems can be solved directly by solving its normal equations. However, this approach is numerically unstable. Moreover, computing and maintaining the inverse of input covariance matrix, if it exists, for streaming data leads to the accumulation of numerical errors while solving the normal equations. Since QR decomposition can handle a much wider range of matrix by avoiding the condition-number-squaring effect, it is a more numerically

stable alternative [142].

For federated regression, authors in [143] seek to learn exact feature sparsity for sparse linear regression whereas our focus is to use federated setup to incrementally and accurately update the model parameters across regression problems. For linear classification and regression models, authors in [144] proposed *CoLa*, a decentralized training algorithm based on communication-efficient *CoCoA* [145] with guarantees for convergence rates. However, *CoLa* approximately solves a local optimization problem on each worker while coordinating with its neighbors via averaging the shared global estimates. In contrast, our federated regression problem setup in Section 7.3.1 is based on learning an accurate global model parameter without requiring the workers to solve local optimization in each round of streaming data. Moreover, our communication is based on simple *gather of local summaries* at the master whereas *CoLa* works with more expensive *all-reduce* to compute locally averaged shared vector on all workers. There have also been recent studies that have examined fault tolerance for machine learning applications in data center environments [146, 147], and on remote devices [125].

In machine learning, data summarizing have been used in the form of *coresets*, and *sketches* [148, 149, 150, 151]. While *coresets* are the set of data samples that are retained, *sketches* are defined as linear maps of few or all points in the original dataset. Unlike the above summary designs which solve the problem approximately compared to full data set, we use a simple technique which preserves the input covariance, and accurately solves the equivalent problem presented in Equation (7.2) for our federated setups as demonstrated in Figure 7.15-7.16.

7.3 Preliminaries

In this section, we first describe the notation. Then, we discuss the problem setup for incremental regression. Finally, we list the federated requirements, and formally define the federated regression problem to perform incremental updates using streaming data.

Let, $\mathbf{w} \in \mathbb{R}^d$ be a d -dimensional parameter. Assume there are p workers, each with $n_{ik} \gg d$ independent samples being collected as a data stream over some observation window in every round k . Here, $i \in [p]$, and $k \in [K]$, where, $[p]$ and $[k]$ are shorthand notations to denote the sets,

$\{1, 2, \dots, p\}$ and $\{1, 2, \dots, K\}$, respectively. For a never-ending stream, $K \rightarrow \infty$. Each worker $i \in [p]$ has batch size of n_{ik} samples which are stored locally as $(\mathbf{X}_i, \mathbf{y}_i)_k$ for round $k \in [K]$. Here, data matrix $\mathbf{X}_{ik} \in \mathbb{R}^{n_{ik} \times d}$ comprises rows of d -dimensional predictor variables, and $\mathbf{y}_{ik} \in \mathbb{R}^{n_{ik}}$ is the corresponding local response vector. At each round $k \in [K]$, total number of data samples participating to incrementally update the global model parameter is $N_k = \sum_{i=1}^p n_{ik}$.

7.3.1 Problem Setup

We assume that the data distribution across p workers is identical as our goal is to incrementally update the global model parameter \mathbf{w} shared by all workers. For this study, we do not deal with non independent and identically distributed (non-i.i.d.) data which is generally associated with personalized models [152] and concept drift [153]. For load balancing across homogeneous workers, we use equal batch size $n_{ik} = n$ at every round $k \in [K]$ for each worker $i \in [p]$. This implies $N_k = \sum_{i=1}^p n_i = np$. However, it is always possible to work with uneven batch size across the workers. If $n = 0$ at any round, it can be assumed that worker $i \in [p]$ is either offline or does not have any new data available to participate in that round of incremental update. This understanding will be later used in RIVER-BASIN scheme to model dynamic networks. For our problem setup, at each round $k \in [K]$, we assume the local data matrix $\mathbf{X}_{ik} \in \mathbb{R}^{n \times d}$ at each worker $i \in [p]$ is full column-rank. The local data matrix with local response vector $\mathbf{y}_{ik} \in \mathbb{R}^{n_k}$ amount to global data matrix $\mathbf{X}_k \in \mathbb{R}^{N_k \times d}$ and global response vector $\mathbf{y}_k \in \mathbb{R}^{N_k}$.

We define the incremental regression problem as performing Least-Mean-Squares (LMS) optimization in each round $k \in [K]$. Here, the objective is to minimize the sum of squared loss between the observed predictions and the true response with the data samples collected from round $k = 1$ onwards, $(\mathbf{X}_{1:k}, \mathbf{y}_{1:k})$. The output is a sequence of global model parameters $\mathbf{w}_1 \rightarrow \mathbf{w}_2 \rightarrow \dots \rightarrow \mathbf{w}_K$ computed in each round $k \in [K]$.

$$\mathbf{w}_k = \arg \min_{\mathbf{w} \in \mathbb{R}^d} f(\|\mathbf{X}_{1:k} \mathbf{w} - \mathbf{y}_{1:k}\|_2) + g(\mathbf{w}), \quad (7.1)$$

where, $\mathbf{X}_{1:k} = \text{vstack}(\mathbf{X}_1, \dots, \mathbf{X}_k)$, and $\mathbf{y}_{1:k} = \text{vstack}(\mathbf{y}_1, \dots, \mathbf{y}_k)$ denote the vertical con-

catenation of the data samples until round $k \in [K]$. Equation (7.1) represents a family of LMS regression problems. For linear regression, $f(z) = \frac{z^2}{\sum_1^k N_k}$, and $g(\mathbf{w}) = 0$ whereas for ridge regression $f(z)$ remains the same but $g(\mathbf{w}) = \lambda \|\mathbf{w}\|_2^2$, where, $\lambda > 0$ is ℓ_2 -regularization hyperparameter. On using $g(\mathbf{w}) = \lambda \|\mathbf{w}\|_1$ above, we can then model Equation (7.1) as a LASSO where ℓ_1 -regularization induces sparsity for feature selection. Finally, by combining both ℓ_1 - and ℓ_2 -regularizations, Equation (7.1) converts to elastic-net regression.

7.3.2 Problem Statement

Designing a federated regression solver to incrementally update the global model parameter over successive rounds of data streams must meet the following requirements.

- $\mathcal{R}1$.** Data collected at each worker is naturally decentralized, hence, it should never leave its device of origin, i.e. never be saved on a centralized server nor shared among peers.
- $\mathcal{R}2$.** Each worker has low memory, hence, data samples collected locally at each round can not be stored between successive model updates.
- $\mathcal{R}3$.** At any round, the model can not be retrained from scratch by re-using data samples from all the previous rounds.
- $\mathcal{R}4$.** Each worker has limited compute capabilities, hence, local calculations must be simple and dependent on its current batch size only.
- $\mathcal{R}5$.** Workers must coordinate to perform distributed computations and solve the global model accurately in each round.
- $\mathcal{R}6$.** Communication bandwidth may be limited, hence, the transmission volume per worker must be small with one-shot communication with the master in each round.
- $\mathcal{R}7$.** Solver should be robust and fault-tolerant to straggling and offline workers.
- $\mathcal{R}8$.** Each worker must get access to the most recent global model parameter at end of each round.

It is evident that any attempt to store $(\mathbf{X}_{1:k}, \mathbf{y}_{1:k})$ for solving Equation (7.1) in any round $k \in [K]$ will directly violate the requirements $\mathcal{R}2 - \mathcal{R}3$. Considering all the above federated requirements, we are interested in answering the following question:

Problem Statement. *Given each worker $i \in [p]$ contains $n \gg d$ data samples generated as i.i.d. in each round $k \in [K]$, is it possible to efficiently and accurately update the global model parameter $\mathbf{w} \in \mathbb{R}^d$ during each round via one-shot communication?*

The efficiency of incremental model update means that the proposed solver should be *fast* and *scalable* by demonstrating a *constant* execution time over successive rounds.

7.4 Rapid Incremental Solver

We design a rapid incremental solver, RIVER for three streaming setups satisfying all the above listed federated requirements ($\mathcal{R}1 - \mathcal{R}8$) as well as answer the problem introduced in Section 7.3.2 affirmatively. We now introduce the three streaming setups.

1. **Stream.** Here data is being generated (or collected from sensor) at each round on a single worker and the objective is to incrementally update the model parameter across the data *streams*. Figure 7.1(a) illustrates a typical *Stream* setup with $p = 1$ worker and $K = 3$ rounds. We name the solver for this setup as RIVER-STREAM.
2. **Tributary.** This is a federated setup with same set of workers participating in every round and each of them is generating (or collecting) its data stream. The objective is to incrementally update the global model as the data is fed from these *tributaries* (workers). Figure 7.1(b) illustrates a typical *Tributary* setup with $p = 2$ workers and $K = 3$ rounds. We name the solver for this setup as RIVER-TRIBUTARY.
3. **Basin.** This is a federated setup with different set of workers participating in any round with their own data batch. Basin models situations in which a worker(s) may be inactive because of being offline/faulty or may be straggling, or simply not collecting new data. Also, it encompasses situations where new worker(s) join the network. Hence, the objective is to incrementally update the global model based on data being drained into a *basin* formed by

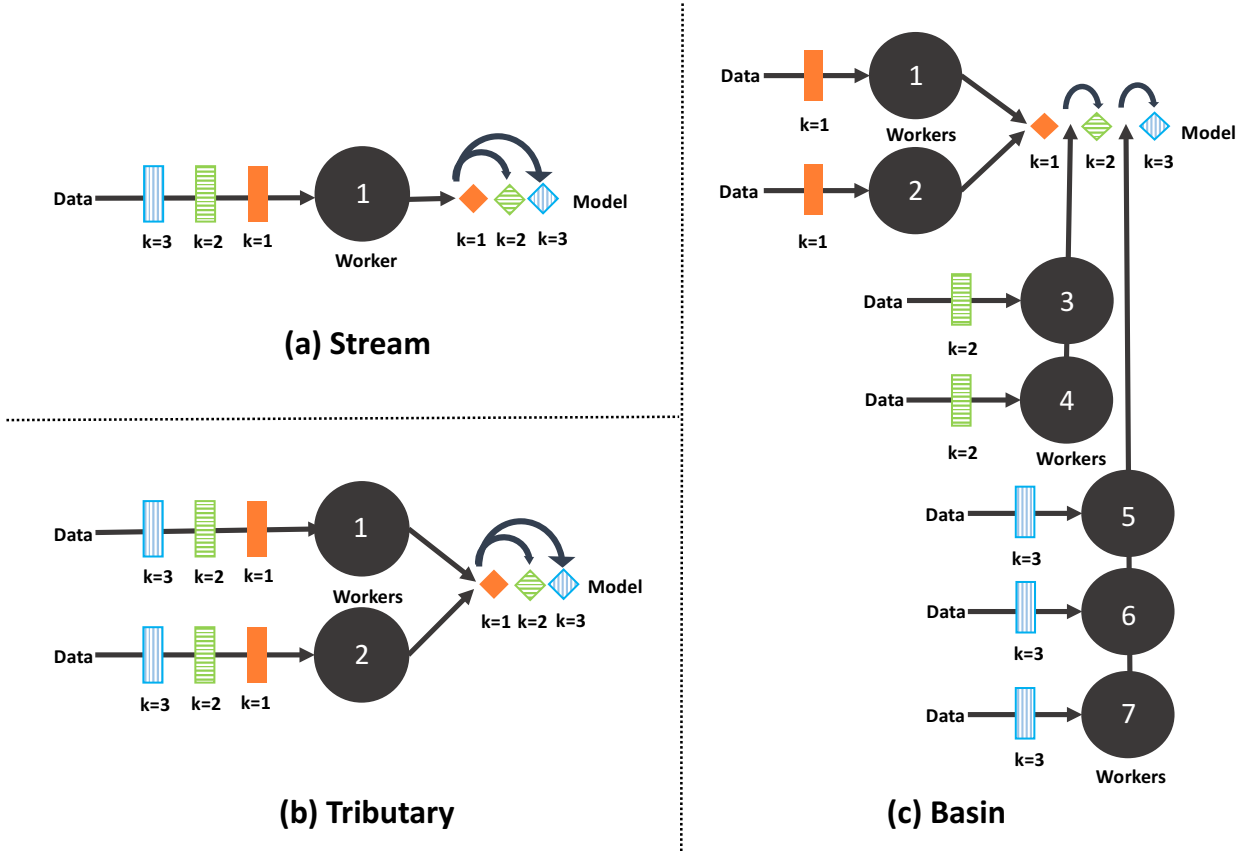


Figure 7.1: Various streaming setups, round $k \in [3]$ (a) Stream: only one worker, where data is generated in every round (b) Tributary: fixed number of workers, where data is generated by each worker in every round (c) Basin: dynamic number of workers, where in each round different groups of workers participate in the network with their data

only active set of workers for that round. Figure 7.1(c) illustrates a typical *Basin* setup with $p = \{2, 2, 3\}$ active workers over $K = 3$ rounds. Here, we can observe that new workers $i = \{3, 4\}$ participate in round $k = 2$ whereas original workers $i = \{1, 2\}$ drop from the network. In round $k = 3$, we have worker $i = 1$ rejoin, while worker $i = 3$ drops and a new worker $i = 5$ joins the network. *Basin* can be trivially extended to model situations where for any given round, the active workers behave as tributaries by generating multiple rounds of data streams. We name the solver for this setup as RIVER-BASIN.

The main idea behind the proposed RIVER schemes is to *accurately* summarize the local data generated (or collected) in the current round for all workers in *parallel* and use these local sum-

maries to incrementally update the global summary of earlier data streams. Finally, we learn the new model parameters by applying common LMS solvers from the popular scikit-learn library on the updated global summary without explicitly retraining on all previous data samples. Next, we provide details to implement the above ideas using which we design a workflow for each of the three RIVER schemes.

7.4.1 RIVER-STREAM

Here, we lay the groundwork for implementing the above ideas around local and global summary for a single worker in a typical *Stream* setup. Then we will simply extend these techniques for the federated setups, *Tributary* and *Basin*.

Definition 7.1 (*Accurate Summary*). Given a tall data matrix, $\mathbf{A} \in \mathbb{R}^{n \times d}$ where $n \gg d$, its accurate summary is defined as a matrix $\mathbf{S} \in \mathbb{R}^{n' \times d'}$ if its first dimension is relatively smaller, i.e., $n' < n$ and if the covariance matrix is preserved, i.e., $\mathbf{S}^T \mathbf{S} = \mathbf{A}^T \mathbf{A}$.

In light of the above, our first step is to accurately construct local summary for a tall full-rank data matrix $\mathbf{X}_k \in \mathbb{R}^{n \times d}$ collected at round $k \in [K]$ of our *Stream* setup. We recall that any full-rank matrix \mathbf{X}_k with $n > d$ can be uniquely decomposed into an orthogonal matrix, $\mathbf{Q} \in \mathbb{R}^{n \times n}$ and an upper trapezoidal matrix, $\mathbf{R} \in \mathbb{R}^{n \times d}$ via QR decomposition [142] such that $\mathbf{X}_k = \mathbf{Q}\mathbf{R}$. Note that the bottom $(n - d)$ rows of \mathbf{R} are zero-rows. Hence, \mathbf{R} can be efficiently stored as $\mathbf{R} \leftarrow \mathbf{R}[0 : d, :]$ comprising of top- d rows. Also note that the input covariance matrix $\mathbf{X}_k^T \mathbf{X}_k = (\mathbf{Q}\mathbf{R})^T \mathbf{Q}\mathbf{R} = \mathbf{R}^T \mathbf{Q}^T \mathbf{Q}\mathbf{R} = \mathbf{R}^T \mathbf{R}$ is preserved since $\mathbf{Q}^T \mathbf{Q} = \mathbf{I}$, where, \mathbf{I} is the identity matrix.

For the setup phase when round $k = 1$, the first model parameter \mathbf{w}_1 is computed by solving Equation (7.1) equivalently as follows

$$\mathbf{w}_1 = \arg \min_{\mathbf{w} \in \mathbb{R}^d} f(\|\mathbf{Q}_1 \mathbf{R}_1 \mathbf{w} - \mathbf{y}_1\|_2) + g(\mathbf{w}), \quad (7.2)$$

where, $\mathbf{X}_1 = \mathbf{Q}_1 \mathbf{R}_1$. Since $\mathbf{Q}_1 \mathbf{Q}_1^T = \mathbf{Q}_1^T \mathbf{Q}_1 = \mathbf{I}$, we note that

$$\begin{aligned} \|\mathbf{X}_1 \mathbf{w} - \mathbf{y}_1\|_2 &= \|\mathbf{Q}_1 \mathbf{R}_1 \mathbf{w} - \mathbf{y}_1\|_2 = \|\mathbf{Q}_1 \mathbf{R}_1 \mathbf{w} - \mathbf{Q}_1 \mathbf{Q}_1^T \mathbf{y}_1\|_2 \\ &= \|\mathbf{Q}_1\|_2 \|\mathbf{R}_1 \mathbf{w} - \mathbf{Q}_1^T \mathbf{y}_1\|_2 = \|\mathbf{R}_1 \mathbf{w} - \mathbf{Q}_1^T \mathbf{y}_1\|_2 \end{aligned}$$

Hence, solving Equation (7.1) with data $(\mathbf{X}_1, \mathbf{y}_1)$ is equivalent to solving Equation (7.2) with $(\mathbf{R}_1, \mathbf{Q}_1^T \mathbf{y}_1)$. Here, $\mathbf{R}_1 \leftarrow \mathbf{R}_1[0 : d, :] \in \mathbb{R}^{d \times d}$. Since only the top d -rows of \mathbf{R}_1 are stored, we also drop the bottom $(n - d)$ elements of the transformed response vector and simply store $\mathbf{Q}_1^T \mathbf{y}_1 \leftarrow (\mathbf{Q}_1^T \mathbf{y}_1)[0 : d] \in \mathbb{R}^d$. Moreover, performing QR-decomposition via Householder transformation [142] avoids the explicit computation and storing of $n \times n$ matrix \mathbf{Q}_1 which becomes prohibitive for large sample size n in our setup. The above modifications help with memory and computational efficiency of the RIVER without sacrificing the model correctness in Equation (7.2). Hence, $(\mathbf{R}_1, \mathbf{Q}_1^T \mathbf{y}_1)$ is the local summary of $(\mathbf{X}_1, \mathbf{y}_1)$. We now define the *local summary* of the data $(\mathbf{X}_k, \mathbf{y}_k)$ at round $k \in [K]$.

Definition 7.2 (Local Summary). *Given a tall data matrix, $\mathbf{X}_k \in \mathbb{R}^{n \times d}$ where $n \gg d$ and $\mathbf{X}_k = \mathbf{Q}_k \mathbf{R}_k$ via Householder transformation [142]. Then, the upper triangular matrix $\mathbf{R}_k \leftarrow \mathbf{R}_k[0 : d, :] \in \mathbb{R}^{d \times d}$ is its accurate summary since its first dimension, $d \ll n$, and the covariance matrix is preserved, i.e., $\mathbf{R}^T \mathbf{R} = \mathbf{X}_k^T \mathbf{X}_k$ (as per DEFINITION 1). With top- d elements of $\mathbf{Q}_k^T \mathbf{y}_k \leftarrow (\mathbf{Q}_k^T \mathbf{y}_k)[0 : d] \in \mathbb{R}^d$, $(\mathbf{R}_k, \mathbf{Q}_k^T \mathbf{y}_k)$ is the local summary of $(\mathbf{X}_k, \mathbf{y}_k)$.*

For the streaming phase when round $k > 1$, we aim to compute the sequence of global model parameters $\rightarrow \mathbf{w}_2 \rightarrow, \dots, \rightarrow \mathbf{w}_K$ by solving Equation (7.1). Hence, we hope to find a *global summary* to capture the information from the previous data samples. We now define *global summary*.

Definition 7.3 (Global Summary). *Assume a collection of data streams over round $k \in [K]$ stored as a single concatenated set $(\mathbf{X}_{1:k}, \mathbf{y}_{1:k}) \in \left(\mathbb{R}^{\sum_1^k N_k \times d}, \mathbb{R}^{\sum_1^k N_k} \right)$. Let, $\mathbf{X}_{1:k} = \mathbf{Q} \mathbf{R}$ via Householder transformation [142]. For round $r = 1$, the local summary $(\mathbf{R}_1, \mathbf{Q}_1^T \mathbf{y}_1)$ of single data batch $(\mathbf{X}_1, \mathbf{y}_1)$ is also the global summary. Then, the local summary $(\mathbf{R}, \mathbf{Q}^T \mathbf{y}_{1:k})$ of the single concatenated set $(\mathbf{X}_{1:k}, \mathbf{y}_{1:k})$ as per DEFINITION 2 is also the global summary.*

However, the requirements $\mathcal{R}2 - \mathcal{R}3$ do not permit storing the entire history of data samples $(\mathbf{X}_{1:k}, \mathbf{y}_{1:k})$ as a single concatenated set and retraining the model from scratch. To address these issues, we use the inspiration from sequential TSQR algorithm which computes the QR decomposition of a matrix stored in a 1-D block row layout [141] via reduction scheme. Based on the above algorithm, *local summary* $(\mathbf{R}_k, \mathbf{Q}_k^T \mathbf{y}_k)$ at round $k \in [K]$ can be leveraged to incrementally update the *global summary* $(\mathbf{R}, \mathbf{Q}^T \mathbf{y}_{1:k})$.

Illustration. We will show to leverage the *local summary* computed in round $k = 2$ to update the *global summary* of samples. Note that the *global summary* $(\mathbf{R}, \mathbf{Q}^T \mathbf{y}_{1:k}) \leftarrow (\mathbf{R}_1, \mathbf{Q}_1^T \mathbf{y}_1)$ at the completion of round $k = 1$. For the new data stream $(\mathbf{X}_2, \mathbf{y}_2)$ collected in round $k = 2$, its *local summary* is $(\mathbf{R}_2, \mathbf{Q}_2^T \mathbf{y}_2)$. We create

$$\mathbf{R}_{stack} = \text{vstack}(\mathbf{R}, \mathbf{R}_k) \in \mathbb{R}^{2d \times d}$$

$$\mathbf{y}_{stack} = \text{vstack}(\mathbf{Q}^T \mathbf{y}_{1:k-1}, \mathbf{Q}_k^T \mathbf{y}_k) \in \mathbb{R}^{2d}$$

by vertically concatenating the corresponding *local summary* from current round $k = 2$ with the *global summary* from previous round $(k - 1)$. Then, we compute the *local summary* of $(\mathbf{R}_{stack}, \mathbf{y}_{stack})$ which now is the updated *global summary* $(\mathbf{R}, \mathbf{Q}^T \mathbf{y}_{1:k})$. In Figure 7.2, we illustrate a generic workflow of RIVER-STREAM with a single worker and depict the recursion of *global summary* with the latest *local summary* to solve for the next model parameter. Here, it is to be noted that *the data samples, the corresponding QR-decomposition factors, and the local summary are never stored across successive rounds. Moreover, the model is never retrained from scratch using the concatenated set of all previous data samples. Rather, the scheme simply runs the LMS solver on the incrementally updated global summary to solve for the most recent model parameter, thereby satisfying the requirements $\mathcal{R}2 - \mathcal{R}3$, and $\mathcal{R}8$. All the summary calculations in Figure 7.2 are based on DEFINITION 2. We will provide the time and memory analysis in Section 7.5 and check if requirement $\mathcal{R}4$ is also satisfied.*

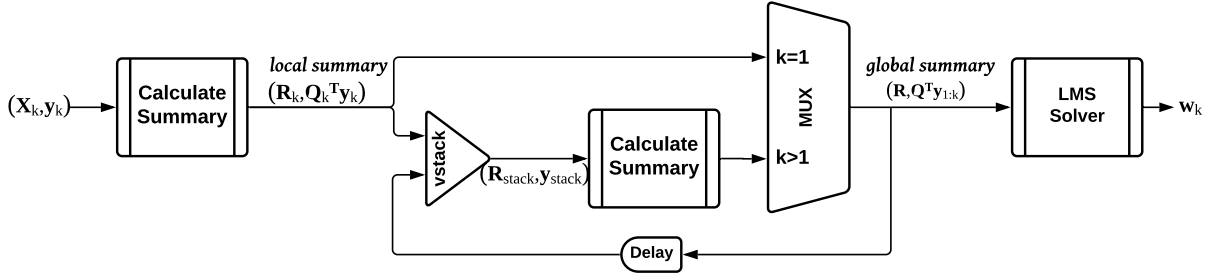


Figure 7.2: Workflow for RIVER-Stream on a single worker

7.4.2 RIVER-TRIBUTARY

Recall in our first federated setup, *Tributary*, the same set of workers (p is fixed) participate in each round of model update. Here, each worker $i \in [p]$ is collecting its own data $(\mathbf{X}_{ik}, \mathbf{y}_{ik})$ in every round $k \in [K]$ under naturally decentralized setup as discussed in Section 7.3.1. To ensure that the data is not centralized or shared with peer workers, we build upon the RIVER-STREAM scheme where for round k , we first calculate the *local summaries* $(\mathbf{R}_{ik}, \mathbf{Q}_{ik}^T \mathbf{y}_{ik})$ on all the workers $i \in [p]$ in parallel. As discussed earlier, there is no further need for any worker i to store the data $(\mathbf{X}_{ik}, \mathbf{y}_{ik})$ or their Householder QR-decomposition factors, $\mathbf{Q}_{ik} \mathbf{R}_{ik}$ once the *local summaries* have been computed for the current round k . This satisfies our federated requirement $\mathcal{R}1$. Moving on, the workers coordinate with each other in each round $k \in [K]$ over a communication network to calculate (and update) the *global summary* on a master (chosen among the p workers via MPI process) following which the global model parameter is learnt. Here, the coordination is via *one-shot communication* of the *local summaries* $(\mathbf{R}_{ik}, \mathbf{Q}_{ik}^T \mathbf{y}_{ik})$ to the master via a *gather* routine. This results in a vertical concatenation of the *local summaries*, $(\mathbf{R}_{k,stack}, \mathbf{y}_{k,stack}) \in (\mathbb{R}^{pd \times d}, \mathbb{R}^{pd})$ in round k following which the master runs the RIVER-STREAM for updating the *global summary* and accurately solving for the next global model parameter $\mathbf{w}_k \in \mathbb{R}^d$. We illustrate the above described workflow of RIVER-TRIBUTARY in Figure 7.3 with $p = 2$ workers. We can observe that the *local summaries* can be discarded to save memory once these have been *gathered* at the master. Finally, to ensure that each worker receives the latest model parameter, the master *broadcasts* \mathbf{w}_k

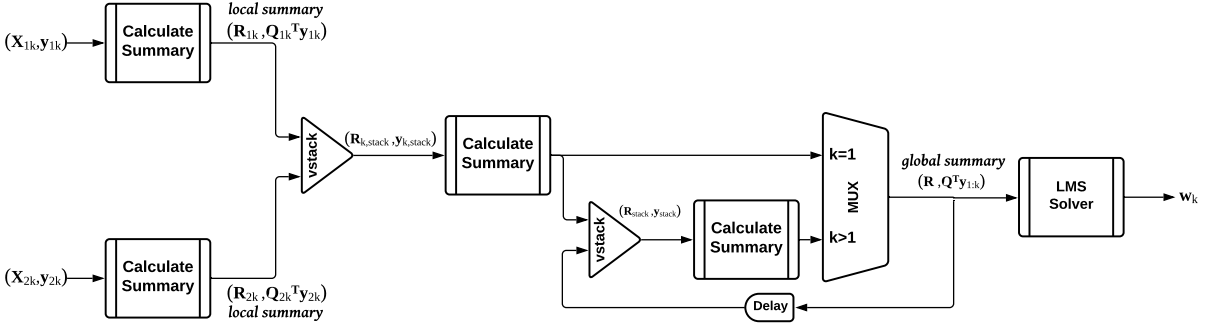


Figure 7.3: Workflow for RIVER-TRIBUTARY with $p = 2$ workers

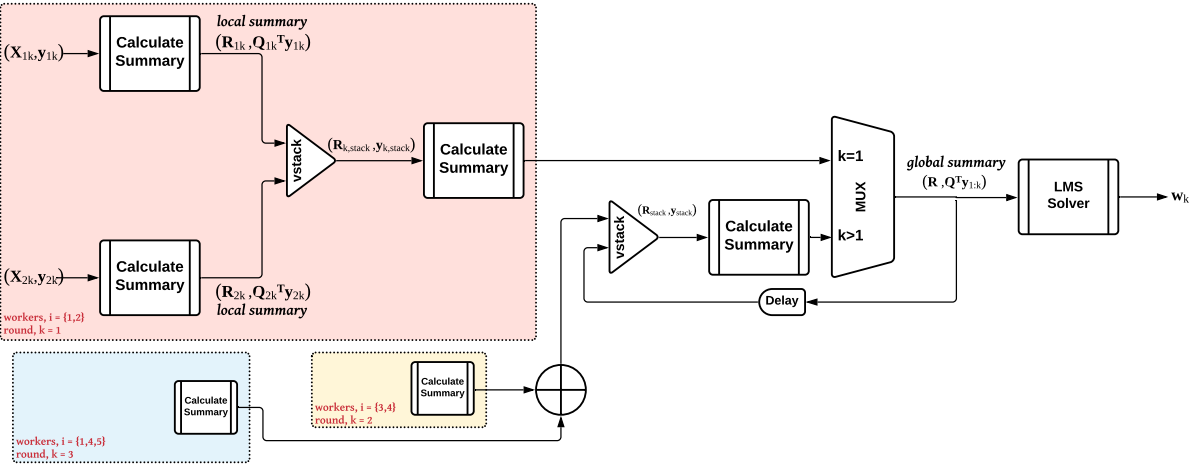


Figure 7.4: Workflow for RIVER-BASIN with number of active workers, $p = \{2, 2, 3\}$ joining in round, $k = \{1, 2, 3\}$, respectively

at end of each round $k \in [K]$. Based on above discussion, the proposed RIVER-TRIBUTARY also meets the other federated requirements $\mathcal{R}5 - \mathcal{R}6$, and $\mathcal{R}8$. Since RIVER-TRIBUTARY runs RIVER-STREAM at the master, the requirements $\mathcal{R}2 - \mathcal{R}3$ are implicitly met. We will provide the *one-shot* communication cost incurred during the *gather* process in Section 7.5 and confirm that it also satisfies the requirement $\mathcal{R}6$.

7.4.3 RIVER-BASIN

We present our second federated scheme RIVER-BASIN to solve incremental regression in the *Basin* setup. Recall that the unique characteristic of this setup is to model situations where a worker(s) may become inactive/unresponsive (offline/faulty) or may be straggling or may not have new incoming data. It also encompasses situations when a new worker(s) may join the active worker list in any round which is analogous to setting up a new device(s) in the network. The previous two schemes, RIVER-STREAM and RIVER-TRIBUTARY together satisfy the requirements $\mathcal{R}1 - \mathcal{R}6$, and $\mathcal{R}8$. RIVER-BASIN aims to fulfill the requirement $\mathcal{R}7$ and demonstrate $\mathcal{C}6$ characteristic by ensuring robustness and fault tolerance in the above dynamic network situations. We base its implementation on the RIVER-TRIBUTARY such that it meets all the requirements of the latter. The main idea behind RIVER-BASIN scheme is to probe and create a set of workers that are active (or online) and available in a round $k \in [K]$ with their data to update the model. These various sets of active workers across multiple rounds drain the *Basin* resulting in an updated global model parameter w_k . Once the active set of workers is identified, the usual RIVER-TRIBUTARY scheme follows. In other words, each round $k \in [K]$ of RIVER-BASIN is basically running RIVER-TRIBUTARY on a dedicated set of active workers for that round k . This insight is best captured in the RIVER-BASIN workflow illustrated in Figure 7.4 where a new active worker set is formed in each round $k \in [3]$ based on *Basin* setup in Figure 7.1(c). Here, for round $k = 1$, the active set comprises $p = 2$ workers, namely, $i = \{1, 2\}$. For round $k = 2$, another active set is created of same size $p = 2$ but with entirely new workers, $i = \{3, 4\}$. Finally, in round $k = 3$, the active set comprises $p = 3$ workers, $i = \{1, 4, 5\}$ where worker $i = 1$ rejoins, worker $i = 3$ becomes offline and a new worker $i = 5$ joins the network. It can be seen that RIVER-BASIN is naturally robust to worker failure and averse to straggling workers as they may join a new active set in any future round to contribute towards global model update. Since a new *master* is potentially assigned among the current active set, the updated model parameter and the same fixed-size *global summary* are *broadcasted* to “all” the workers on completion of each round to adapt to dynamic network. If a new or previously offline worker joins an active set, it could simply request for the

most recent model parameter and *global summary* from any of its recently active neighbor, in case it is assigned the next *master*. Hence, RIVER-BASIN built using RIVER-TRIBUTARY satisfies all the requirements $\mathcal{R}1 - \mathcal{R}8$ listed in Section 7.3.2.

7.5 Complexity Analysis

In this section we analyse the asymptotic complexity associated with the computation, memory requirement and communication overhead for the proposed RIVER schemes. From Figures 7.2 - 7.4, we observe the modularity in design and workflow (characteristic $\mathcal{C}7$) where RIVER-TRIBUTARY with $p = 1$ worker is basically RIVER-STREAM while RIVER-BASIN with same set of active workers in each round is RIVER-TRIBUTARY. As discussed earlier, the underlying principle in all three schemes is to first *accurately* compute the *local summary* of the data in each round $k \in [K]$ followed by recursively updating the *global summary*. Finally, run an LMS solver from popular scikit-learn library on the *global summary* to compute the next model parameter.

7.5.1 Computation Time

Here, we discuss the time spent per round in computing the *local summary* (\mathcal{T}_{local}^{cp}), updating the *global summary* ($\mathcal{T}_{global}^{cp}$), and running the LMS solver (\mathcal{T}_{LMS}^{cp}) to obtain the next model parameter.

(Local summary) As per DEFINITION 2, calculating *local summary* involves factorizing the data matrix $\mathbf{X}_k \in \mathbb{R}^{n \times d}$ in $(\mathbf{X}_k, \mathbf{y}_k)$ via Householder QR-decomposition [142], i.e., $\mathbf{X}_k = \mathbf{Q}_k \mathbf{R}_k$. The cost of this computation is $\mathcal{O}(nd^2 - d^3/3)$. The next step is to apply Householder-reflectors representing \mathbf{Q}_k and implicitly calculate $\mathbf{Q}_k^T \mathbf{y}_k$ in $\mathcal{O}(nd)$ [142]. Hence, the asymptotic cost to compute *local summary* for each worker during each round is, $\mathcal{T}_{local}^{cp} = \mathcal{O}(nd^2 - d^3/3 + nd) = \mathcal{O}(nd^2)$ for $n \gg d$ as per our problem setup. For sparse input data, Given's rotation-based QR decomposition [142] can be used to generate data summaries in the proposed RIVER schemes. Given's has lower operation count than Householder since non-zeros can be successively annihilated. For rank-deficient systems $r < d$, Rank-Revealing QR [154] can be used with an extra cost of $\mathcal{O}(d^2 r)$ which is negligible since $r < d \ll n$ in our streaming setup with continuous collection of data samples (n) while feature dimension, d is typically fixed for the problem.

(Global summary) As discussed in Section 7.4.1, we leverage *local summary* to recursively update the *global summary* for round $k > 1, k \in [K]$. Here, we perform a vertical concatenation of p *local summaries* with the previous *global summary* to effectively form $\mathbf{R}_{stack} \in \mathbb{R}^{(p+1)d \times d}$ and $\mathbf{y}_{stack} \in \mathbb{R}^{(p+1)d}$ at the *master*. Recall in RIVER-STREAM, we have a single worker, $p = 1$ acting as the *master* in each round. In our federated schemes, $p > 1$ remains fixed with same worker set in each round of RIVER-TRIBUTARY, and $p = \{p_k \geq 1\}$ with varying active worker set across multiple rounds in RIVER-BASIN. To update the *global summary*, we simply calculate *local summary* of $(\mathbf{R}_{stack}, \mathbf{y}_{stack})$ at the *master* with $\mathcal{T}_{global}^{cp} = \mathbf{O}((p+1)d \cdot d^2 - d^3/3 + (p+1)d \cdot d) = \mathbf{O}(pd^3)$.

(LMS solver) This is the last stage in the RIVER computational workflow which is dependent on the federated optimization problem. Once the *global summary* of the data is updated for the current round, Least-Mean-Squares solver is applied to *global summary* matrix of size $d \times d$ for solving federated regression. From scikit-learn library, LMS={LinearRegression, Ridge, Lasso, ElasticNet} ¹ based on the optimization problem. The computation cost for running the scikit-learn’s Ridge solver on the above *global summary* matrix is $\mathcal{T}_{LMS}^{cp} = \mathbf{O}(d \cdot d^2) = \mathbf{O}(d^3)$. Similarly, it is possible to directly apply solvers for other LMS problems mentioned above or any quadratic programming problems such as SVM classification [155]. For example, the *dual* objective function for SVM is similar to Equation (7.1) comprising a Gram matrix of the input data as follows.

$$\mathbf{w}_{dual} = \arg \min_{\mathbf{w} \in \mathbb{R}^n} \frac{1}{2} \mathbf{w}^T (\mathbf{X}\mathbf{X}^T + \rho \mathbf{I}) \mathbf{w} + g(\mathbf{w}),$$

which on reformulation via $\mathbf{X} = \mathbf{Q}\mathbf{R}$ and $\hat{\mathbf{w}} = \mathbf{Q}^T \mathbf{w}$ results in following equation similar to Equation (7.2)

$$\hat{\mathbf{w}}_{dual} = \arg \min_{\hat{\mathbf{w}} \in \mathbb{R}^n} \frac{1}{2} \hat{\mathbf{w}}^T (\mathbf{R}\mathbf{R}^T + \rho \mathbf{I}) \hat{\mathbf{w}} + g(\hat{\mathbf{w}}),$$

where, $\rho > 0$ is a penalty constant that determines trade-off between SVM margin maximization and training error minimization.

¹https://scikit-learn.org/stable/modules/linear_model.html

Since the model learning is performed only using the *global summary* at the *master*, RIVER can also be applied to run any SVM solver or any gradient descent-based solvers for any reformulated quadratic optimization problem under federated and incremental learning settings. Hence, RIVER could be universally applicable (characteristic **C8**) to problems beyond regression.

(Total) The overall computation cost for RIVER per round for each worker is $\mathcal{O}(nd^2)$, i.e. *linear* in batch size (n) to generate *local summary*. For the assigned master, the cost is an additional $\mathcal{O}(pd^3 + d^3) = \mathcal{O}(pd^3)$, i.e., *independent* of batch size to recursively update *global summary* and incrementally learn the next model parameter. This validates **C1** and **C4** characteristics while satisfying **R4** requirement.

7.5.2 Memory Consumption

Recall, the above RIVER schemes only need to store and update the *global summary* of dimensions $(d \times d, d \times 1)$ and the model parameter of size d which persist between successive rounds. That means, at the completion of each round all workers discard their data and response pair of size $\mathcal{O}(nd + n) = \mathcal{O}(nd)$, the corresponding Householder vectors of size $\mathcal{O}(nd)$ [142], and the *local summary* pair of size $\mathcal{O}(d^2 + d) = \mathcal{O}(d^2)$ to conserve memory and to avoid risk of data privacy over a period of time. In light of above, each worker will require $\mathcal{O}(d^2 + d + d) = \mathcal{O}(d^2)$, i.e., *independent* of batch size to persistently store the *global summary* and the model parameter across successive streaming rounds. This validates **C2** and **C5** characteristics. However, it is to be noted that the current design for RIVER does not yet guarantee privacy to safeguard the computations against malicious or corrupt adversary workers and is left for our future work in privacy-preserving computations to generate data summaries.

7.5.3 Communication Overhead

The federated schemes, RIVER-TRIBUTARY and RIVER-BASIN involve *one-shot* communication of *local summary* between the worker and the coordinating master via *gather* process per round as discussed earlier. Hence the communication volume per worker per round is $\frac{d(d-1)}{2}$ elements, i.e., $\mathcal{O}(d^2)$ which is *independent* of the number of samples in the current batch of streaming

data. Also, the network topology could be flexible and chosen as a *binary-reduction* tree topology than *all-to-one* star topology. For our problem setup, $d \ll n$ which makes the communication overhead negligible compared to the computation time of $\mathcal{O}(nd^2 + pd^3)$ thereby validating **C3** characteristic and satisfying requirement **R6**.

7.6 Experiment and Results

Here, we present experimental details and performance evaluation of the proposed federated schemes, RIVER-TRIBUTARY and RIVER-BASIN. Without loss of generality, we use ridge regression (RIDGECV) with cross-validation ($= 3$) as our LMS choice from popular scikit-learn library. However, RIVER is also applicable to other LMS regression problems discussed in Section 7.3.1.

7.6.1 Hardware Description

We conduct our experiments on *Ada*², a supercomputer hosted at the Texas A&M High Performance Research Computing. *Ada* is an Intel x86-64 Linux cluster with 856 compute nodes (17,436 total cores) most of which are IBM NeXtScale nx360 M4 dual socket servers based on the Intel Xeon 2.5GHz E5-2670 v2 10-core processor, commonly known as the Ivy Bridge.

7.6.2 Experimental Setup

We use the *Ada* cluster to simulate the federated setups *Tributary* and *Basin* with $p = 16$ workers³. To ensure that the simulated workers do not benefit from the shared memory and fast intra-node interconnection network, each worker was simulated on a single core from different nodes running a single MPI process. For *Basin*, each core had a 50% chance to be active for a certain round to simulate a varying active worker set. We used the Anaconda Python distribution and *mpi4py* Python package for MPI-based communication across the cores. For each experiment, the incremental learning was reported across 20 rounds of generated data streams. Linear algebra was handled by LAPACK/BLAS, through the Intel Math Kernel Library.

To experiment with multiple data sets with various batch sizes (n) and feature dimensions (d),

²<https://hprc.tamu.edu/wiki/Ada:Intro>

³based on the number of nodes assigned to an *Ada* user in a single session

we create $n \times d$ synthetic data per worker in each of the 20 rounds of incremental learning. The entries in the data were uniformly randomly generated in $[-100, 100)$ with zero-centering. For scaling studies on batch size, we created data batches of $n \times 100$ sizes, where $n \in \{500, 1000, 2500, 5000, 10000, 50000\}$ for *Tributary*, and of $n \times 10$ sizes, where $n \in \{500, 1000, 2500\}$ for *Basin*. For scaling across feature size, we created data batches of sizes $2500 \times d$, where $d \in \{5, 10, 50, 100, 500, 1000\}$ for *Tributary*, and of sizes $500 \times d$, where $d \in \{5, 10, 50, 100\}$ for *Basin*.

We compared the RIVER schemes of federated solvers against three solvers: XY-CUMULATIVE which is our standard baseline where data is communicated to a single worker and the ridge regression model is solved on the concatenated dataset, QR-CUMULATIVE which is our proof-of-concept based on creating a single summary via Householder-QR [142] on the concatenated data, and running the regression solver on the summary, and finally the classical recursive least square method, RECURSIVE-LS [156, 157] for incremental learning. RECURSIVE-LS is an online algorithm for computing the best model estimate from all the measurements it has seen up to the current time. It comprises of computationally cheaper initialization step for first batch and computationally expensive update steps which are constant for successive streaming data rounds ⁴.

7.6.3 Results and Discussions

Here we present and discuss performance of the proposed solver in terms of scalability, model accuracy, and analyze timing breakdown.

7.6.3.1 Scalability

We perform scaling studies across batch size n , across feature dimension d , and across number of workers p in Figure 7.5-7.8, Figure 7.9-7.12 and Figure 7.13-7.14, respectively to demonstrate the effectiveness of creating the *local summaries* and the *global summary* in our proposed federated RIVER schemes.

⁴<https://cpb-us-w2.wpmucdn.com/sites.gatech.edu/dist/2/436/files/2017/07/22-notes-6250-f16.pdf>

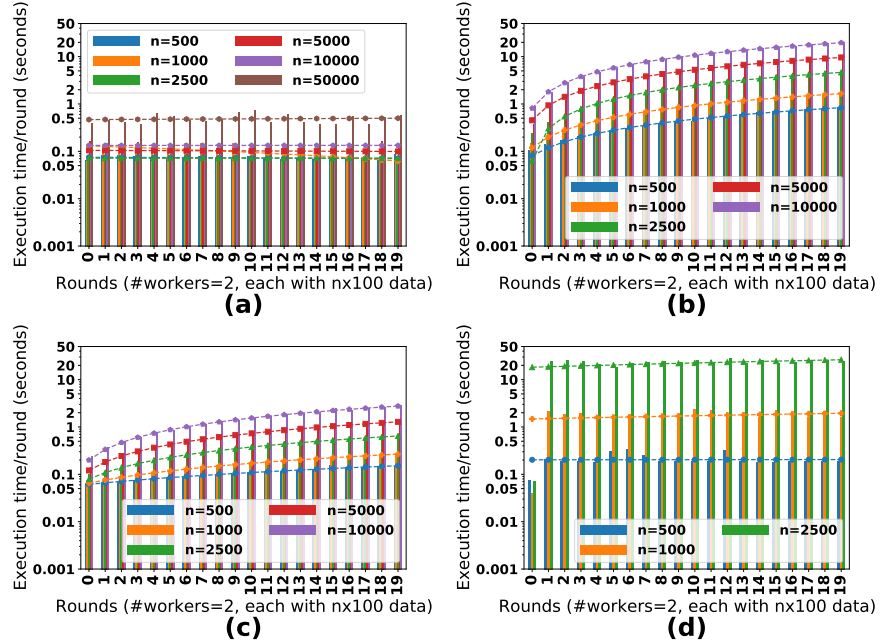


Figure 7.5: Tributary: scaling across batch size, n (a) RIVER (b) Xy-Cumulative (c) QR-Cumulative (d) Recursive-LS

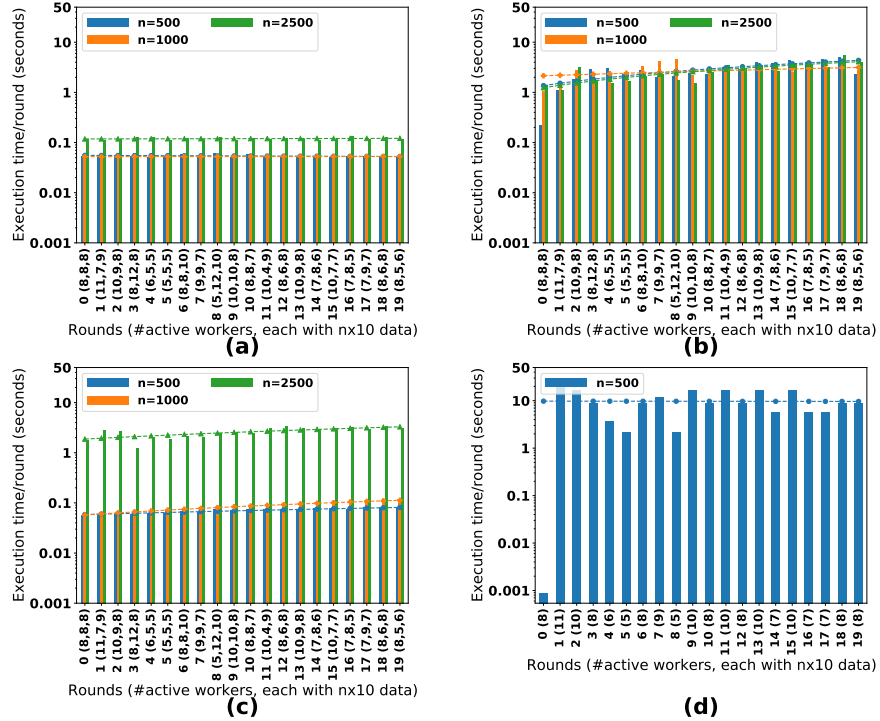


Figure 7.6: Basin: scaling across batch size, n (a) RIVER (b) Xy-Cumulative (c) QR-Cumulative (d) Recursive-LS

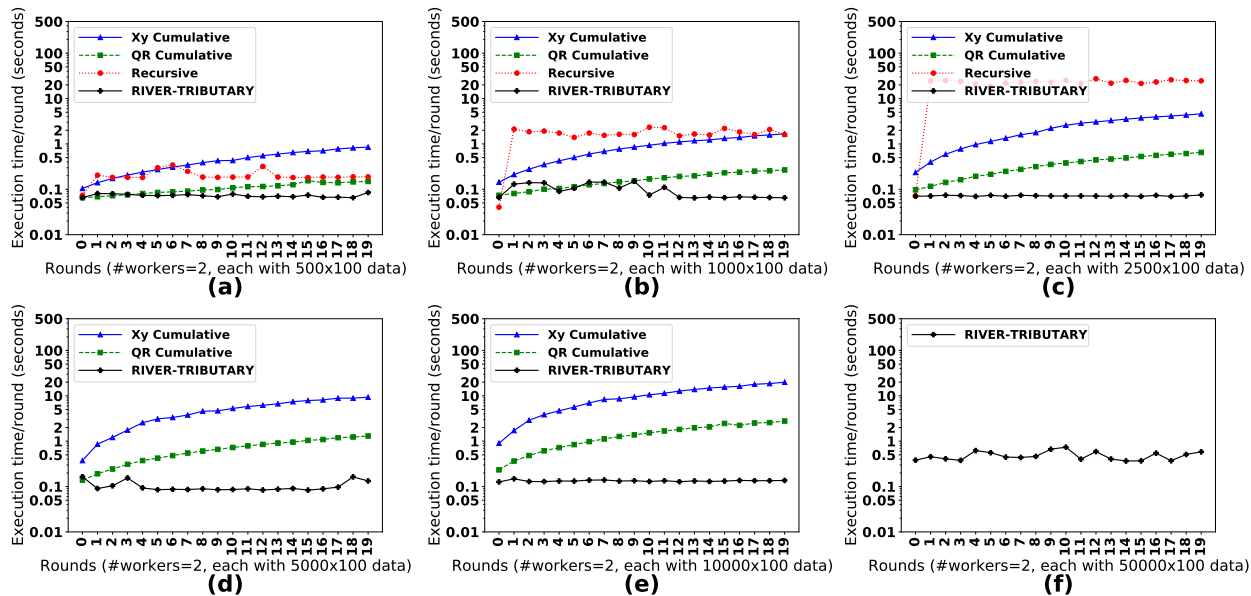


Figure 7.7: Tributary time, various n (a) 500×100 (b) 1000×100 (c) 2500×100 (d) 5000×100 (e) $10K \times 100$ (f) $50K \times 100$

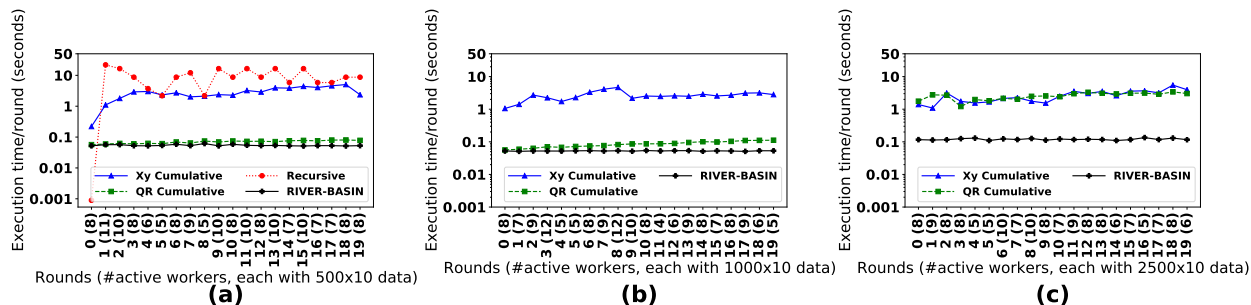


Figure 7.8: Basin time (a) 500×10 (b) 1000×10 (c) 2500×10

Scalability across Batch Size. In Figure 7.5 and 7.6, we present the scaling performance of the proposed federated solvers, RIVER-TRIBUTARY and RIVER-BASIN across various batch sizes n and compare with the baselines. In Figure 7.5(a) and 7.6(a), we observe the execution time of the proposed RIVER schemes is *constant* over the 20 streaming rounds for any batch size n . This justifies that the computation time per round is *linearly* dependent on a given batch size as discussed in Section 7.5.1. While the *Tributary* results were reported on $p = 2$ workers, the above observation for the RIVER-BASIN in Figure 7.6(a) becomes even more significant in the context of

varying size of the active worker set in each round. This could also be easily explained by *parallel* computation of *local summaries* on the workers, and the same fixed size of *global summary* to incrementally learn the next model parameter, independent of batch size as per Section 7.5.1. Such *constant* trend is also observed in RECURSIVE-LS solver in Figure 7.5(d) and 7.6(d) but is much slower compared to our proposed solvers. Figure 7.5(b)(c) and 7.6(b)(c) demonstrate poor scaling of XY-CUMULATIVE and QR-CUMULATIVE solvers across batch size on *Tributary* and *Basin* setups. Here, with each concatenated data batch over successive rounds, the execution time grows *linearly* with streaming rounds and number of workers. Hence, it is impossible to support continuous data streams indefinitely and infeasible for federated setups. From Figure 7.7-7.8, we observe that the proposed RIVER schemes consistently outperform the other solvers for various batch sizes. The separation in execution time becomes more pronounced up to 200x (in *Tributary*, Figure 7.7(e)) and 50x (in *Basin*, Figure 7.8(c)) across streaming rounds for large batch sizes. This demonstrates the efficiency of RIVER schemes in handling large amounts of data streams compared to the other solvers. For instance, RECURSIVE-LS runs out of memory when $n \geq 5K$ in *Tributary* setup shown in Figure 7.7(d)-(f), and when $n \geq 1000$ in *Basin* setup as in Figure 7.8(b)(c). Note, none of the other solvers could compete with RIVER-TRIBUTARY on $n = 50K$ samples as depicted in Figure 7.7(f).

Scalability across Feature Dimension. In Figure 7.9-7.10, we present the scaling performance of RIVER-TRIBUTARY and RIVER-BASIN across various feature dimension d , and compare with the baselines. In Figure 7.9(a) and 7.10(a), we observe the execution time of the proposed RIVER schemes is *constant* over the 20 streaming rounds for any feature dimension d , similar to our analysis across batch size earlier. In contrast, the baseline solvers scale poorly across feature dimension with increasing execution time per round. From Figure 7.11 and 7.12, we observe that the proposed RIVER schemes consistently outperform the baseline solvers for various feature dimension as well. For smaller feature dimension, $d \ll n$, RIVER is faster by 400x as demonstrated in Figure 7.11(a)-(d), and Figure 7.12(a)-(c). For relatively larger feature dimension, it is faster by around 70x on *Tributary* (Figure 7.11(e)(f)), and by 200x on *Basin* setup (Figure 7.12(d)).

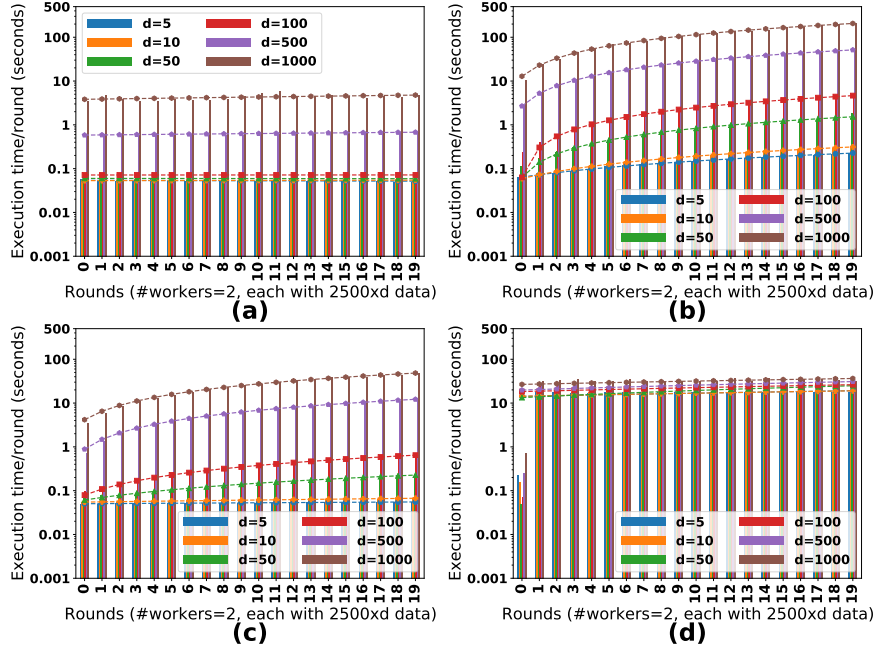


Figure 7.9: Tributary: scaling across feature, d (a) RIVER (b) Xy-Cumulative (c) QR-Cumulative (d) Recursive-LS

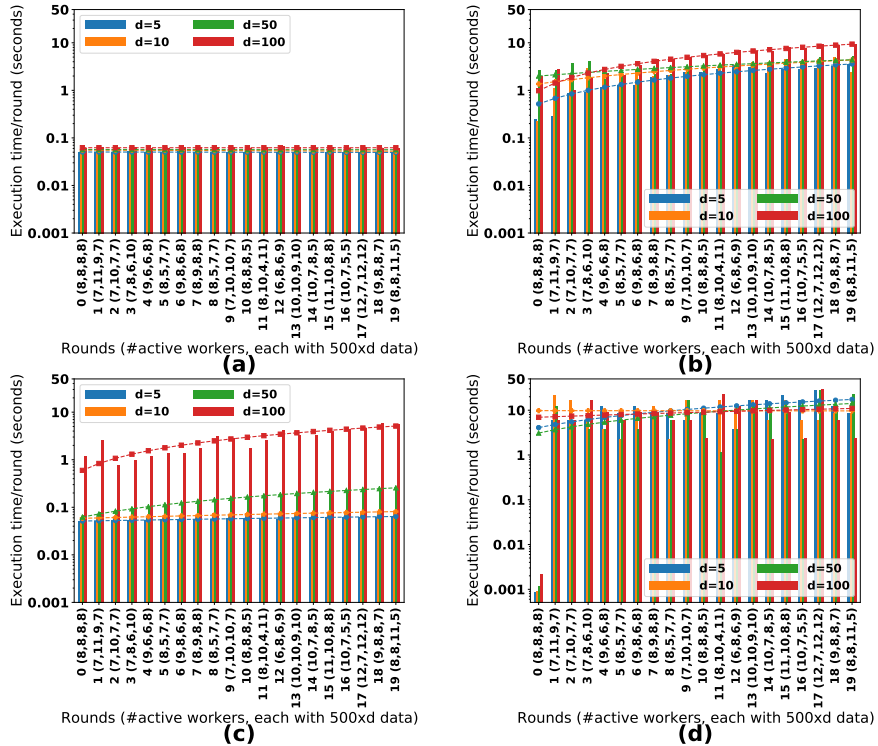


Figure 7.10: Basin: scaling across feature, d (a) RIVER (b) Xy-Cumulative (c) QR-Cumulative (d) Recursive-LS

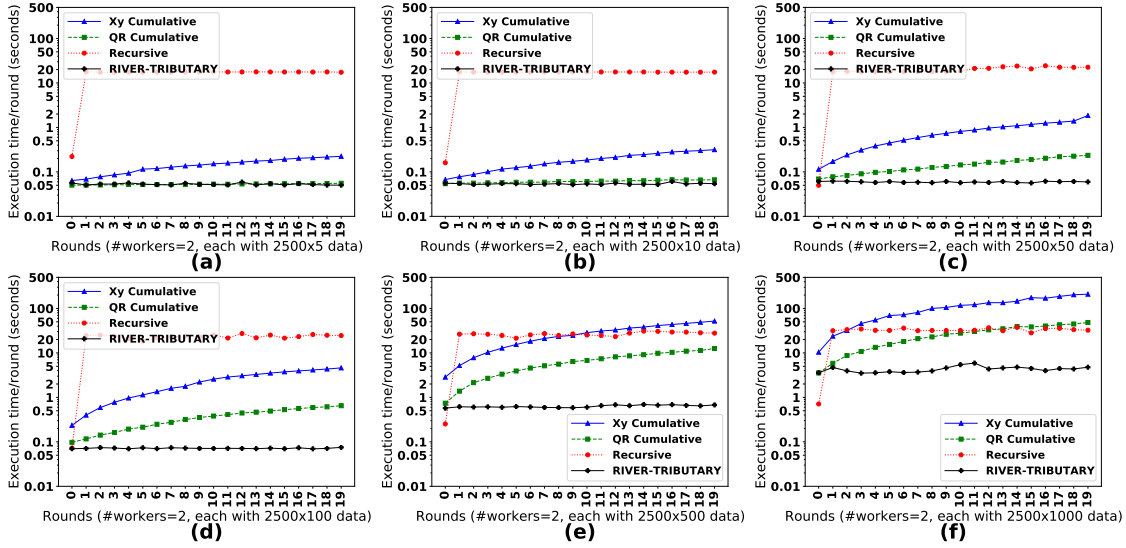


Figure 7.11: Tributary time, various d (a) 2500×5 (b) 2500×10 (c) 2500×50 (d) 2500×100 (e) 2500×500 (f) 2500×1000

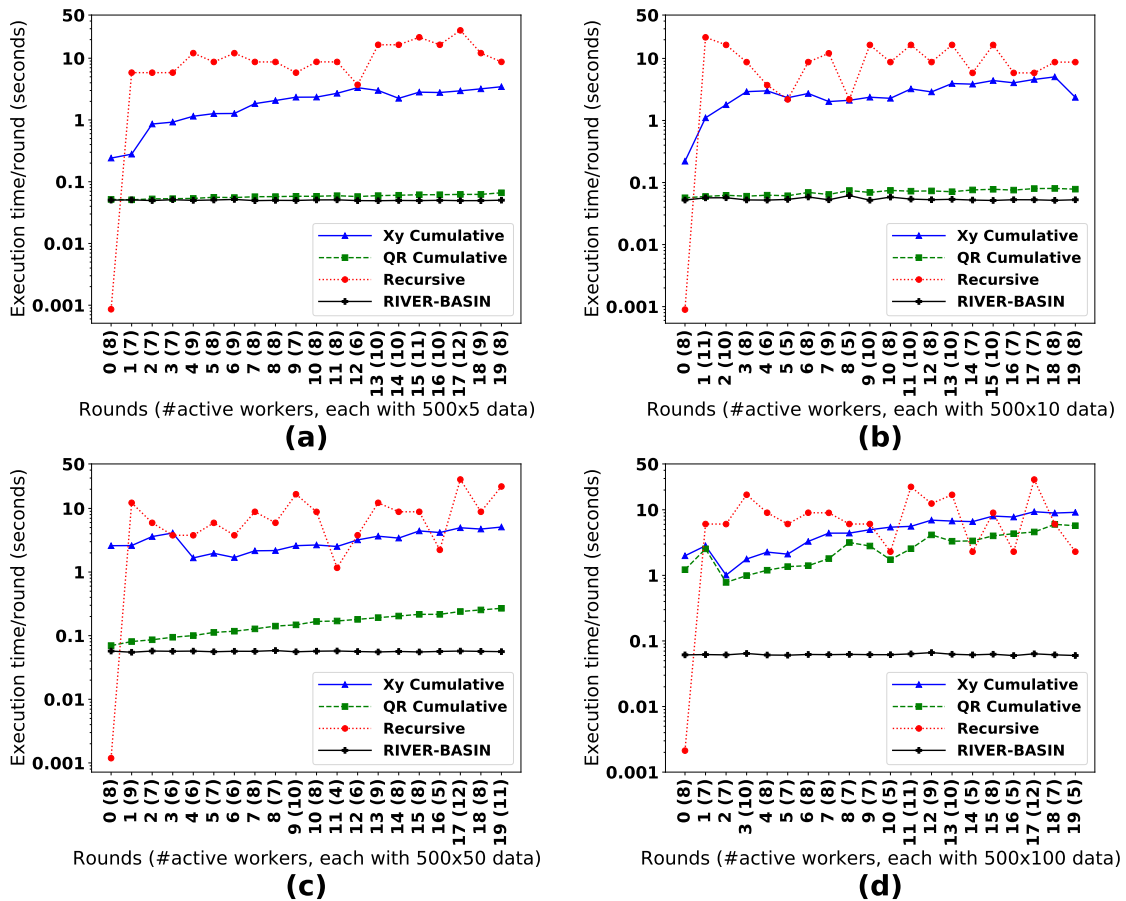


Figure 7.12: Basin time, $500 \times d$ (a) 5 (b) 10 (c) 50 (d) 100

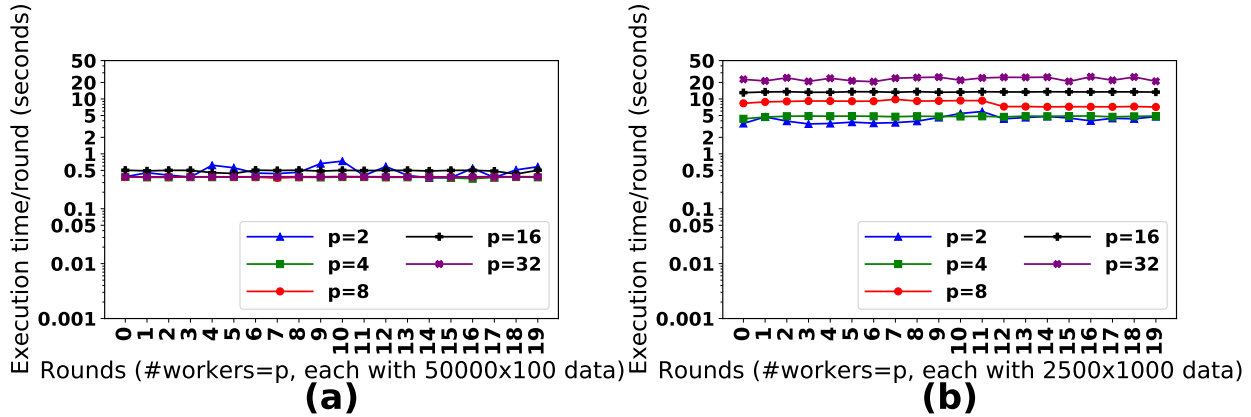


Figure 7.13: Tributary: scaling across workers p (a) RIVER, large n , $50K \times 100$ (b) RIVER, large d , 2500×1000

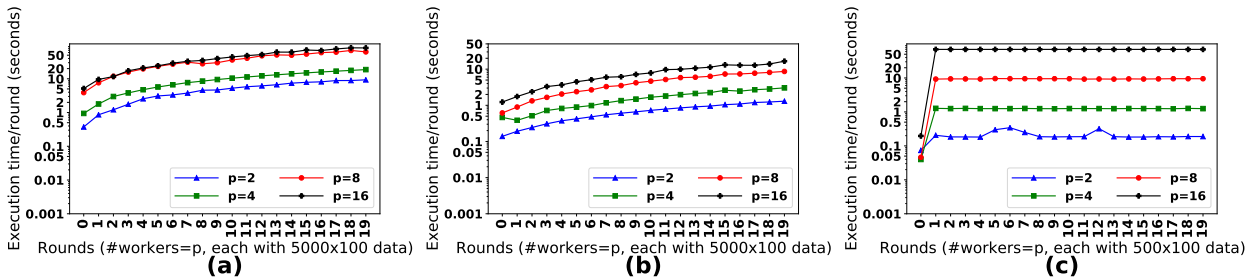


Figure 7.14: Tributary: scaling across workers p (a) Xy-Cumulative (b) QR-Cumulative (c) Recursive-LS

Scalability across Workers. We demonstrate the scaling performance for RIVER-TRIBUTARY and other baseline solvers across various number of workers (tributaries) p in Figure 7.13 and Figure 7.14 respectively. For any given solver, the workload per worker chosen is the maximum data set size amongst the various datasets discussed in Section 7.6.2 that could be supported by the solver for various p . For our problem setup with $n \gg d$, we observe linear scaling for the proposed RIVER-TRIBUTARY scheme in Figure 7.13(a) as the execution time stays constant while the workload is increased in direct proportion to the number of workers. When d is of the same order as n , we observe the scaling to be not perfectly linear with p in Figure 7.13(b) as expected with computational complexity of $\mathcal{O}(nd^2 + pd^3)$. For competing baseline solvers, namely, XY-CUMULATIVE,

QR-CUMULATIVE, and RECURSIVE-LS in Figure 7.14(a)-(c) we observe very poor scaling with increasing p and also observe relatively smaller dataset (workload) support compared to the proposed scheme. Finally, note that for *Basin* setups, we do not explicitly perform scaling analysis with p as each streaming round of *Basin*, $k \in [20]$ is a *Tributary* with various sets of active workers.

7.6.3.2 Accuracy

In Figure 7.15-7.16, we report the relative accuracy of the federated RIVER schemes by measuring the mean-squared-error between the updated model parameter and that from the baseline XY-CUMULATIVE during each round of streaming data across. We observe that the proposed RIVER schemes learn the model *accurately* in each round thereby validating *accurate* constructions of local and global summaries as discussed in Section 7.4.1. Our above results on *Basin* in Figure 7.16 also demonstrate the robustness of our proposed fault-tolerant scheme with varying active worker set. We mapped the relative accuracy of QR-CUMULATIVE on the right Y-axis to demonstrate that the proposed federated schemes follow the exact trend. In contrast, RECURSIVE-LS model asymptotically converges to our incrementally learnt model with increasing rounds of data streams.

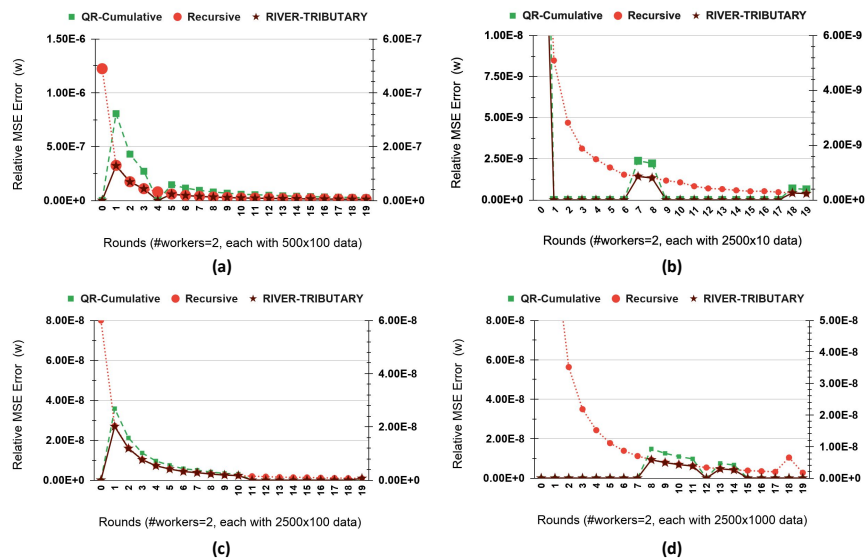


Figure 7.15: Tributary: model error relative to Xy-Cumulative (a) 500×100 (b) 2500×10 (c) 2500×100 (d) 2500×1000 . QR-Cumulative results are mapped to right Y-axis.

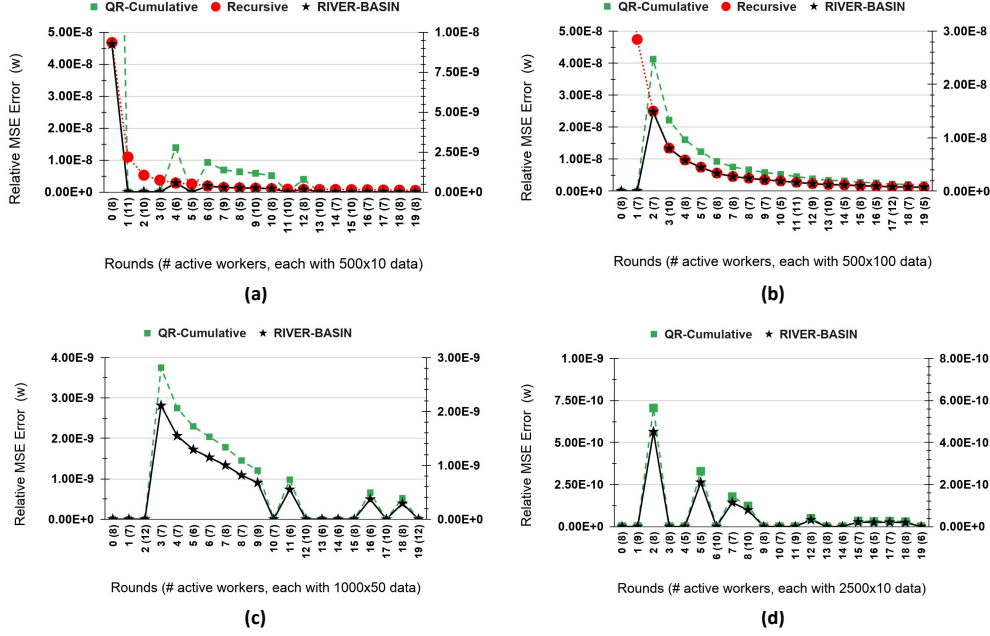


Figure 7.16: Basin: model error relative to Xy-Cumulative (a) 500×10 (b) 500×100 (c) 1000×50 (d) 2500×10 . QR-Cumulative results are mapped to right Y-axis.

7.6.3.3 Timing Breakdown Analysis

We profile the execution time of the proposed federated schemes and the other baseline solvers in Figure 7.17-7.18, and measure the time spent in each stage across 20 rounds of incremental learning. The execution time of the above solvers can be broadly separated into computation time and communication time. For our proposed schemes, from Section 7.5 recall the computation time comprises time spent calculating *local summary* (\mathcal{T}_{local}^{cp}), *global summary* ($\mathcal{T}_{global}^{cp}$), and running the *LMS* solver (\mathcal{T}_{LMS}^{cp}). The communication time in the RIVER schemes includes the time spent in *gathering* the *local summaries* from all workers and the *broadcast* of the updated model parameter at end of each round. Figures 7.17(a) and 7.18(a) demonstrate the time spent by master in the proposed schemes. Firstly, we observe that the execution time per round is fairly constant in each round of data stream compared to other solvers as discussed in our scaling studies earlier. Next, we demonstrate that the communication time is negligible in our proposed schemes. Also, we notice that the \mathcal{T}_{local}^{cp} is constant in each round as expected and is relatively more dominant that

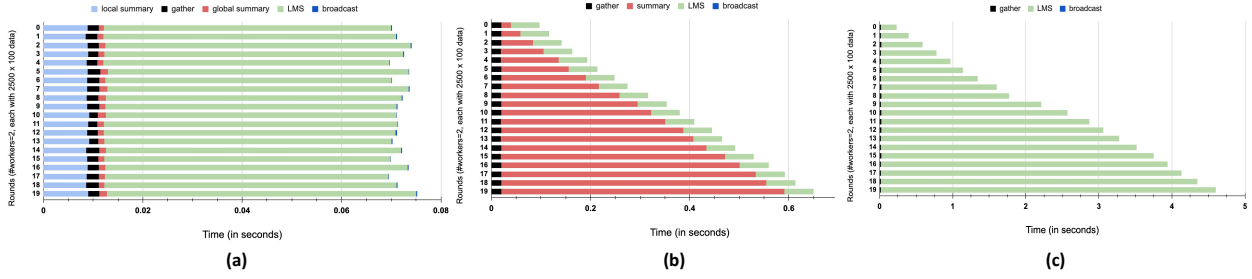


Figure 7.17: Tributary: Timing breakdown analysis (a) RIVER (b) QR-Cumulative (c) Xy-Cumulative

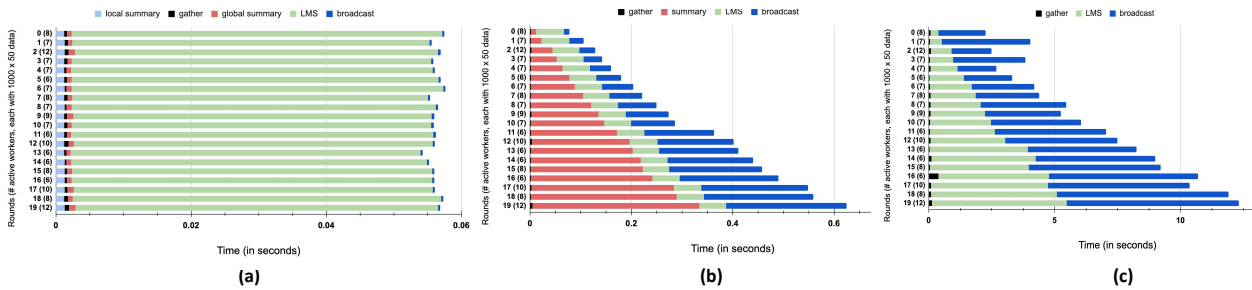


Figure 7.18: Basin: Timing breakdown analysis (a) RIVER (b) QR-Cumulative (c) Xy-Cumulative

the $\mathcal{T}_{global}^{cp}$ as per our complexity analysis in Section 7.5. Finally, we observe \mathcal{T}_{LMS}^{cp} to be the most dominant time slice where a model is learnt using the updated *global summary*. Putting the above time slices in perspective of other baseline solvers, we can understand the need and efficiency of first summarizing the data locally, rather than sending the complete data over the network to a central server. Figures 7.17(b) and 7.18(b) for QR-CUMULATIVE solvers show *summarizing* the concatenated data on the central server is a more time-consuming process than running the LMS solver. Therefore, it is best performed locally on each worker, and to not overburden the computations at the master. Figures 7.17(c) and 7.18(c) for XY-CUMULATIVE solvers demonstrate the computationally expensive training on the concatenated data and need the need to generate data summary before the training. In Figure 7.18(b)(c), we notice that the *broadcast* time in baseline solvers becomes increasingly significant to take into account fault-tolerant by introducing redundancy of also sharing the original data across the network. In contrast, RIVER-BASIN in Figure

7.18(a) demonstrates negligible broadcast time since its design is inherently fault-tolerant, and only requires the *global summary* to incrementally update the next round’s summary and model update.

7.7 Summary

Federated learning enables distributed computing among multiple workers to incrementally update the model on streaming data to alleviate the issues with data sharing. To satisfy the federated requirements $\mathcal{R}1 - \mathcal{R}8$, in this chapter we design a rapid incremental solver called RIVER. We apply RIVER to solve federated regression on three streaming setups, namely, *Stream*, *Tributary*, and *Basin*. RIVER schemes for the above setups have been designed in a modular workflow. The design is based on the idea of summarizing the local data of each worker and then pooling them together to attain a recursive snapshot of all the changes made to global model. Through analytical and empirical studies, we show that RIVER exhibits all the characteristics $\mathcal{C}1 - \mathcal{C}8$ expected in a federated solver such as ideal scalability across batch size, feature dimension, and number of workers, low memory usage, negligible communication overhead, fault tolerance, and robustness of the global model. By using RIVER on various dataset sizes, we are able to incrementally update the global model 50x to 400x faster than other baselines without any degradation in accuracy thereby demonstrating the need for efficient data summaries in federated setups. RIVER’s universal design can be applied to work on high-performance computing clusters as well as with other quadratic programming problems with similar formulation. A number of interesting future areas may be to apply RIVER in multi-party communication that incorporates different cryptography techniques, explore RIVER in context of non-i.i.d data and heterogeneous systems as workers, and design efficient accelerators for real-time edge analytics based on streaming RIVER workflows.

8. CONCLUSIONS

We conclude by summarizing the dissertation and discussing future research directions.

8.1 Conclusions

This dissertation proposes to integrate ideas from fields of machine learning, distributed computing, and hardware design for designing efficient and scalable framework for enabling distributed edge intelligence via decentralized machine learning. Conventional machine learning works on a centralized approach of collecting large amounts of data from the edge and running computationally intensive training algorithms on High Performance Computing cloud server. Once the model is trained, it is deployed on edge for inference and prediction. With proliferation of smart IoT devices and intelligence moving to the resource-constrained edge, it is paramount to design AI training solutions with requirements of distributed edge intelligence such as data privacy, real-time machine learning with low latency, less communication overhead and reduce communication frequency, scalability, energy efficiency, model robustness and adaptability to streaming data. In light of the above Edge AI requirements and challenges with conventional centralized frameworks, we systematically develop a decentralized machine learning framework to train AI models across multiple workers.

First, we propose a relaxed synchronization approach called *LSDA* for solving generic parallel quadratic programming problems with guaranteed convergence and without loss of accuracy to the original solution obtained with fully synchronization approach. Here, we also proposed an analytical solution an optimal synchronization period for workers to combine their local updates and communicate periodically. We show a 160x speedup in solution time for a large-scale quadratic programming problem. We empirically demonstrate that the relaxed synchronization technique reduces communication overhead by 99.65% in comparison to the tightly synchronization implementation. Next, we revisited the classical Householder-based QR decomposition to design and advocate for Householder sketches as effective means to directly summarize the data

at the edge to further train and accelerate least-mean-squares model globally on one of the workers. Our results with the proposed *LMS-QR* solver show Householder sketch speeds up existing LMS solvers in the scikit-learn library up to 100x-400x. Also, it is 10x-100x faster than the above strong baseline with similar numerical stability. Our results for distributed implementation show a near-negligible communication overhead with linear scalability. The summarization techniques developed here is applicable to similar convex machine learning problems. Then, we combine the above proposed relaxed synchronization and Householder sketch techniques to parallelize convex machine learning optimization problem. Specifically, we worked with kernel Support Vector Machines to devise a memory-efficient training algorithm called *QRSVM* which was trained on Householder sketches using parallel dual ascent and optimal step size. We analytically derived a relationship between optimal synchronization period and optimal step size (or learning rate) for iterative solvers. The experiments demonstrate that the proposed *QRSVM* algorithm trains kernel SVM 8x to 10x faster than competing parallel solvers on 16 workers. We found that Householder-sketch renders the original dense convex machine learning problem into sparse formulation which is completely separable for parallel training in a distributed framework. This motivated us to present a communication-efficient implementation of *distributed-QRSVM* with negligible communication overhead to scale model training on large-scale datasets and large number of workers. Experiments on benchmark data sets with up to five million samples demonstrate negligible communication overhead and linear scalability. Execution times are vast improvements over other widely used packages. Furthermore, the proposed algorithm has linear time complexity with respect to the number of samples making it ideal for training on decentralized environments such as smart embedded systems and edge-based IoT. Next, we built a first-of-its-kind multiple FPGA accelerator system codesigned for energy-efficient training of machine learning such as SVM. Each FPGA is synthesized to operate at 125 MHz with a power dissipation of 39 Watts. The proposed FPGA co-designed accelerator system is around 3x to 24x faster than the embedded edge processor (ARM) system, and around 1.7x faster than the cloud processor (Broadwell). For large datasets, the proposed system achieves 2x to 8x lower energy consumption compared to the ARM proces-

sor, and 6.5x lower than the Broadwell processor. Finally, we integrated federated learning with incremental learning using distributed Householder sketches to develop *RIVER* schemes for rapid and incremental training of machine learning model on both single and federated setups, namely, *RIVER-STREAM*, *RIVER-TRIBUTARY*, and *RIVER-BASIN*. The proposed schemes demonstrate ideal scalability across batch size, feature dimension, and number of workers, low memory usage, negligible communication overhead, fault tolerance, and robustness of the global model. On various dataset sizes, *RIVER* incrementally updates the global model 50x to 400x faster than other baselines without any degradation in accuracy thereby demonstrating the effectiveness of accurate and efficient data summaries in incremental federated machine learning setups.

8.2 Future Directions

While addressing current gaps in our understanding of the world by creating new knowledge, we believe that a dissertation should also open up new research questions and opportunities for the community to pursue. This section discusses some exciting future directions under the realm of distributed edge intelligence and computer systems.

8.2.1 Secure Multi-Party Decentralized Machine Learning

A computer's processing power offers the potential for all kinds of valuable insights to be derived from data, so valuable that some have become convinced that data is the new oil. But, in the same way, data cannot produce value unless it is flowing. Many firms do not share data because they are highly reluctant to deal with personal and confidential information. The downside of this is that parties who have mutually mistrusting attitudes are denied the insight they could have gained from jointly analyzing their data. Secure multi-party computation (MPC) was introduced for performing calculations on behalf of multiple mutually distrusting parties where each participating party provide their input, but they do not see the inputs of other parties. In theory, this can lead to a way to deriving value from data without revealing it. In light of the above, the research question that arises is *How to enable parties to perform secure machine learning over their decentralized data without violating privacy?*

The MPC protocol guarantees that neither party can see any information about the other party’s data. Cryptographic methods, on the other hand, can be computationally expensive for the companies holding the data because of the repeated encryption and decryption process. Differential privacy based distributed machine learning poses another option for acquiring a model from multiple data sources in a private manner. Several techniques, such as the aggregation of local classifiers and the exchange of gradient information, are employed to minimize empirical risk. A recent body of work aims to create such secure multi-party computation framework for training distributed regression models [46, 47, 48, 49]. However these algorithms focus on linear learning models only and leave nonlinear learning models unexplored. It is typically assumed that a trusted central server facilitates and manages distributed machine learning algorithms, while studying the privacy and security issues associated with nonlinear learning models is largely unexplored. Moreover, a semi-honest master or malicious workers can act as adversary to disrupt the learning process. The above gaps and challenges motivate us to design secure computing frameworks for multi-party decentralized training of machine learning and deep learning models. For example, authors in [158] used Blockchain [159]-based decentralized learning system to protect the system from potential Byzantine attacks. We may seek to extend our Householder-sketch work to answer *Is it possible to design privacy-preserving sketch and coresets to protect the data summary computations against malicious parties during decentralized training?*

8.2.2 Codedesign AutoML Systems for Distributed Edge Intelligence

Breakthroughs in deep learning have been achieved with the availability of big data, rise in computational power, advances in hardware acceleration, and the recent algorithmic advancements [160]. However, designing accurate artificial neural networks manually is challenging due to variety of data types, learning tasks, hyperparameter optimizations and various options of hardware platforms such as GPUs, FPGAs, ASICs, microcontrollers, etc which makes it difficult to design one globally efficient architecture. As a result, the recent research direction is to automate the the design of deep neural networks through automated machine learning (AutoML) and more specifically neural architecture search (NAS) [161]. NAS utilizes reinforcement learning [162], evolu-

tionary algorithms [163], or other approaches to automatically discover the best model, rather than manually designing one for a given learning task. However, applying NAS to real-world problems for deployment in resource-limited platforms, such as IoT, mobile, and embedded systems still poses significant challenges and is not widely practical. Some recent work has now focused on using multi-objective optimization algorithms in the NAS search strategy by taking into account hardware constraints such as execution latency, energy consumption, memory footprint, etc. This kind of NAS, called hardware-aware NAS (HW-NAS), has been growing in popularity to automatically design both the DNN model and its hardware accelerator by allowing the bottom-up design of the two parts taking into account accuracy and efficiency of the overall design [164, 165, 166]. While the hardware accelerator design space is fixed in above work, the authors in [167, 168, 169] focus on algorithm/accelerator codesign by co-searching the both design spaces simultaneously to generate FPGA-based solutions. One could also look further into silicon photonics based accelerators for deep learning [170]. It will be interesting to introduce design parameters with respect to distributed machine learning such as communication bandwidth, number of parallel workers, local and global energy consumption, synchronization period etc in the joint search space to codesign autoML systems for distributed edge intelligence. This might help find more improved design architectures than to the ones proposed in [3, 171] for multi-accelerator systems. For example, designing temperature-aware optimizer for federated machine learning to collaboratively train a DNN model across system of multiple generated accelerators. If we have a global constraint on system latency (or performance/thermal budget), *how to efficiently optimize training a DNN and communicating its parameters/gradients across multiple connected devices without violating local design constraints? What does the tradeoff between performance and energy (or area) look like for codesigned autoML systems? How to generate autoML architecture/accelerator satisfying thermal/energy constraints by synchronizing less frequently or communicating efficiently?*

8.2.3 Systems for Lifelong Multi-Agent Learning

Learning through continuous machine learning or Lifelong Learning [172] is an advanced paradigm based on hallmarks of human intelligence in which knowledge gained from previous

tasks is used to support learning in the future. The learner acquires more knowledge and becomes more proficient in learning. However, conventional machine learning is still performed in isolation, only suitable for well-defined and narrow tasks, where it learns a model given a standard training dataset with a large number of training examples. There is no attempt to retain the acquired knowledge and apply it to future learning. In contrast, humans can learn with little data or effort. We make an attempt in this direction by designing RIVER in Chapter 7 for incremental federated learning. We envision a future where physical robots would interact with humans and systems in real-life environments, autonomous vehicles would coordinate with each other to derive insights from each other's experience for understanding and navigating challenging surrounding environment, swarm of drones would be deployed in mission critical scenarios and disaster relief where latency is paramount consideration, future planetary missions with energy-efficient autonomous rovers would rely on solar energy and battery backup for intelligent multi-agent coordination and gathering new insights about our universe to help better understand its origin and possibility of life on other planets. Different smart/IoT devices might have different local constraints with respect to hardware resources and computing capabilities. Working together as a connected smart home system, for instance, the energy budget of the home might be fixed within which the smart devices need to work. Some questions one could seek to address may be: *How frequently the model on these connected device needs to be retrained on its continuously connected data?* or *How much amount of communication/synchronization is required as a trade-off for accuracy to satisfy the global energy budget?* or *How to design a customized (adaptive) optimizer depending on the customer-chosen energy budget/plan for the month?* These applications call for lifelong multi-agent learning capabilities to be incorporated in future systems. We believe without a method of accumulating and incrementally building on the knowledge learned, a system will probably never achieve true intelligence.

REFERENCES

- [1] K. Lee, R. Bhattacharya, J. Dass, V. N. S. P. Sakuru, and R. N. Mahapatra, “A relaxed synchronization approach for solving parallel quadratic programming problems with guaranteed convergence,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 182–191, May 2016.
- [2] J. Dass, V. P. Sakuru, V. Sarin, and R. N. Mahapatra, “Distributed qr decomposition framework for training support vector machines,” in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 753–763, IEEE, 2017.
- [3] J. Dass, Y. Narawane, R. N. Mahapatra, and V. Sarin, “Distributed training of support vector machine on a multiple-fpga system,” *IEEE Transactions on Computers*, vol. 69, no. 7, pp. 1015–1026, 2020.
- [4] J. Dass and R. Mahapatra, “Householder sketch for accurate and accelerated least-mean-squares solvers,” in *Proceedings of the 38th International Conference on Machine Learning*, vol. 139, pp. 2467–2477, PMLR, 2021.
- [5] S. Si, C.-J. Hsieh, and I. S. Dhillon, “Memory efficient kernel approximation,” *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 682–713, 2017.
- [6] J. Dass, V. Sarin, and R. N. Mahapatra, “Fast and communication-efficient algorithm for distributed support vector machine training,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 5, pp. 1065–1076, 2018.
- [7] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, “Deepface: Closing the gap to human-level performance in face verification,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1701–1708, 2014.
- [8] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Łukasz Kaiser, S. Gouws,

- Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” *CoRR*, vol. abs/1609.08144, 2016.
- [9] B. Shillingford, Y. Assael, M. W. Hoffman, T. Paine, C. Hughes, U. Prabhu, H. Liao, H. Sak, K. Rao, L. Bennett, *et al.*, “Large-scale visual speech recognition,” *arXiv preprint arXiv:1807.05162*, 2018.
- [10] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, “End to end learning for self-driving cars,” *arXiv preprint arXiv:1604.07316*, 2016.
- [11] S. Mehrotra, “On the implementation of a primal-dual interior point method,” *SIAM Journal on Optimization*, vol. 2, no. 4, pp. 575–601, 1992.
- [12] K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, H. Cui, and E. Y. Chang, “Parallelizing support vector machines on distributed computers,” in *Advances in Neural Information Processing Systems 20*, pp. 257–264, 2008.
- [13] E. Osuna, R. Freund, and F. Girosi, “An improved training algorithm for support vector machines,” in *Neural Networks for Signal Processing VII. Proceedings of the 1997 IEEE Signal Processing Society Workshop*, pp. 276–285, Sep. 1997.
- [14] J. Platt, “Sequential minimal optimization: A fast algorithm for training support vector machines,” 1998.
- [15] C.-C. Chang and C.-J. Lin, “Libsvm: a library for support vector machines,” *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, p. 27, 2011.
- [16] T. Joachims, “Making large-scale svm learning practical,” tech. rep., Technical report, 1998.
- [17] G. Zanghirati and L. Zanni, “A parallel solver for large quadratic programs in training support vector machines,” *Parallel computing*, vol. 29, no. 4, pp. 535–551, 2003.

- [18] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.
- [19] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, *et al.*, “Exascale computing study: Technology challenges in achieving exascale systems,” 2008.
- [20] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, “Gpu cluster for high performance computing,” in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, p. 47, IEEE Computer Society, 2004.
- [21] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, “Gpu computing,” *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.
- [22] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with cuda,” *Queue*, vol. 6, no. 2, pp. 40–53, 2008.
- [23] K. Lee, R. Bhattacharya, and V. Gupta, “A switched dynamical system framework for analysis of massively parallel asynchronous numerical algorithms,” in *American Control Conference (ACC), 2015. Proceedings of the 2015*, pp. 1095–1100, IEEE, 2015.
- [24] J. Tsitsiklis, D. Bertsekas, and M. Athans, “Distributed asynchronous deterministic and stochastic gradient optimization algorithms,” *IEEE transactions on automatic control*, vol. 31, no. 9, pp. 803–812, 1986.
- [25] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989.
- [26] A. Nedić, D. P. Bertsekas, and V. S. Borkar, “Distributed asynchronous incremental subgradient methods,” *Studies in Computational Mathematics*, vol. 8, pp. 381–407, 2001.
- [27] G. Scutari, D. P. Palomar, and S. Barbarossa, “Asynchronous iterative water-filling for gaussian frequency-selective interference channels,” *Information Theory, IEEE Transactions on*, vol. 54, no. 7, pp. 2868–2878, 2008.

- [28] E. Wei and A. Ozdaglar, “On the $o(1/k)$ convergence of asynchronous distributed alternating direction method of multipliers,” in *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*, pp. 551–554, IEEE, 2013.
- [29] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar, “An asynchronous parallel stochastic coordinate descent algorithm,” in *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2014.
- [30] R. Zhang and J. Kwok, “Asynchronous distributed admm for consensus optimization,” in *International Conference on Machine Learning*, pp. 1701–1709, 2014.
- [31] R. Nishihara, L. Lessard, B. Recht, A. Packard, and M. I. Jordan, “A general analysis of the convergence of admm,” *arXiv preprint arXiv:1502.02009*, 2015.
- [32] L. Lessard, B. Recht, and A. Packard, “Analysis and design of optimization algorithms via integral quadratic constraints,” *arXiv preprint arXiv:1408.3595*, 2014.
- [33] P. Buchholz, M. Fischer, and P. Kemper, “Distributed steady state analysis using kronecker algebra,” *Numerical Solutions of Markov Chains (NSMC’99), Prezas Universitarias de Zaragoza, Zaragoza, Spain*, pp. 76–95, 1999.
- [34] C. Sanderson, “Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments,” tech. rep., NICTA, 2010.
- [35] G. A. Seber and A. J. Lee, *Linear regression analysis*, vol. 329. John Wiley & Sons, 2012.
- [36] A. E. Hoerl and R. W. Kennard, “Ridge regression: Biased estimation for nonorthogonal problems,” *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970.
- [37] R. Tibshirani, “Regression shrinkage and selection via the lasso: a retrospective,” *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 73, no. 3, pp. 273–282, 2011.

- [38] H. Zou and T. Hastie, “Regularization and variable selection via the elastic net,” *Journal of the royal statistical society: series B (statistical methodology)*, vol. 67, no. 2, pp. 301–320, 2005.
- [39] A. C. Kidd, M. McGettrick, S. Tsim, D. L. Halligan, M. Bylesjo, and K. G. Blyth, “Survival prediction in mesothelioma using a scalable lasso regression model: instructions for use and initial performance using clinical predictors,” *BMJ open respiratory research*, vol. 5, no. 1, 2018.
- [40] G. M. D’Angelo, D. Rao, and C. C. Gu, “Combining least absolute shrinkage and selection operator (lasso) and principal-components analysis for detection of gene-gene interactions in genome-wide association studies,” in *BMC proceedings*, vol. 3, p. S62, Springer, 2009.
- [41] T. Panagiotidis, T. Stengos, and O. Vravosinos, “On the determinants of bitcoin returns: A lasso approach,” *Finance Research Letters*, vol. 27, pp. 235–240, 2018.
- [42] C. Wang, R. Pan, X. Wan, Y. Tan, L. Xu, C. S. Ho, and R. C. Ho, “Immediate psychological responses and associated factors during the initial stage of the 2019 coronavirus disease (covid-19) epidemic among the general population in china,” *International journal of environmental research and public health*, vol. 17, no. 5, p. 1729, 2020.
- [43] S. A. Lauer, K. H. Grantz, Q. Bi, F. K. Jones, Q. Zheng, H. R. Meredith, A. S. Azman, N. G. Reich, and J. Lessler, “The incubation period of coronavirus disease 2019 (covid-19) from publicly reported confirmed cases: estimation and application,” *Annals of internal medicine*, vol. 172, no. 9, pp. 577–582, 2020.
- [44] G. Pandey, P. Chaudhary, R. Gupta, and S. Pal, “Seir and regression model based covid-19 outbreak predictions in india,” *arXiv preprint arXiv:2004.00958*, 2020.
- [45] A. Maalouf, I. Jubran, and D. Feldman, “Fast and accurate least-mean-squares solvers,” in *Advances in Neural Information Processing Systems*, pp. 8305–8316, 2019.
- [46] M. Ben-Or, S. Goldwasser, and A. Wigderson, “Completeness theorems for non-cryptographic fault-tolerant distributed computation,” in *Proceedings of the Twentieth An-*

- nual ACM Symposium on Theory of Computing*, STOC '88, (New York, NY, USA), p. 1–10, Association for Computing Machinery, 1988.
- [47] A. P. Sanil, A. F. Karr, X. Lin, and J. P. Reiter, “Privacy preserving regression modelling via distributed computation,” in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 677–682, 2004.
- [48] A. Gascón, P. Schoppmann, B. Balle, M. Raykova, J. Doerner, S. Zahur, and D. Evans, “Privacy-preserving distributed linear regression on high-dimensional data,” *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 4, pp. 345–364, 2017.
- [49] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica, “Helen: Maliciously secure cooperative learning for linear models,” in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 724–738, IEEE, 2019.
- [50] J. M. Phillips, “Coresets and sketches,” *arXiv preprint arXiv:1601.00617*, 2016.
- [51] I. Jubran, A. Maalouf, and D. Feldman, “Introduction to coresets: Accurate coresets,” *arXiv preprint arXiv:1910.08707*, 2019.
- [52] P. Drineas, M. W. Mahoney, and S. Muthukrishnan, “Sampling algorithms for l2 regression and applications,” in *SODA '06*, 2006.
- [53] D. Feldman, M. Monemizadeh, C. Sohler, and D. P. Woodruff, “Coresets and sketches for high dimensional subspace approximation problems,” in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, 2010.
- [54] K. L. Clarkson and D. P. Woodruff, “Numerical linear algebra in the streaming model,” in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, (New York, NY, USA), p. 205–214, Association for Computing Machinery, 2009.
- [55] C. Carathéodory, “Über den variabilitätsbereich der koeffizienten von potenzreihen, die gegebene werte nicht annehmen,” *Mathematische Annalen*, vol. 64, no. 1, pp. 95–115, 1907.

- [56] A. S. Householder, “Unitary triangularization of a nonsymmetric matrix,” *Journal of the ACM (JACM)*, vol. 5, no. 4, pp. 339–342, 1958.
- [57] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3. JHU press, 2012.
- [58] Å. Björck, “Numerics of gram-schmidt orthogonalization,” *Linear Algebra and Its Applications*, vol. 197, pp. 297–316, 1994.
- [59] A. George and M. T. Heath, “Solution of sparse linear least squares problems using givens rotations,” *Linear Algebra and its applications*, vol. 34, pp. 69–83, 1980.
- [60] C. C. Paige and M. A. Saunders, “Lsqqr: An algorithm for sparse linear equations and sparse least squares,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 8, no. 1, pp. 43–71, 1982.
- [61] D. C.-L. Fong and M. Saunders, “Lsmr: An iterative algorithm for sparse least-squares problems,” *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2950–2971, 2011.
- [62] A. Nitanda, “Stochastic proximal gradient descent with acceleration techniques,” in *Advances in Neural Information Processing Systems*, pp. 1574–1582, 2014.
- [63] X. Meng, M. A. Saunders, and M. W. Mahoney, “Lsrn: A parallel iterative solver for strongly over-or underdetermined systems,” *SIAM Journal on Scientific Computing*, vol. 36, no. 2, pp. C95–C118, 2014.
- [64] H. Avron, P. Maymounkov, and S. Toledo, “Blendenpik: Supercharging lapack’s least-squares solver,” *SIAM Journal on Scientific Computing*, vol. 32, no. 3, pp. 1217–1236, 2010.
- [65] C. J. Burges, “A tutorial on support vector machines for pattern recognition,” *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 121–167, 1998.
- [66] C. K. Williams and M. Seeger, “Using the nyström method to speed up kernel machines,” in *Advances in neural information processing systems*, pp. 682–688, 2001.

- [67] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, “Communication-optimal parallel and sequential qr and lu factorizations,” *SIAM Journal on Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012.
- [68] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero, “Elemental: A new framework for distributed memory dense matrix computations,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 2, pp. 1–24, 2013.
- [69] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H. D. Nguyen, and E. Solomonik, “Reconstructing householder vectors from tall-skinny qr,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 1159–1170, IEEE, 2014.
- [70] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, “Scikit-learn: Machine learning in python,” *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.
- [71] M. Kaul, B. Yang, and C. S. Jensen, “Building accurate 3d spatial networks to enable next generation intelligent transportation systems,” in *2013 IEEE 14th International Conference on Mobile Data Management*, vol. 1, pp. 137–146, IEEE, 2013.
- [72] G. Hebrail and A. Berard, “Individual household electric power consumption data set.” <https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption>, 2012.
- [73] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, spring joint computer conference*, pp. 483–485, 1967.
- [74] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” *Foundations and Trends® in Machine learning*, vol. 3, no. 1, pp. 1–122, 2011.

- [75] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan, “A dual coordinate descent method for large-scale linear svm,” in *Proceedings of the 25th international conference on Machine learning*, pp. 408–415, ACM, 2008.
- [76] V. N. Vapnik and S. Kotz, “Estimation of dependences based on empirical data,” vol. 40, 1982.
- [77] E. Osuna, R. Freund, and F. Girosi, “An improved training algorithm for support vector machines,” in *Neural Networks for Signal Processing [1997] VII. Proceedings of the 1997 IEEE Workshop*, pp. 276–285, IEEE, 1997.
- [78] G. X. Yuan, C. H. Ho, and C. J. Lin, “Recent Advances of Large-Scale Linear Classification,” *Proceedings of the IEEE*, vol. 100, pp. 2584–2603, Sept 2012.
- [79] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, “Distributed optimization and statistical learning via the alternating direction method of multipliers,” *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.
- [80] T. Zhang, “Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms,” in *Proceedings of the Twenty-first International Conference on Machine Learning, ICML '04*, (New York, NY, USA), pp. 116–, ACM, 2004.
- [81] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter, “Pegasos: Primal estimated sub-gradient solver for svm,” *Mathematical programming*, vol. 127, no. 1, pp. 3–30, 2011.
- [82] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” in *Proceedings of COMPSTAT'2010*, pp. 177–186, Springer, 2010.
- [83] T. Joachims, “Training Linear SVMs in Linear Time,” in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06*, (New York, NY, USA), pp. 217–226, ACM, 2006.
- [84] Q. V. Le, A. J. Smola, and S. Vishwanathan, “Bundle methods for machine learning,” in *Advances in neural information processing systems*, pp. 1377–1384, 2007.

- [85] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, “LIBLINEAR: A Library for Large Linear Classification,” *J. Mach. Learn. Res.*, vol. 9, pp. 1871–1874, June 2008.
- [86] K.-W. Chang, C.-J. Hsieh, and C.-J. Lin, “Coordinate descent method for large-scale l_2 -loss linear support vector machines,” *The Journal of Machine Learning Research*, vol. 9, pp. 1369–1398, 2008.
- [87] C.-C. Chang and C.-J. Lin, “LIBSVM: A library for support vector machines,” *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [88] Z. A. Zhu, W. Chen, G. Wang, C. Zhu, and Z. Chen, “P-packsvm: Parallel primal gradient descent kernel svm,” in *2009 Ninth IEEE International Conference on Data Mining*, pp. 677–686, Dec 2009.
- [89] C. K. I. Williams and M. Seeger, “Using the nyström method to speed up kernel machines,” in *Advances in Neural Information Processing Systems 13* (T. K. Leen, T. G. Dietterich, and V. Tresp, eds.), pp. 682–688, MIT Press, 2001.
- [90] K. Zhang, L. Lan, Z. Wang, and F. Moerchen, “Scaling up kernel svm on limited resources: A low-rank linearization approach,” in *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics (AISTATS-12)* (N. D. Lawrence and M. A. Girolami, eds.), vol. 22, pp. 1425–1434, 2012.
- [91] A. Rahimi and B. Recht, “Random features for large-scale kernel machines,” in *Advances in Neural Information Processing Systems 20* (J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, eds.), pp. 1177–1184, Curran Associates, Inc., 2008.
- [92] D. Hush, P. Kelly, C. Scovel, and I. Steinwart, “Qp algorithms with guaranteed accuracy and run time for support vector machines,” *Journal of Machine Learning Research*, vol. 7, no. May, pp. 733–769, 2006.
- [93] B. Schölkopf and A. Smola, “Kernel methods and support vector machines,” 2003.

- [94] S. Fine and K. Scheinberg, “Efficient svm training using low-rank kernel representations,” *J. Mach. Learn. Res.*, vol. 2, pp. 243–264, Mar. 2002.
- [95] A. J. Smola and B. Schölkopf, “Sparse greedy matrix approximation for machine learning,” in *Proceedings of the Seventeenth International Conference on Machine Learning, ICML ’00*, (San Francisco, CA, USA), pp. 911–918, Morgan Kaufmann Publishers Inc., 2000.
- [96] P. Drineas and M. W. Mahoney, “On the nystrom method for approximating a gram matrix for improved kernel-based learning,” *J. Mach. Learn. Res.*, vol. 6, pp. 2153–2175, Dec. 2005.
- [97] O. Bousquet and A. Elisseeff, “Stability and generalization,” *The Journal of Machine Learning Research*, vol. 2, pp. 499–526, 2002.
- [98] A. S. Householder, “Unitary Triangularization of a Nonsymmetric Matrix,” *J. ACM*, vol. 5, pp. 339–342, Oct. 1958.
- [99] W. Gander, “Algorithms for the QR decomposition,” *Res. Rep.*, vol. 80, no. 02, pp. 1251–1268, 1980.
- [100] S. G. Johnson, “18.335J - Introduction to Numerical Methods, Fall 2010,” *MIT OpenCourseWare: Massachusetts Institute of Technology*, 2010.
- [101] C.-J. Hsieh, S. Si, and I. S. Dhillon, “Fast prediction for large-scale kernel machines,” in *Advances in Neural Information Processing Systems 27* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 3689–3697, Curran Associates, Inc., 2014.
- [102] M. P. Forum, “Mpi: A message-passing interface standard,” tech. rep., Knoxville, TN, USA, 1994.
- [103] H. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik, “Parallel support vector machines: The cascade svm,” *Advances in neural information processing systems*, vol. 17, pp. 521–528, 2004.

- [104] Y. You, J. Demmel, K. Czechowski, L. Song, and R. Vuduc, “Ca-svm: Communication-avoiding support vector machines on distributed systems,” in *2015 IEEE International Parallel and Distributed Processing Symposium*, pp. 847–859, IEEE, 2015.
- [105] T. Menzies, S. Majumder, N. Balaji, K. Brey, and W. Fu, “500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow),” in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pp. 554–563, IEEE, 2018.
- [106] P. Liu, K.-K. R. Choo, L. Wang, and F. Huang, “Svm or deep learning? a comparative study on remote sensing image classification,” *Soft Computing*, vol. 21, no. 23, pp. 7053–7065, 2017.
- [107] S. Kim, S. Kavuri, and M. Lee, “Deep network with support vector machines,” in *International Conference on Neural Information Processing*, pp. 458–465, Springer, 2013.
- [108] Y. Tang, “Deep learning using linear support vector machines,” *arXiv preprint arXiv:1306.0239*, 2013.
- [109] S. M. Erfani, S. Rajasegarar, S. Karunasekera, and C. Leckie, “High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning,” *Pattern Recognition*, vol. 58, pp. 121–134, 2016.
- [110] Y. Ju, J. Guo, and S. Liu, “A deep learning method combined sparse autoencoder with svm,” in *2015 international conference on cyber-enabled distributed computing and knowledge discovery*, pp. 257–260, IEEE, 2015.
- [111] M. Papadonikolakis and C.-S. Bouganis, “Novel cascade fpga accelerator for support vector machines classification,” *IEEE transactions on neural networks and learning systems*, vol. 23, no. 7, pp. 1040–1052, 2012.
- [112] M. Qasimeh, A. Sagahyroon, and T. Shanableh, “Fpga-based parallel hardware architecture for real-time image classification,” *IEEE Transactions on Computational Imaging*, vol. 1, no. 1, pp. 56–70, 2015.

- [113] X. Song, H. Wang, and L. Wang, "Fpga implementation of a support vector machine based classification system and its potential application in smart grid," in *2014 11th International Conference on Information Technology: New Generations*, pp. 397–402, April 2014.
- [114] M. Ruiz-Llata, G. Guarnizo, and M. Yébenes-Calvino, "Fpga implementation of a support vector machine for classification and regression," in *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–5, July 2010.
- [115] M. Papadonikolakis and C. Bouganis, "A scalable fpga architecture for non-linear svm training," in *2008 International Conference on Field-Programmable Technology*, pp. 337–340, Dec 2008.
- [116] M. B. Rabieah and C. Bouganis, "Fpga based nonlinear support vector machine training using an ensemble learning," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Sep. 2015.
- [117] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf, "A massively parallel fpga-based coprocessor for support vector machines," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pp. 115–122, April 2009.
- [118] P. B. A. Phear, R. K. Rajkumar, and D. Isa, "Efficient non-iterative fixed-period svm training architecture for fpgas," in *IECON 2013 - 39th Annual Conference of the IEEE Industrial Electronics Society*, pp. 2408–2413, Nov 2013.
- [119] D. Anguita, A. Boni, and S. Ridella, "A digital architecture for support vector machines: theory, algorithm, and fpga implementation," *IEEE Transactions on Neural Networks*, vol. 14, pp. 993–1009, Sep. 2003.
- [120] H. Shoukourian, T. Wilde, A. Auweter, and A. Bode, "Predicting the energy and power consumption of strong and weak scaling hpc applications," *Supercomputing frontiers and innovations*, vol. 1, no. 2, pp. 20–41, 2014.

- [121] J. Konečný, H. B. McMahan, and D. Ramage, “Federated optimization: Distributed optimization beyond the datacenter,” in *NIPS Optimization for Machine Learning Workshop*, 2015.
- [122] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtarik, A. T. Suresh, and D. Bacon, “Federated learning: Strategies for improving communication efficiency,” in *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.
- [123] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, “Communication-efficient learning of deep networks from decentralized data,” in *Artificial Intelligence and Statistics*, pp. 1273–1282, PMLR, 2017.
- [124] Q. Yang, Y. Liu, T. Chen, and Y. Tong, “Federated machine learning: Concept and applications,” *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.
- [125] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečný, S. Mazzocchi, H. B. McMahan, *et al.*, “Towards federated learning at scale: System design,” *arXiv preprint arXiv:1902.01046*, 2019.
- [126] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, “Federated learning: Challenges, methods, and future directions,” *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.
- [127] N. Rieke, J. Hancox, W. Li, F. Milletari, H. R. Roth, S. Albarqouni, S. Bakas, M. N. Galtier, B. A. Landman, K. Maier-Hein, *et al.*, “The future of digital health with federated learning,” *NPJ digital medicine*, vol. 3, no. 1, pp. 1–7, 2020.
- [128] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” *International Conference on Learning Representations (ICLR)*, 2016.
- [129] F. Haddadpour, M. M. Kamani, A. Mokhtari, and M. Mahdavi, “Federated learning with compression: Unified analysis and sharp guarantees,” in *International Conference on Artificial Intelligence and Statistics*, pp. 2350–2358, PMLR, 2021.

- [130] C. P. Diehl and G. Cauwenberghs, "Svm incremental learning, adaptation and optimization," in *Proceedings of the International Joint Conference on Neural Networks, 2003.*, vol. 4, pp. 2685–2690, IEEE, 2003.
- [131] S. Ruping, "Incremental learning with support vector machines," in *Proceedings 2001 IEEE International Conference on Data Mining*, pp. 641–642, IEEE, 2001.
- [132] P. E. Utgoff, "Incremental induction of decision trees," *Machine learning*, vol. 4, no. 2, pp. 161–186, 1989.
- [133] R. Polikar, L. Upda, S. S. Upda, and V. Honavar, "Learn++: An incremental learning algorithm for supervised neural networks," *IEEE transactions on systems, man, and cybernetics, part C (applications and reviews)*, vol. 31, no. 4, pp. 497–508, 2001.
- [134] L. Bruzzone and D. F. Prieto, "An incremental-learning neural network for the classification of remote-sensing images," *Pattern Recognition Letters*, vol. 20, no. 11-13, pp. 1241–1248, 1999.
- [135] R. Langone, O. M. Agudelo, B. De Moor, and J. A. Suykens, "Incremental kernel spectral clustering for online learning of non-stationary data," *Neurocomputing*, vol. 139, pp. 246–260, 2014.
- [136] S. Mehrkanoon, O. M. Agudelo, and J. A. Suykens, "Incremental multi-class semi-supervised clustering regularized by kalman filtering," *Neural Networks*, vol. 71, pp. 88–104, 2015.
- [137] T.-J. Chin and D. Suter, "Incremental kernel principal component analysis," *IEEE transactions on image processing*, vol. 16, no. 6, pp. 1662–1674, 2007.
- [138] J. Ye, Q. Li, H. Xiong, H. Park, R. Janardan, and V. Kumar, "Idr/qr: An incremental dimension reduction algorithm via qr decomposition," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 9, pp. 1208–1222, 2005.

- [139] X. Zhang, L. Cheng, D. Chu, L.-Z. Liao, M. K. Ng, and R. C. Tan, “Incremental regularized least squares for dimensionality reduction of large-scale data,” *SIAM Journal on Scientific Computing*, vol. 38, no. 3, pp. B414–B439, 2016.
- [140] L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, *et al.*, *ScaLAPACK users’ guide*. SIAM, 1997.
- [141] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, “Communication-optimal parallel and sequential qr and lu factorizations,” *SIAM Journal on Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012.
- [142] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3. JHU press, 2012.
- [143] A. Barik and J. Honorio, “Exact support recovery in federated regression with one-shot communication,” *arXiv preprint arXiv:2006.12583*, 2020.
- [144] L. He, A. Bian, and M. Jaggi, “Cola: Decentralized linear learning,” in *Advances in Neural Information Processing Systems*, vol. 31, 2018.
- [145] V. Smith, S. Forte, M. Chenxin, M. Takáč, M. I. Jordan, and M. Jaggi, “Cocoa: A general framework for communication-efficient distributed optimization,” *Journal of Machine Learning Research*, vol. 18, p. 230, 2018.
- [146] C. Yu, H. Tang, C. Renggli, S. Kassing, A. Singla, D. Alistarh, C. Zhang, and J. Liu, “Distributed learning over unreliable networks,” in *International Conference on Machine Learning*, pp. 7202–7212, PMLR, 2019.
- [147] A. Qiao, B. Aragam, B. Zhang, and E. Xing, “Fault tolerance in iterative-convergent machine learning,” in *International Conference on Machine Learning*, pp. 5220–5230, PMLR, 2019.
- [148] J. M. Phillips, “Coresets and sketches,” *arXiv preprint arXiv:1601.00617*, 2016.

- [149] P. Drineas, M. W. Mahoney, and S. Muthukrishnan, “Sampling algorithms for l_2 regression and applications,” in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pp. 1127–1136, 2006.
- [150] D. Feldman, M. Monemizadeh, C. Sohler, and D. P. Woodruff, “Coresets and sketches for high dimensional subspace approximation problems,” in *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pp. 630–649, SIAM, 2010.
- [151] K. L. Clarkson and D. P. Woodruff, “Numerical linear algebra in the streaming model,” in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 205–214, 2009.
- [152] A. Fallah, A. Mokhtari, and A. Ozdaglar, “Personalized federated learning with theoretical guarantees: A model-agnostic meta-learning approach,” in *Advances in Neural Information Processing Systems*, vol. 33, pp. 3557–3568, 2020.
- [153] G. Widmer and M. Kubat, “Learning in the presence of concept drift and hidden contexts,” *Machine learning*, vol. 23, no. 1, pp. 69–101, 1996.
- [154] T. F. Chan, “Rank revealing qr factorizations,” *Linear algebra and its applications*, vol. 88, pp. 67–82, 1987.
- [155] J. Dass, V. Sarin, and R. N. Mahapatra, “Fast and communication-efficient algorithm for distributed support vector machine training,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 5, pp. 1065–1076, 2019.
- [156] D. P. Bertsekas, “Incremental least squares methods and the extended kalman filter,” *SIAM Journal on Optimization*, vol. 6, no. 3, pp. 807–822, 1996.
- [157] G. Mateos and G. B. Giannakis, “Distributed recursive least-squares: Stability and performance analysis,” *IEEE Transactions on Signal Processing*, vol. 60, no. 7, pp. 3740–3754, 2012.

- [158] X. Chen, J. Ji, C. Luo, W. Liao, and P. Li, “When machine learning meets blockchain: A decentralized, privacy-preserving and secure design,” in *2018 IEEE International Conference on Big Data (Big Data)*, pp. 1178–1187, IEEE, 2018.
- [159] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, “Blockchain challenges and opportunities: A survey,” *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.
- [160] H. Benmeziame, K. E. Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang, “A comprehensive survey on hardware-aware neural architecture search,” *arXiv preprint arXiv:2101.09336*, 2021.
- [161] M. S. Abdelfattah, Ł. Dudziak, T. Chau, R. Lee, H. Kim, and N. D. Lane, “Best of both worlds: Automl codesign of a cnn and its hardware accelerator,” in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.
- [162] B. Zoph and Q. V. Le, “Neural architecture search with reinforcement learning,” *arXiv preprint arXiv:1611.01578*, 2016.
- [163] Y. Liu, Y. Sun, B. Xue, M. Zhang, and G. Yen, “A survey on evolutionary neural architecture search,” *arXiv preprint arXiv:2008.10937*, 2020.
- [164] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, “Mnasnet: Platform-aware neural architecture search for mobile,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.
- [165] H. Cai, L. Zhu, and S. Han, “Proxylessnas: Direct neural architecture search on target task and hardware,” *arXiv preprint arXiv:1812.00332*, 2018.
- [166] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, “Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search,” in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10734–10742, 2019.

- [167] W. Jiang, X. Zhang, E. H.-M. Sha, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, “Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search,” in *Proceedings of the 56th Annual Design Automation Conference 2019*, pp. 1–6, 2019.
- [168] C. Hao, Y. Chen, X. Liu, A. Sarwari, D. Sew, A. Dhar, B. Wu, D. Fu, J. Xiong, W.-m. Hwu, *et al.*, “Nais: Neural architecture and implementation search and its applications in autonomous driving,” in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE, 2019.
- [169] Y. Li, C. Hao, X. Zhang, X. Liu, Y. Chen, J. Xiong, W.-m. Hwu, and D. Chen in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.
- [170] D. Dang, J. Dass, and R. Mahapatra, “Convlight: A convolutional accelerator with memristor integrated photonic computing,” in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pp. 114–123, 2017.
- [171] J. Dass, Y. Narawane, R. Mahapatra, and V. Sarin, “Fpga-based distributed edge training of svm,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA ’19*, (New York, NY, USA), p. 121, Association for Computing Machinery, 2019.
- [172] Z. Chen and B. Liu, “Lifelong machine learning,” *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 12, no. 3, pp. 1–207, 2018.