Avoiding Communication in First Order Methods for Optimization

Aditya Devarakonda

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-92
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-92.html

July 24, 2018

Avoiding Communication in First Order Methods for Optimization

by

Aditya Devarakonda

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

and the Designated Emphasis

in

Computational and Data Science and Engineering

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James W. Demmel, Chair
Professor Michael W. Mahoney
Professor Adityanand Guntuboyina

Summer 2018

Avoiding Communication in First Order Methods for Optimization

Abstract

Avoiding Communication in First Order Methods for Optimization

by

Aditya Devarakonda

Doctor of Philosophy in Computer Science
and the Designated Emphasis

in

Computational and Data Science and Engineering

University of California, Berkeley

Professor James W. Demmel, Chair

Machine learning has gained renewed interest in recent years due to advances in computer hardware (processing power and high-capacity storage) and the availability of large amounts of data which can be used to develop accurate, robust models. While hardware improvements have facilitated the development of machine learning models in a single machine, the analysis of large amounts of data still requires parallel computing to obtain shorter running times or where the dataset cannot be stored on a single machine.

In addition to hardware improvements, algorithm redesign is also an important direction to further reduce running times. On modern computer architectures, the cost of moving data (communication) from main memory to caches in a single machine is orders of magnitude more expensive than the cost of performing floating-point operations (computation). On parallel machines the cost of moving data from one processor to another over an interconnection network is the most expensive operation. The large gap between computation and communication suggests that algorithm redesign should be driven by the goal of avoiding communication and, if necessary, decreasing communication at the expense of additional computation.

Many problems in machine learning solve mathematical optimization problems which, in most non-linear and non-convex cases, requires iterative methods. This thesis is focused on deriving communication-avoiding variants of the block coordinate descent method, which is a first-order method that has strong convergence rates for many optimization problems. Block coordinate descent is an iterative algorithm which at each iteration samples a small subset of rows or columns of the input matrix, solves a subproblem using just the chosen rows or columns, and obtains a partial solution. This solution is then iteratively refined until the optimal solution is reached or until convergence criteria are met. In the parallel case, each iteration of block coordinate descent requires communication. Therefore, avoiding communication is key to attaining high performance.

This thesis adapts well-known techniques from existing work on communication-avoiding (CA) Krylov and s-step Krylov methods. CA-Krylov methods unroll vector recurrences and rearrange the sequence of computation in way that defers communication for s iterations, where s is a tunable parameter. For CA-Krylov methods the reduction in communication cost comes at the expense of numerical instability for large values of s.

We apply a similar recurrence unrolling technique to block coordinate descent in order to obtain communication-avoiding variants which solve the L2-regularized least-squares, L1-regularized least-squares, Support Vector Machines, and Kernel problems. Our communication-avoiding variants reduce the latency cost by a tunable factor of s at the expense of a factor of s increase in computational and bandwidth costs for the L2 and L1 least-squares and SVM problems. The CA-variants for these problems require additional computation and bandwidth in order to update the residual vector. For CA-kernel methods the computational and bandwidth costs do not increase. This is because the CA-variants of kernel methods can reuse elements of the kernel matrix already computed and therefore do not need to compute and communicate additional elements of the kernel matrix.

Our experimental results illustrate that our new, communication-avoiding methods can obtain speedups of up to $6.1\times$ on a Cray XC30 supercomputer using MPI for parallel processing. For CA-kernel methods we show modeled speedups of $26\times$, $120\times$, and $197\times$ for MPI on a predicted Exascale system, Spark on a predicted Exascale system, and Spark on a cloud system, respectively. Furthermore we also experimentally confirm that our algorithms are numerically stable for large values of s.

Finally, we also present an adaptive batch size technique which reduces the latency cost of training convolutional neural networks (CNN). With this technique we have achieved speedups of up to $6.25\times$ when training CNNs on up to 4 NVIDIA P100 GPUs. Furthermore, we were able to train the ImageNet dataset using the ResNet-50 network with a batch size of up to $524{,}228$ which would allow neural network training to attain a higher fraction of peak GPU performance than training with smaller batch sizes.

To my parents Satya and Srinivas and to Aravind and Holly.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to acknowledge first and foremost the support from my advisor, Jim Demmel. His guidance and advice has had a great influence in shaping my work. He has been a wonderful mentor over the past six years and I have learned a lot from Jim's rigorous approach to the design and analysis of parallel algorithms. Thanks to my co-advisor Michael Mahoney whose advice and guidance shaped the broader machine learning aspects in this thesis. In particular, the chapter on kernel methods has benefited greatly from discussions with Michael. I would also like to thank Adityanand Guntuboyina for his feedback during the early stages of this thesis. Last but not least, I would like to thank Kathy Yelick for her comments and feedback on architectural trends and performance tuning which have directly influenced the parallel implementations of the algorithms presented in this thesis.

I would also like to acknowledge my co-authors and collaborators Jim Demmel, Kimon Fountoulakis, Michael Garland, Michael Mahoney, and Maxim Naumov. Thanks to Kimon Fountoulakis who has contributed to the research in several chapters. I have gained a lot of insight about machine learning and optimization through our collaboration. Thanks to Michael Garland and Maxim Naumov from whom I've learned a lot about deep learning and for their contributions to Chapter 7.

I'd like to thank members of the Bebop group (alphabetical order): Grey Ballard, Ben Brock, Aydın Buluç, Erin Carson, Razvan Carbunescu, Orianna DeMasi, Grace Dinh, Michael Driscoll, Marquita Ellis, Andrew Gearhart, Evangelos Georganas, Laura Grigori, Nick Knight, Penporn Koanantakool, Becca Roelofs, Alex Rusciano, Oded Schwartz, Harsha Simhadri, Edgar Solomonik, and Yang You. I'd like to thank Michael Mahoney's group (alphabetical order): Alex Gittens, Fred Roosta, Julian Shun, and Shusen Wang, Peng Xu, and Jiyan Yang. I couldn't have asked for better colleagues.

Thanks to Ria Briggs, Roxana Infante, Tamille Choteau, Kostadin Ilov, Tiffany Reardon, Shirley Salanio, Audrey Sillers, Angela Waxman, Matthew Santillan and the rest of the EECS staff for their positivity and support. Thanks to the BiasBusters and EECS Peers communities. Their tireless work and enthusiasm was a constant source of motivation during the writing of this thesis.

I would like to acknowledge several funding sources whose support has made it possible to complete this thesis. This includes support from the ParLab which was funded in part by Microsoft (Award #024263), Intel (Award #024894) and by matching funding by U.C. Discovery (Award #DIG0710227), by NSF Grants CNS-0720906 and CCF-0747390. I would also like to acknowledge support from an EECS department fellowship, the National Science Foundation for the Graduate Research Fellowship (Grant No. DGE 1106400), the Department of Energy Office of Science (DEGAS project), Cray, and NVIDIA (for support during my internship). Thanks to the National Energy Research Scientific Computing Center for access to their supercomputing resources.

Chapter 1

Introduction

The volume of data currently generated by sensor networks, social media, and computational science has made manual data processing nearly impossible. The need to automatically process and interpret this data has driven the rapid development and deployment of machine learning algorithms and tools which have enabled progress in many critical fields. For example, the automotive industry is researching and testing autonomous driving technologies to make the roads safer. The pharmaceutical and biotechnology industries are leveraging machine learning in order to reduce the decade long search to find breakthrough drugs. The field of scientific computing is also turning to machine learning for answers to many important questions. For example, in high performance computing, machine learning is being used to automatically tune machine and algorithm hyperparameters in order to maximize performance. In high-energy physics, deep neural networks are being used to find important features in images from particle collisions. The number of fields that are discovering new insights with the aid of machine learning is continually increasing.

Machine learning encompasses many different problems like regression, classification, clustering, and dimensionality reduction. All of these problems have the common task of creating models from an input dataset which can subsequently be used as predictors (i.e., accurately classify, cluster, etc.) on new data. All of these applications require some form of mathematical optimization to find the best predictor (i.e., one that minimizes the prediction error) obtained by maximizing or minimizing some objective function representing the machine learning problem being solved.

The rise of machine learning has coincided with a slowdown of Moore's law and stagnation of advances in single-core processor technology. Consequently, this has led many manufacturers to increase the number of cores per processor rather than rely on improvements in clock-frequency alone[1]. This shift in processor technology suggests that all machine learning algorithms must be parallel in order to take advantage of the additional cores and in order to achieve peak processor performance. This is especially true for many machine learning algorithms which use linear algebra kernels (e.g., matrix multiply, solving a system of linear equations, etc.). Fast machine learning

[1] Moore's law states that transistor densities (i.e., smaller transistors) double approximately every 12 to 18 months. Moore's law is closely related to Dennard scaling [43] which states that transistor size is roughly inversely proportional to clock frequency.

algorithms are critical to making progress in the numerous fields relying on them and achieving this goal requires advances in computer architecture, algorithm design, and software development.

1.1 Iterative Machine Learning

Machine learning problems, especially those which require mathematical optimization, are iterative[2]. By iterative, we mean that an initial guess of the solution is refined until convergence criteria are met. Iterative methods are especially useful when solution accuracy less than machine precision is sufficient. In practice, machine learning applications require just a few digits of accuracy (typically less than single-precision accuracy) in order to avoid overfitting the solution to the input dataset[3].

This thesis studies block coordinate descent [122], which is a first-order method (i.e., method which only use the first derivative of the optimization problem) to solve various machine learning optimization problems. Block coordinate descent is an iterative method which selects a subset of the rows or columns from the input dataset, solves the optimization problem for just the chosen rows or columns, and updates the elements (or coordinates) of the solution which correspond to the indices of the rows or columns chosen. The block size, i.e., the number of rows or columns chosen, and how the rows or columns are chosen are important tuning parameters that depend on the dataset and the parameters of the hardware used. We will study several choices of block size but assume that the rows or columns are chosen uniformly at random.

1.2 Communication-Avoiding Algorithms

On modern computer architectures, computation, i.e., the time to perform a floating-point operation, is orders of magnitude less than the time to communicate. By communication we mean the time to move data between levels of the memory hierarchy or the time to move data from one processor to another over an interconnection network (infiniband, ethernet, etc.). This suggests that the traditional approach to algorithm analysis which relies on just computational complexity is not accurate and, in reality, that there is a tradeoff between computational and communication complexity. A better model for an algorithm's running time would be the sum of computational cost and communication cost.

Machine learning problems routinely process datasets which do not fit in the main-memory of a single machine. As a result, efficiently processing this data requires large numbers of machines. While increasing the number of machines can result in fast computation time it also requires expensive communication. Since communication is orders of magnitude more expensive than computation, performance degrades when communication becomes the dominant cost. This is further exacerbated for iterative algorithms, like block coordinate descent, which require communication

[2]Note that solving linear systems can be solved directly. However, for non-linear and non-convex optimization problems iterative methods are often the only option.

[3]Overfitting decreases the solution's ability to generalize and perform as a good predictor for new data.

at every iteration until convergence. Therefore, rearranging machine learning algorithms to avoid communication is a requirement to scaling these algorithms efficiently.

Communication-avoiding (CA) algorithms are a new class of algorithms which avoid communication through careful algorithmic transformations [4] and attain large speedups on modern parallel architectures. Much of numerical linear algebra has been reorganized to avoid communication and has led to significant performance improvements over existing state-of-the-art libraries [4, 3, 20, 67, 113, 121]. Progress has also been made in iterative numerical linear algebra with the development of CA-Krylov methods [20, 40, 67] and s-step methods [119, 29, 30, 72, 120]. The results from CA-Krylov methods are particularly relevant to this thesis. The CA-Krylov methods work extends existing s-step Krylov methods research by combining a new, matrix powers kernel [40, 67, 85, 86] with extensively modified s-step Krylov methods to avoid communication [20, 40, 67]. These methods avoid communication by unrolling the vector update recurrences by a factor of s and eliminating dependencies between the iterations (i.e., communication) through the use of the matrix powers kernel along with an algebraic re-arrangement of the vector updates to break dependencies. In practice, this rearrangement has resulted in large speedups. One drawback of CA-Krylov methods is that the convergence and stability may be weaker than the standard Krylov methods in finite precision as s increases. As a result, experimentally confirming the numerical stability, in addition to deriving CA-variants of block coordinate descent methods, is an important goal.

1.3 Thesis Contributions

Communication-avoiding algorithms have been studied for several decades and has resulted in the development of numerous new algorithms that attain large speedups over the prior state-of-the-art algorithms. In addition to the development of new algorithms, this work has also contributed theoretical lower bounds on communication and numerical error analysis. In particular, this thesis is motivated by the techniques presented s-step and CA-Krylov methods literature, which we extend in this thesis to iterative block coordinate descent algorithms for convex optimization problems.

We begin with background (Chapter 2) on notation, theoretical performance model, communication cost, relationship to CA-Krylov methods, and relationship to existing communication reducing approaches in machine learning. Chapter 3 derives and analyzes communication-avoiding algorithms which solve the primal and dual L2-regularized least-squares problem and comparison to Krylov methods. Chapter 4 focuses on the L1-regularized least-squares problem which introduces a non-linear optimization problem. Chapter 5 derives communication-avoiding algorithms for binary classification using Support Vector Machines (SVM). Chapter 6 introduces and derives communication-avoiding algorithms for kernel methods where regression and binary classification is performed in a high-dimensional Hilbert space. Chapter 7 presents an adaptive batch size technique which reduces communication for image classification in deep Convolutional Neural Networks. Some of the material presented in this thesis have appeared in published work and, where applicable, we cite the published manuscript, list co-authors, and describe the overlap. The primary results and contributions of this thesis are:

- The derivation of new communication-avoiding block coordinate descent methods which solve the L2-regularized least-squares (Chapter 3), L1-regularized least-squares (Chapter 4), Support Vector Machines (Chapter 5), and kernel problems (Chapter 6).
- Algorithm analysis to derive computation, communication, and storage costs of the standard block coordinate descent and new, communication-avoiding block coordinate descent algorithms in Chapters 3, 4, and 5 for the various optimization problems. The analysis proves the primary claim of this thesis, that the new communication-avoiding block coordinate descent algorithms reduce the latency cost by a tunable factor, s, at the expense of a factor of s additional computation and bandwidth.
- Derivations of computation, communication and storage costs of standard block coordinate descent and new, communication-avoiding block coordinate descent algorithms for kernel methods in Chapter 6. The proofs show that the communication-avoiding algorithms reduce the latency cost by a factor of s and have the same computation and bandwidth costs as the standard algorithm.
- Implementation of all the algorithms presented in Chapters 3-6 in MATLAB and presentation of experimental results confirming the numerical stability of the communication-avoiding block coordinate descent algorithms. These experiments illustrate that the new algorithms have the same convergence behavior as the standard algorithms for large values of s. The values of s are chosen to be very large, beyond practical interest, in order to stress the potential numerical instability as much as possible in order to provide confidence that our algorithms are stable for a large range of s values.
- Implementation of algorithms described in Chapters 3-5 in C++ using MPI [62] for parallel processing on a Cray XC30 supercomputer (Edison). We use 1D-block row layout when solving the primal problems and 1D-block column layout when solving the dual problems.
- Strong scaling, running time breakdown, and convergence vs. running time experiments in Chapters 3-5 which illustrate that the communication-avoiding block coordinate descent algorithms can attain speedups of up to $6.1\times$ over the standard algorithms on up to $12,288$ MPI processes with datasets obtained from the LIBSVM [26] data repository.
- Modeled strong scaling and speedup experiments for kernel methods in Chapter 6 which illustrate that our communication-avoiding algorithms can attain model speedups of up to $26.97\times$, $120.44\times$, and $147.48\times$ on a predicted Exascale system with MPI, with Spark, and a cloud system with Spark, respectively.
- The derivation of a new adaptive batch size technique for training deep convolutional neural networks in Chapter 7. We show that the batch size can be adaptively increased in addition to learning rate decay.
- Implementation of the adaptive batch size technique written in PyTorch [98] with GPU acceleration.

- Convergence vs. iterations experiments to illustrate that the adaptive batch size technique allows us to dynamically increase the batch size without significantly altering the convergence of training and test error for Alexnet, ResNet20, ResNet50, and VGG19 networks for image classification problems.
- Performance experiments on 1 NVIDIA P100 GPU which illustrates speedups of up to $6.25\times$ for ResNet-20 and $3.54\times$ for VGG19 when training on the CIFAR-100 dataset.
- Convergence vs. iterations experiments which illustrate that the ImageNet dataset can be successfully trained using the ResNet-50 network on 4 NVIDIA P100 GPUs with a batch size of $524,288$, which would allow neural network training to attain a higher fraction of peak GPU performance than training with smaller batch sizes.

Chapter 2

Background

We begin this chapter by introducing the notation used in this thesis. We use A to refer to the input matrix with m rows, which are the data points, and n columns, which are the features. We use y for the m-dimensional vector of labels, x for the n-dimensional solution vector of primal optimization problems, and α for the m-dimensional solution vector of dual optimization problems. We use the scalar, b, to refer to the block size used for block coordinate descent. When we refer to algorithms as coordinate descent it is an implicit way of saying that $b = 1$. We will use standard notation for asymptotic computation and communication complexities of the algorithms analyzed. $f(n) = O(g(n))$ states that there exists a constant $c > 0$ such that $|f(n)| \leq c|g(n)|$ for large enough n. We use $\arg\min$ when writing optimization problems to mean that the solution to the optimization problem is the argument which minimizes the optimization problem. The notation $[n]$ is used as short form for $\{1, 2, \ldots n\}$. When we use $\log$ in the algorithm analysis sections, it refers to logarithms with base 2.

2.1 Theoretical Performance Model

In this work we assume that we are working on a homogenous parallel machine with P processors that are interconnected using a binary tree. We assume that processors can communicate with each other using point-to-point messages and collective messages. We use a parallel performance model that has three terms: computation, bandwidth, and latency. The first term represents the floating-point operations cost, bandwidth refers to the amount of data that is moved over a network, and latency refers to the cost of sending a message. Note that each term of this model has associated hardware costs. We use γ to refer to the amount of time it takes to perform a floating-point operation on data stored locally on the processor, β to refer to the reciprocal bandwidth of the interconnection network, and ζ to refer to the hardware overhead for sending a message. We assume that each processor can execute at peak processor speed (i.e. $1/\gamma$) and ignore sequential communication (i.e. time to move data from DRAM to caches). We use F to refer to an algorithm's per processor floating-point operations cost along the critical path, W to refer to the number of words moved from one processor to another, and L to refer to the number of messages sent or received

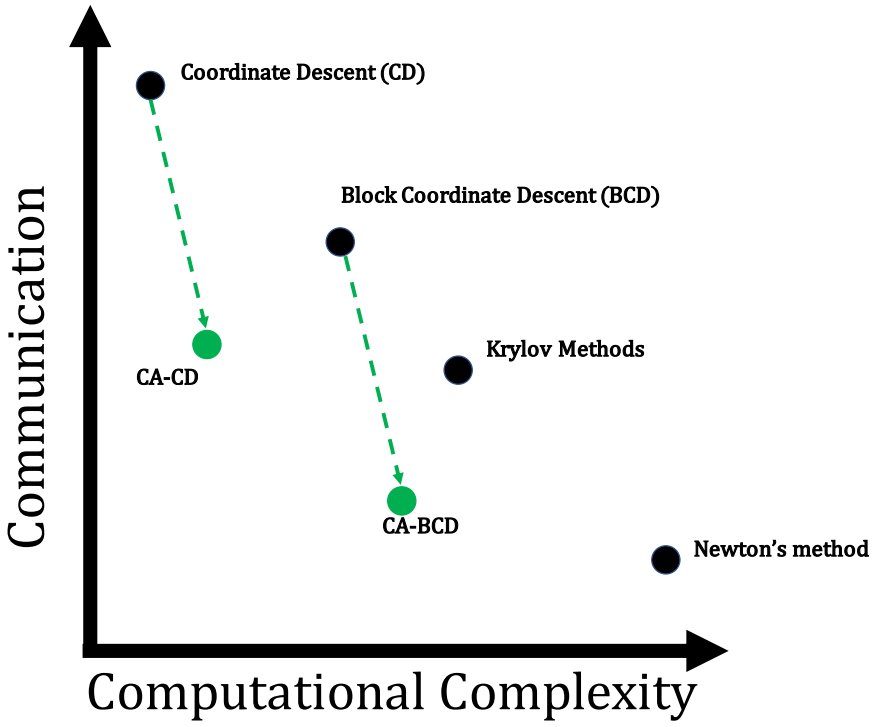


Figure 2.1: Illustration of where coordinate descent (CD), block coordinate descent (BCD), Krylov methods (for quadratic problems), and Newton's method lie in the computation and communication tradeoff space. The communication-avoiding variants of CD and BCD reduce communication at the expense of computation. Note that this figure suggests two ways to reduce communication. One way is to derive communication-avoiding variants and the other to select an optimization method which has the appropriate computation and communication tradeoff for the given parallel computers.

by a processor. From these quantities, the running time for an algorithm can be bounded by

$$T \leq \gamma F + \beta W + \zeta L. \tag{2.1}$$

We assume equality in (2.1) when we use the model to predict the running time of an algorithm. Finally, we use MPI [62] as the communication library to implement our parallel algorithms. Since we assume a binary tree network topology, the cost of an MPI_Allreduce collective operation for a message size of n words costs $n \log P$ in bandwidth and $\log P$ in latency.

2.2 Related Work

We begin a summary of related work by first comparing coordinate descent (CD) and block coordinate descent (BCD) to other iterative methods which can be used to solve optimization problems.

In particular, we will consider comparisons with Krylov methods and Newton's method. We assume that the matrix A is distributed between the processors and that the layout is chosen such that the computation, bandwidth, and latency costs are as small as possible. Since A is distributed, any computation with A requires communication. Since iterative methods use A at every iteration, we can use number of iterations as a proxy for the latency cost. CD only requires dot-products and is computationally the cheapest in terms of flops per iteration but may require many iterations to converge. BCD requires a Gram matrix computation with a (tunable) block size number of rows or columns of A. Therefore, BCD requires more flops per iteration which is more expensive than CD but can converge in fewer iterations. Krylov methods iteratively compute parallel matrix-vector products with A (i.e. uses all rows and columns of A) and requires fewer iterations to converge than CD or BCD. Newton's method for optimization is a second-order method, which requires computing the Hessian. Once the Hessian is computed, Newton's method converges rapidly and requires the fewest number of iterations. Figure 2.1 illustrates the difference between CD, BCD, Krylov, and Newton's method in the computation and communication tradeoff space. Our communication-avoiding CD and BCD algorithms (plotted in green) reduce the communication cost by a factor of s at the expense of a factor of s additional bandwidth and computation. This illustrates that communication can be reduced in two ways: implicitly by reducing the number of iterations (e.g. pre-conditioning and/or using a higher-order method) and explicitly by deriving communication-avoiding versions of the existing method. Our work focuses on the latter approach.

We will now summarize the related work and highlight the differences with this thesis. The CoCoA framework [111] is a parallel optimization framework which starts by dividing data points from an input matrix between parallel machines and then approximately solves a given optimization problem using a second order method on only data points stored locally on each machine. Naturally, solving the optimization problem locally on each machine would not yield a good solution to the global problem on all data points. In order to incorporate the solutions from non-local data points all machines communicate their local solutions and combine them by summing or averaging. CoCoA reduces communication by altering the number of local iterations (i.e., how accurately the local problem is solved) performed before communicating local solutions. Suppose that all data points are stored on a single processor, then CoCoA is essentially a Newton-type method which converges faster and with no communication, but at the expense of higher flops cost per iteration. However, if each processor stores only one data point, then CoCoA acts like dual coordinate descent which has slower convergence and lots of communication, but with lower flops cost per iteration. CoCoA strikes a tradeoff between convergence, communication, and computational complexity by storing a subset of the data points (greater than 1 but less than m) and allowing a tunable number of local iterations before communicating. As a result, CoCoA implicitly reduces communication by selecting a trade off point between convergence rate and communication. In contrast, our technique explicitly reduces communication by selecting a trade off point between computation, bandwidth, and latency without altering the convergence behavior. Since CoCoA uses a local dual coordinate descent algorithm, additional speedups might be possible by replacing it with our communication-avoiding dual coordinate descent algorithm. This is left for future work.

HOGWILD! [100] is a lock-free approach to stochastic gradient descent (SGD). SGD is an algorithm which solves optimization problems by iteratively sampling a small set of data points from

the input matrix and solves a sub-problem on only those data points. When SGD is parallelized, the small set of data points can be distributed between different processors and each processor can solve the optimization problem on just its data points. Once each processor obtains a local solution, the global solution can be obtained by summing local solutions. Note that the summation requires inter-processor synchronization, which HOGWILD! attempts to avoid. Unlike the standard parallel SGD, HOGWILD! does not enforce synchronization. Without synchronization, the global solution may be updated using stale local solutions (i.e., a processor has updated the global solution using a local solution that is one of more iterations behind other processors). The main results in HOGWILD! show that if the solution updates are sparse (i.e. each processor only modifies a part of the solution) or that the solution updates are not too stale then running without synchronization does not affect the final solution with high probability.

CYCLADES [95] uses a conflict-graph analysis of the solutions updates associated with groups of data points in SGD to ensure that these updates are conflict free. If that is the case then data points which lead to conflict-free updates can be assigned to different processors. As a result, the race conditions associated with HOGWILD! can be avoided but at the expense of building and analyzing a conflict-graph. In contrast to HOGWILD! and CYCLADES, our technique does not alter convergence rates and does not require conflict-free updates.

DUAL-LOCO [65] introduces a framework which reduces communication between processors by communicating a small matrix of randomly projected features. While DUAL-LOCO requires just one communication round, it does not apply to proximal least-squares problems (i.e., Lasso, elastic-net, etc.) and introduces an (additive) approximation error.

CA-SVM [124] eliminates communication in SVM by performing an initial K-means clustering as a pre-processing step to partition the data and subsequently training SVM classifiers locally on each processor. Communication is reduced significantly, but at the cost of accuracy. Like CoCoA, CA-SVM uses dual coordinate descent to solve the local problem. Replacing it with our CA-DCD algorithm may yield additional speedups.

P-packSVM [131] applies a similar approach to ours and derives a CA version of SVM using stochastic subgradient descent on the primal SVM problem. However, we generalize the technique to more optimization problems and, specifically, to dual coordinate descent for SVM which has been shown to converge faster than subgradient descent [68].

There are many algorithms which solve least-squares [90, 47, 101, 102, 91], SVM, and kernel problems [97, 27, 107, 81, 68]. In this thesis we focus on randomized variants of accelerated and non-accelerated Coordinate Descent (CD) and Block Coordinate Descent (BCD) since they have optimal convergence rates among the class of first-order methods [47, 90, 101, 102, 68, 27].

2.3 Parallel Block Coordinate Descent

Figure 2.2 illustrates the computations in parallel BCD. For the figure, we assume that A is tall and skinny with more data points than features (we do not assume anything about the shape of A in later chapters). We assume that A is partitioned row-wise so that the dot-products required in BCD can be performed in parallel by all processors. Similarly we assume that vectors in the

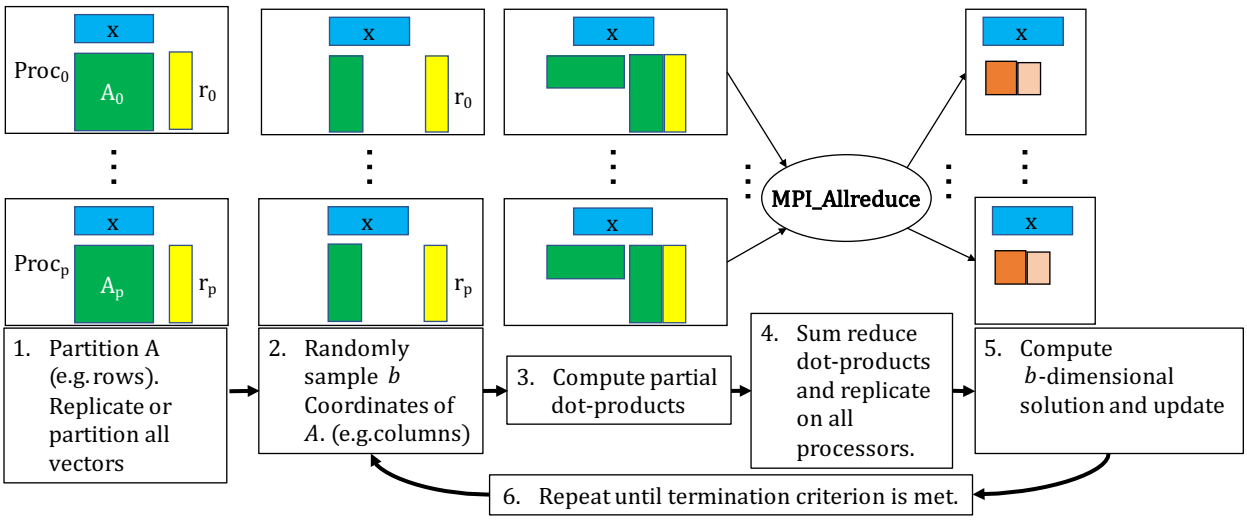


Figure 2.2: A high-level depiction of the Block Coordinate Descent method (independent of the minimization problem being solved). The matrix A is 1D-row partitioned and w.l.o.g. depicted as being dense. Vectors in the partitioned dimension are also partitioned (in this case residual vector, r). Vectors in the non-partitioned dimension and all scalars are replicated (in this case x). Each processor selects the same column indices (by using the same random generator seed). Computation of (partial) dot-products is a local GEMM operation. After that, the results are combined using an all-reduce with summation. Due to data replication all processors can independently compute this iteration's solution and perform vector updates.

row-dimension are partitioned, but that vectors in the column-dimension are replicated. Note that $b > 1$ implies that dot-products become GEMM (GEneral Matrix-Multiplication) computations. After the GEMM computations an MPI_Allreduce with summation combines each processor's contributions. Since Allreduce redundantly stores results on all processors, computing the solution to the subproblem and updating all vectors can be performed without communication. Finally, this process is repeated until a termination criterion is met. The mathematical details (and complexity) of solving the subproblem and updating vectors might vary based on the optimization problem (proximal least-squares, SVM, etc.). However, parallel BCD's sequence of operations can be summarized by Figure 2.2. Each iteration requires synchronization, therefore, avoiding these synchronization costs could lead to faster and more scalable BCD methods.

Chapter 3

Avoiding Communication in L2-Regularized Least-Squares

The algorithms, analysis, and experimental results presented in this chapter are a result of joint work with co-authors James Demmel, Kimon Fountoulakis, and Michael W. Mahoney. This work also appeared as a technical report [45] and is currently under reviewer for publication.

We extend the communication-avoiding technique to machine learning where scalable algorithms are especially important given the enormous amount of data. Block coordinate descent methods are routinely used in machine learning to solve optimization problems [90, 101, 122]. Given a sparse dataset $A \in \mathbb{R}^{m \times n}$ where the rows are data points and the columns are features, the block coordinate descent method can compute the regularized or unregularized least squares solution by iteratively solving a subproblem using a block of b columns (or features) of A [90, 101, 122]. This process is repeated until the solution converges to a desired accuracy or until the number of iterations has reached a user-defined limit. If A is distributed (in 1D-row or 1D-column layout) across P processors then the algorithm communicates at each iteration in order to solve the subproblem. As a result, the running time for such methods is often dominated by communication cost which increases with P. We briefly summarize our contributions:

- We present communication-avoiding algorithms for block coordinate descent and block dual coordinate descent that *p*rovably reduce the latency cost by a factor of s.
- We analyze the operational, communication and storage costs of the classical and our new communication-avoiding algorithms under two data partitioning schemes and describe their performance tradeoffs.
- We perform numerical experiments to illustrate that the communication-avoiding algorithms are numerically stable for all choices of s tested.
- We show performance results to illustrate that the communication-avoiding algorithms can be up to $6.1\times$ faster than the standard algorithms on up to 1024 nodes of a Cray XC30 supercomputer using MPI.

Summary of Ops and Memory costs			
Algorithm	Data layout	Ops cost (F)	Memory cost (M)
BCD	1D-row	$O\left(\frac{Hb^2fm}{P} + Hb^3\right)$	$O\left(\frac{fmn+m}{P} + b^2 + n\right)$
CA-BCD		$O\left(\frac{Hb^2sfm}{P} + Hb^3\right)$	$O\left(\frac{fmn+m}{P} + b^2s^2 + n\right)$
BDCD	1D-column	$O\left(\frac{H'b'^2fn}{P} + H'b'^3\right)$	$O\left(\frac{fmn+n}{P} + b'^2 + m\right)$
CA-BDCD		$O\left(\frac{H'b'^2sfn}{P} + H'b'^3\right)$	$O\left(\frac{fmn+n}{P} + b'^2s^2 + m\right)$

Summary of Communication costs			
Algorithm	Data layout	Latency cost (L)	Bandwidth cost (W)
BCD	1D-row	$O\left(H \log P\right)$	$O\left(Hb^2 \log P\right)$
CA-BCD		$O\left(\frac{H}{s} \log P\right)$	$O\left(Hb^2s \log P\right)$
BDCD	1D-column	$O\left(H' \log P\right)$	$O\left(H'b'^2 \log P\right)$
CA-BDCD		$O\left(\frac{H'}{s} \log P\right)$	$O\left(H'b'^2s \log P\right)$

Table 3.1: Ops (F), Latency (L), Bandwidth (W) and Memory per processor (M) costs comparison along the critical path of classical BCD (Thm. 3.3.1), BDCD (Thm. 3.3.2) and communication-avoiding BCD (Thm. 3.3.6) and BDCD (Thm. 3.3.7) algorithms for 1D-block row and 1D-block column data partitioning, respectively. H and H' are the number of iterations and b and b' are the block sizes for BCD, and BDCD. We assume that $A \in \mathbb{R}^{m \times n}$ is sparse with fmn non-zeros that are uniformly distributed, $0 < f \leq 1$ is the density of A, P is the number of processors and s is the recurrence unrolling parameter. We assume that the $b \times b$ and $b' \times b'$ Gram matrices computed at each iteration for BCD and BDCD, respectively, are dense.

Our results reduce the latency cost (at the expense of additional flops and bandwidth) in the primal and dual block coordinate descent methods by a factor of s on distributed-memory architectures, for dense and sparse updates without changing the convergence behavior, in exact arithmetic. Our results show that our CA-methods attain speedups despite the increase in flops and bandwidth costs. Table 3.1 summarizes the critical path costs of the algorithms considered in this paper. Hereafter we refer to the primal method as block coordinate descent (BCD) and the dual method as block dual coordinate descent (BDCD). The proofs in this paper assume that A is sparse with fmn non-zeros that are uniformly distributed where $0 < f \leq 1$ is the density of A (i.e. $f = \frac{nnz(A)}{mn}$). Each iteration of BCD samples b columns of A (resp. b' rows of A for BDCD), uniformly at random without replacement. The resulting $m \times b$ (resp. $b' \times n$ for BDCD) sampled matrix contains fbm (resp. $fb'n$ for BDCD) non-zeros. These assumptions simplify our analysis and provide insight into scaling behavior for ideal sparse inputs. We leave extensions of our proofs to general sparse matrices for future work. The rest of the chapter is organized as follows: Section 3.1 summarizes existing methods for solving the regularized least squares problem and the communication cost model used to analyze our algorithms. Section 3.2 presents the communication-avoiding derivations of the BCD and BDCD algorithms. Section 3.3 analyzes the

Ops and Memory Costs Comparison		
Algorithm	Ops cost (F)	Memory cost (M)
Krylov methods [4]	$O\left(\frac{kfmn}{P}\right)$	$O\left(\frac{fmn}{P} + \min(m,n) + \frac{\max(m,n)}{P}\right)$

Communication Costs Comparison		
Algorithm	Latency cost (L)	Bandwidth cost (W)
Krylov methods [4]	$O(k \log P)$	$O\left(k \min(m,n) \log P\right)$

Table 3.2: Critical path costs of Krylov methods. k is the number of iterations required for Krylov methods to converge to a desired accuracy. We assume a 1D-block row layout if $n < m$ (1D-block column if $n > m$) and replicate the $\min(m,n)$-dimensional vectors and partition the $\max(m,n)$-dimensional vectors.

operational, communication and storage costs of the classical and communication-avoiding algorithms under the 1D-block column and 1D-block row data layouts. Section 3.4 provides numerical and performance experiments which show that the communication-avoiding algorithms are numerically stable and attain speedups over the standard algorithms. Finally, we conclude in Section 3.6 and describe directions for future work.

3.1 Block Coordinate Descent and Conjugate Gradients

The regularized least-squares problem can be written as the following optimization problem:

$$\arg\min_{x \in \mathbb{R}^n} \frac{\lambda}{2}\|x\|_2^2 + \frac{1}{2m}\|Ax - y\|_2^2 \tag{3.1}$$

where $A \in \mathbb{R}^{m \times n}$ is the data matrix, whose rows are data points and columns are features, $y \in \mathbb{R}^m$ are the labels, $x \in \mathbb{R}^n$ are the weights, and $\lambda > 0$ is a regularization parameter. The unregularized ($\lambda = 0$) and regularized ($\lambda > 0$) least squares problems have been well-studied in literature from directly solving the normal equations to other matrix factorization approaches [41, 10] to Krylov [10, 20, 105] and block coordinate descent methods [11, 111, 106, 115, 122]. Table 3.1 and 3.2 summarize the critical path costs of the iterative methods just described.

We briefly summarize the difference between the BCD and BDCD algorithms, but defer the derivations to Section 3.2. The BCD algorithm solves the primal minimization problem (3.1), whereas, the BDCD algorithm solves the dual minimization problem:

$$\arg\min_{\alpha \in \mathbb{R}^m} \frac{\lambda}{2}\left\|\frac{1}{\lambda m}A^T\alpha\right\|_2^2 + \frac{1}{2m}\|\alpha - y\|_2^2 \tag{3.2}$$

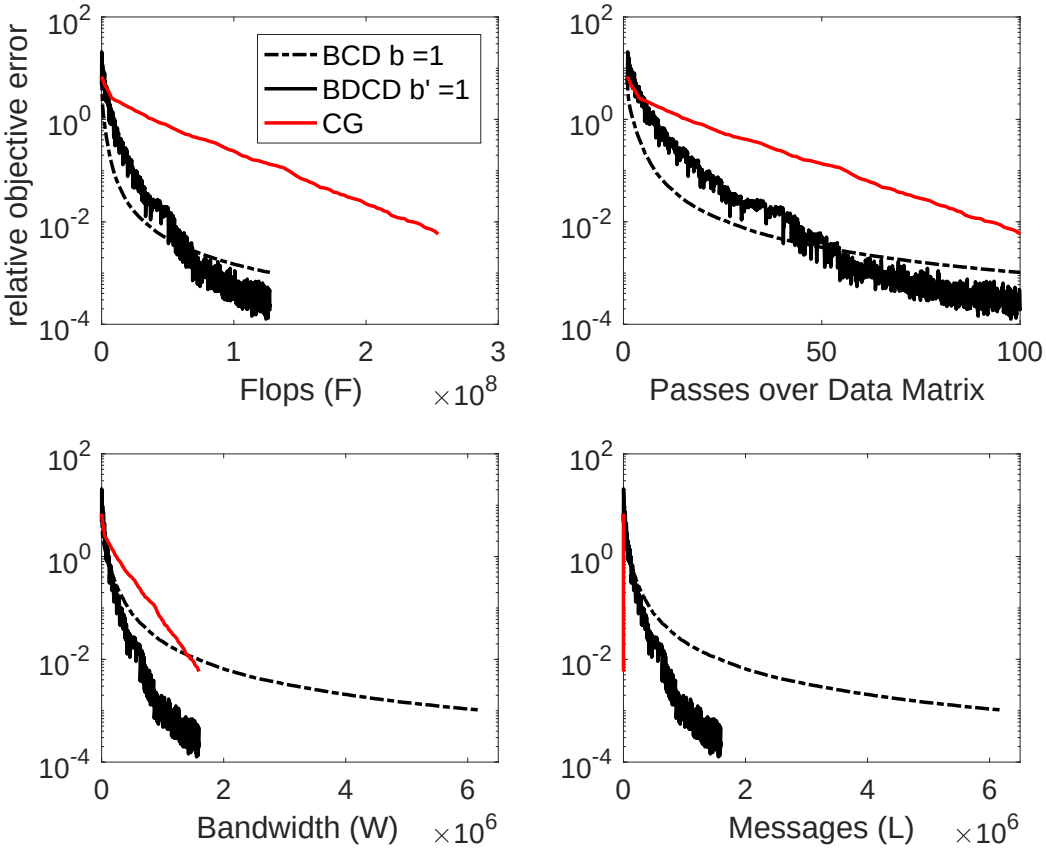


Figure 3.1: Comparison of convergence behavior against algorithm costs of Conjugate Gradients (CG), BCD (with $b = 1$) and BDCD (with $b' = 1$). Convergence is reported in terms of the relative objective error and the experiments are performed on the news20 dataset ($m = 15935$, $n = 62061$, $nnz(A) = 1272569$) obtained from LIBSVM [26]. We fix the number of CG iterations to $k = 100$, BCD iterations to $H = 100n$ and BDCD iterations to $H' = 100m$.

where $\alpha \in \mathbb{R}^m$ is the dual solution vector. The dual problem [106] can be obtained by deriving the convex conjugate of (3.1) and has the following primal-dual solution relationship:

$$x = \frac{1}{\lambda m} A^T \alpha. \tag{3.3}$$

Figure 3.1 illustrates the tradeoff between convergence behavior and algorithm costs of CG, BCD and BDCD. We plot the sequential flops cost and ignore the $\log P$ factor for latency. We allow each algorithm to make 100 passes over A and plot the relative objective error, $\frac{f(A,x_{opt},y)-f(A,x_{alg},y)}{f(A,x_{opt},y)}$, where $f(A,x,y) = \frac{1}{2m}\|A^Tx - y\|_2^2 + \frac{\lambda}{2}\|x\|_2^2$. x_{opt} is computed *a priori* from CG with a tolerance of 10^{-15}, and x_{alg} is the solution obtained from each iteration of CG, BCD or BDCD. Since A is not symmetric, CG requires two matrix-vector products at each iteration (one with A and another

Relative Objective Error Comparison			
CG iteration	CG error	BDCD error	BCD error
0	6.8735	6.8735	6.8735
1	4.5425	7.8231	1.2826
25	0.5115	0.0441	0.0104
50	0.1326	0.0043	0.0031
75	0.0283	5.0779e-04	0.0016
100	0.0058	1.9346e-04	0.0010

Table 3.3: Relative objective errors of CG, BDCD ($b' = 1$) and BCD ($b = 1$). We normalize the BDCD and BCD iterations to match reported CG iterations. If k is the CG iteration, then BCD performs $H = kn$ and BDCD performs $H' = km$ iterations.

with A^T). Therefore, the flops cost of CG is $2\times$ the cost of a matrix-vector product with A. We assume that the two matrix-vector products can be computed with a single pass over A.

If low-accuracy suffices, then BCD and BDCD converge faster in terms of flops and passes over A. However, CG is more bandwidth efficient than BCD (but not BDCD) and is *orders of magnitude* more latency efficient than BCD and BDCD. This suggests that reducing the latency cost of BCD and BDCD is an important step in making these algorithms competitive.

3.2 Communication-Avoiding Derivation

In this section, we re-derive the block coordinate descent (BCD) and block dual coordinate descent (BDCD) algorithms starting from the respective minimization problems. The derivation of BCD and BDCD lead to recurrences which can be unrolled to derive communication-avoiding versions of BCD and BDCD, which we will refer to as CA-BCD and CA-BDCD respectively.

Derivation of Block Coordinate Descent

The minimization problem in (3.1) can be solved by block coordinate descent with the b-dimensional update

$$x_h = x_{h-1} + \mathbb{I}_h \Delta x_h \tag{3.4}$$

where $x_h \in \mathbb{R}^n$ and $\mathbb{I}_h = \left[e_{i_1}, e_{i_2}, \ldots, e_{i_b}\right] \in \mathbb{R}^{n\times b}$, $\Delta x_h \in \mathbb{R}^b$, and $i_l \in [n]$ for $l = 1, 2, \ldots, b$. By substitution in (3.1) we obtain the minimization problem

$$\underset{\Delta x_h \in \mathbb{R}^b}{\arg\min} \frac{\lambda}{2}\|x_{h-1} + \mathbb{I}_h \Delta x_h\|_2^2 + \frac{1}{2m}\|Ax_{h-1} + A\mathbb{I}_h \Delta x_h - y\|_2^2$$

with the closed-form solution

$$\Delta x_h = \left(\frac{1}{m}\mathbb{I}_h^T A^T A \mathbb{I}_h + \lambda \mathbb{I}_h^T \mathbb{I}_h\right)^{-1} \left(-\lambda \mathbb{I}_h^T x_{h-1} - \frac{1}{m}\mathbb{I}_h^T A^T A x_{h-1} + \frac{1}{m}\mathbb{I}_h^T A^T y\right). \tag{3.5}$$

Algorithm 1 Block Coordinate Descent (BCD) Algorithm

1: **Input**: $A \in \mathbb{R}^{m\times n}, y \in \mathbb{R}^m$, $H > 1$, $x_0 \in \mathbb{R}^n$, $b \in \mathbb{Z}_+$ s.t. $b \leq n$
2: **for** $h = 1, 2, \cdots, H$ **do**
3: choose $\{i_l \in [n] | l = 1, 2, \ldots, b\}$ uniformly at random without replacement
4: $\mathbb{I}_h = [e_{i_1}, e_{i_2}, \cdots, e_{i_b}]$
5: $\Gamma_h = \frac{1}{m}\mathbb{I}_h^T A^T A \mathbb{I}_h + \lambda \mathbb{I}_h^T \mathbb{I}_h$
6: $\Delta x_h = \Gamma_h^{-1}\left(-\lambda \mathbb{I}_h^T x_{h-1} - \frac{1}{m}\mathbb{I}_h^T A^T z_{h-1} + \frac{1}{m}\mathbb{I}_h^T A^T y\right)$
7: $x_h = x_{h-1} + \mathbb{I}_h \Delta x_h$
8: $z_h = z_{h-1} + A\mathbb{I}_h \Delta x_h$
9: **Output** x_H

The closed-from solution appears to require a matrix-vector multiply using the entire data matrix to compute $\frac{1}{m}\mathbb{I}_h^T A^T A x_{h-1}$. However, this can be avoided by introducing the auxiliary variable, $z_h = Ax_h$, which, by substituting (3.4), can be re-arranged into a vector update of the form

$$z_h = Ax_{h-1} + A\mathbb{I}_h \Delta x_h = z_{h-1} + A\mathbb{I}_h \Delta x_h \tag{3.6}$$

and the closed-form solution can be written in terms of z_{h-1},

$$\Delta x_h = \left(\frac{1}{m}\mathbb{I}_h^T A^T A \mathbb{I}_h + \lambda \mathbb{I}_h^T \mathbb{I}_h\right)^{-1}\left(-\lambda \mathbb{I}_h^T x_{h-1} - \frac{1}{m}\mathbb{I}_h^T A^T z_{h-1} + \frac{1}{m}\mathbb{I}_h^T A^T y\right). \tag{3.7}$$

In order to make the communication-avoiding BCD derivation easier, let us define

$$\Gamma_h = \frac{1}{m}\mathbb{I}_h^T A^T A \mathbb{I}_h + \lambda \mathbb{I}_h^T \mathbb{I}_h.$$

Then (3.7) can be re-written as

$$\Delta x_h = \Gamma_h^{-1}\left(-\lambda \mathbb{I}_h^T x_{h-1} - \frac{1}{m}\mathbb{I}_h^T A^T z_{h-1} + \frac{1}{m}\mathbb{I}_h^T A^T y\right). \tag{3.8}$$

This re-arrangement leads to the Block Coordinate Descent (BCD) method shown in Algorithm 1. The recurrence in lines 6, 7, and 8 of Algorithm 1 allow us to unroll the BCD recurrences and avoid communication. We begin by changing the loop index from h to $sk + j$ where k is the outer loop index, s is the recurrence unrolling parameter and j is the inner loop index. Assume that we are at the beginning of iteration $sk + 1$ and x_{sk} and z_{sk} were just computed. Then Δx_{sk+1} can be computed by

$$\Delta x_{sk+1} = \Gamma_{sk+1}^{-1}\left(-\lambda \mathbb{I}_{sk+1}^T x_{sk} - \frac{1}{m}\mathbb{I}_{sk+1}^T A^T z_{sk} + \frac{1}{m}\mathbb{I}_{sk+1}^T A^T y\right).$$

Algorithm 2 Communication-Avoiding Block Coordinate Descent (CA-BCD) Algorithm

Input: $A \in \mathbb{R}^{m\times n}, y \in \mathbb{R}^m$, $H > 1$, $x_0 \in \mathbb{R}^n$, $b \in \mathbb{Z}_+$ s.t. $b \leq n$
for $k = 0, 1, \cdots, \frac{H}{s}$ **do**
  for $j = 1, 2, \cdots, s$ **do**
    choose $\{i_l \in [n] | l = 1, 2, \ldots, b\}$ uniformly at random without replacement
    $\mathbb{I}_{sk+j} = [e_{i_1}, e_{i_2}, \cdots, e_{i_b}]$
  let $Y = \left[\mathbb{I}_{sk+1}, \mathbb{I}_{sk+2}, \cdots, \mathbb{I}_{sk+s}\right]^T A^T$.
  compute the Gram matrix, $G = \frac{1}{m} YY^T + \lambda I$.
  for $j = 1, 2, \cdots, s$ **do**
    Γ_{sk+j} are the $b \times b$ diagonal blocks of G.
    $$\Delta x_{sk+j} = \Gamma_{sk+j}^{-1}\Bigg(- \lambda \mathbb{I}_{sk+j}^T x_{sk} - \lambda \textstyle\sum_{t=1}^{j-1} \left(\mathbb{I}_{sk+j}^T \mathbb{I}_{sk+t} \Delta x_{sk+t}\right) - \frac{1}{m}\mathbb{I}_{sk+j}^T A^T z_{sk} - \frac{1}{m}\textstyle\sum_{t=1}^{j-1}\left(\mathbb{I}_{sk+j}^T A^T A \mathbb{I}_{sk+t}\Delta x_{sk+t}\right) + \frac{1}{m}\mathbb{I}_{sk+j}^T A^T y\Bigg)$$
  $x_{sk+s} = x_{sk} + \sum_{t=1}^{s}\left(\mathbb{I}_{sk+t}\Delta x_{sk+t}\right)$
  $z_{sk+s} = z_{sk} + A\sum_{t=1}^{s}\left(\mathbb{I}_{sk+t}\Delta x_{sk+t}\right)$
Output x_H

By unrolling the recurrence for x_{sk+1} and z_{sk+1} we can compute Δx_{sk+2} in terms of x_{sk} and z_{sk}

$$\Delta x_{sk+2} = \Gamma_{sk+2}^{-1}\Bigg(- \lambda \mathbb{I}_{sk+2}^T x_{sk} - \lambda \mathbb{I}_{sk+2}^T \mathbb{I}_{sk+1}\Delta x_{sk+1} - \frac{1}{m}\mathbb{I}_{sk+2}^T A^T z_{sk} - \frac{1}{m}\mathbb{I}_{sk+2}^T A^T A \mathbb{I}_{sk+1}\Delta x_{sk+1} + \frac{1}{m}\mathbb{I}_{sk+2}^T A^T y\Bigg).$$

By induction we can show that Δx_{sk+j} can be computed using x_{sk} and z_{sk}

$$\Delta x_{sk+j} = \Gamma_{sk+j}^{-1}\Bigg(- \lambda \mathbb{I}_{sk+j}^T x_{sk} - \lambda \sum_{t=1}^{j-1}\left(\mathbb{I}_{sk+j}^T \mathbb{I}_{sk+t}\Delta x_{sk+t}\right) - \frac{1}{m}\mathbb{I}_{sk+j}^T A^T z_{sk} - \frac{1}{m}\sum_{t=1}^{j-1}\left(\mathbb{I}_{sk+j}^T A^T A \mathbb{I}_{sk+t}\Delta x_{sk+t}\right) + \frac{1}{m}\mathbb{I}_{sk+j}^T A^T y\Bigg). \quad (3.9)$$

for $j = 1, 2, \ldots, s$. Due to the recurrence unrolling we can defer the updates to x_{sk} and z_{sk} for s steps. Notice that the first summation in (3.9) computes the intersection between the coordinates chosen at iteration $sk + j$ and $sk + t$ for $t = 1, \ldots, j - 1$ via the product $\mathbb{I}_{sk+j}^T \mathbb{I}_{sk+t}$. Note that the intersection is not necessarily 0. For $b > 1$ we select coordinates without replacement only for this iteration. The next iteration is allowed to choose b coordinates (again without replacement) from all of the coordinates (i.e. not just the unchosen coordinates from the previous iteration). In general, when fusing s iterations the intersection is non-empty (in practice,

Algorithm 3 Block Dual Coordinate Descent (BDCD) Algorithm

Input: $A \in \mathbb{R}^{m\times n}, y \in \mathbb{R}^m$, $H' > 1$, $\alpha_0 \in \mathbb{R}^m$, $b' \in \mathbb{Z}_+$ s.t. $b' \leq m$
Initialize: $x_0 \leftarrow \frac{-1}{\lambda m} A^T \alpha_0$
for $h = 1, 2, \cdots, H'$ **do**
    choose $\{i_l \in [m] | l = 1, 2, \ldots, b'\}$ uniformly at random without replacement
    $\mathbb{I}_h = \left[e_{i_1}, e_{i_2}, \cdots, e_{i_{b'}}\right]$
    $\Theta_h = \frac{1}{\lambda m^2} \mathbb{I}_h^T A A^T \mathbb{I}_h + \frac{1}{m} \mathbb{I}_h^T \mathbb{I}_h$
    $\Delta\alpha_h = \frac{1}{m} \Theta_h^{-1} \left(-\mathbb{I}_h^T A x_{h-1} - \mathbb{I}_h^T \alpha_{h-1} + \mathbb{I}_h^T y\right)$
    $\alpha_h = \alpha_{h-1} + \mathbb{I}_h \Delta\alpha_h$
    $x_h = x_{h-1} + \frac{1}{\lambda m} A^T \mathbb{I}_h \Delta\alpha_h$
Output α'_H and x'_H

we have seen this in occur in our experiments). Communication can be avoided in this term by initializing all processors to the same seed for the random number generator. The second summation in (3.9) computes the Gram-like matrices $\mathbb{I}_{sk+j}^T A^T A \mathbb{I}_{sk+t}$ for $t = 1, \ldots, j-1$. Communication can be avoided in this computation by computing the $sb \times sb$ Gram matrix $G = \left(\frac{1}{m}\left[\mathbb{I}_{sk+1}, \mathbb{I}_{sk+2}, \cdots, \mathbb{I}_{sk+s}\right]^T A^T A \left[+\mathbb{I}_{sk+1}, \mathbb{I}_{sk+2}, \cdots, \mathbb{I}_{sk+s}\right] + \lambda I\right)$ once before the inner loop and redundantly storing it on all processors. Finally, we can perform the vector updates

$$x_{sk+s} = x_{sk} + \sum_{t=1}^{s} \left(\mathbb{I}_{sk+t} \Delta x_{sk+t}\right), \tag{3.10}$$

$$z_{sk+s} = z_{sk} + A \sum_{t=1}^{s} \left(\mathbb{I}_{sk+t} \Delta x_{sk+t}\right). \tag{3.11}$$

The resulting CA-BCD algorithm is shown in Alg. 2.

Derivation of Block Dual Coordinate Descent

The solution to the primal problem (3.1) can also be obtained by solving the dual minimization problem shown in (3.2) with the primal-dual relationship shown in (3.3). The dual problem (3.2) can be solved using block coordinate descent which iteratively solves a subproblem in $\mathbb{R}^{b'}$, where $1 \leq b' \leq m$ is a tunable block-size parameter. Let us first define the dual vector update for $\alpha_h \in \mathbb{R}^m$

$$\alpha_h = \alpha_{h-1} + \mathbb{I}_h \Delta\alpha_h. \tag{3.12}$$

Here h is the iteration index, $\mathbb{I}_h = \left[e_{i_1}, e_{i_2}, \ldots e_{i_{b'}}\right] \in \mathbb{R}^{m\times b'}$, $i_l \in [m]$ for $l = 1, 2, \ldots b'$ and $\Delta\alpha_h \in \mathbb{R}^{b'}$. By substitution in (3.2), $\Delta\alpha_h$ is the solution to a minimization problem in $\mathbb{R}^{b'}$ as

desired:

$$\arg\min_{\Delta\alpha_h \in \mathbb{R}^{b'}} \frac{1}{2\lambda m^2} \|A\alpha_{h-1} + A\mathbb{I}_h\Delta\alpha_h\|_2^2 + \frac{1}{2m}\|\alpha_{h-1} + \mathbb{I}_h\Delta\alpha_h - y\|_2^2. \tag{3.13}$$

Finally, due to (3.3) we obtain the primal vector update for $x_h \in \mathbb{R}^n$

$$x_h = x_{h-1} + \frac{1}{\lambda m}A^T\mathbb{I}_h\Delta\alpha_h. \tag{3.14}$$

From (3.12), (3.13), and (3.14) we obtain a block coordinate descent algorithm which solves the dual minimization problem. Henceforth, we refer to this algorithm as block dual coordinate descent (BDCD). Note that by setting $b' = 1$ we obtain the SDCA algorithm [106] with the least-squares loss function.

The optimization problem (3.13) which computes the solution along the chosen coordinates has the closed-form

$$\Delta\alpha_h = \left(\frac{1}{\lambda m^2}\mathbb{I}_h^T AA^T\mathbb{I}_h + \frac{1}{m}\mathbb{I}_h^T\mathbb{I}_h\right)^{-1}\left(\frac{-1}{\lambda m^2}\mathbb{I}_h^T AA^T\alpha_{h-1} - \frac{1}{m}\mathbb{I}_h^T\alpha_{h-1} + \frac{1}{m}\mathbb{I}_h^T y\right). \tag{3.15}$$

Let us define $\Theta_h \in \mathbb{R}^{b'\times b'}$ such that

$$\Theta_h = \left(\frac{1}{\lambda m^2}\mathbb{I}_h^T AA^T\mathbb{I}_h + \frac{1}{m}\mathbb{I}_h^T\mathbb{I}_h\right).$$

From this we have that at iteration h, we compute the solution along the b' coordinates of the linear system

$$\Delta\alpha_h = \frac{1}{m}\Theta_h^{-1}\left(-\mathbb{I}_h^T Ax_{h-1} - \mathbb{I}_h^T\alpha_{h-1} + \mathbb{I}_h^T y\right) \tag{3.16}$$

and obtain the BDCD algorithm shown in Algorithm 3. The recurrence in lines 7, 8, and 9 of Algorithm 3 allow us to unroll the BDCD recurrences and avoid communication. We begin by changing the loop index from h to $sk + j$ where k is the outer loop index, s is the recurrence unrolling parameter and j is the inner loop index. Assume that we are at the beginning of iteration $sk + 1$ and x_{sk} and α_{sk} were just computed. Then $\Delta\alpha_{sk+1}$ can be computed by

$$\Delta\alpha_{sk+1} = -\frac{1}{m}\Theta_{sk+1}^{-1}\left(-\mathbb{I}_{sk+1}^T Ax_{sk} - \mathbb{I}_{sk+1}^T\alpha_{sk} + \mathbb{I}_{sk+1}^T y\right).$$

Furthermore, by unrolling the recurrences for x_{sk+1} and α_{sk+1} we can analogously to (3.9) show by induction that

$$\Delta\alpha_{sk+j} = \frac{1}{m}\Theta_{sk+j}^{-1}\Bigg(-\mathbb{I}_{sk+j}^T Ax_{sk} - \frac{1}{\lambda m}\sum_{t=1}^{j-1}\left(\mathbb{I}_{sk+j}^T AA^T\mathbb{I}_{sk+t}\Delta\alpha_{sk+t}\right) - \mathbb{I}_{sk+j}^T\alpha_{sk} - \sum_{t=1}^{j-1}\left(\mathbb{I}_{sk+j}^T\mathbb{I}_{sk+t}\Delta\alpha_{sk+t}\right) + \mathbb{I}_{sk+j}^T y\Bigg) \tag{3.17}$$

Algorithm 4 Communication-Avoiding Block Dual Coordinate Descent (CA-BDCD) Algorithm

Input: $A \in \mathbb{R}^{m\times n}, y \in \mathbb{R}^m$, $H' > 1$, $\alpha_0 \in \mathbb{R}^m$, $b' \in \mathbb{Z}_+$ s.t. $b' \leq m$
Initialize: $x_0 \leftarrow \frac{-1}{\lambda m} A^T \alpha_0$
for $k = 0, 1, \cdots, \frac{H'}{s}$ **do**
  for $j = 1, 2, \cdots, s$ **do**
    choose $\{i_l \in [m] | l = 1, 2, \ldots, b'\}$ uniformly at random without replacement
    $\mathbb{I}_{sk+j} = \left[e_{i_1}, e_{i_2}, \cdots, e_{i_{b'}}\right]$
  let $Y = A\left[\mathbb{I}_{sk+1}, \mathbb{I}_{sk+2}, \ldots, \mathbb{I}_{sk+s}\right]$.
  compute the Gram matrix, $G' = \frac{1}{\lambda m^2} Y^T Y + \frac{1}{m} I$.
  for $j = 1, 2, \cdots, s$ **do**
    Θ_{sk+j} are the $b' \times b'$ diagonal blocks of G'.
    $$\Delta\alpha_{sk+j} = \frac{1}{m}\Theta_{sk+j}^{-1}\Bigg(- \mathbb{I}_{sk+j}^T A x_{sk} - \frac{1}{\lambda m}\sum_{t=1}^{j-1}\left(\mathbb{I}_{sk+j}^T A A^T \mathbb{I}_{sk+t}\Delta\alpha_{sk+t}\right) - \mathbb{I}_{sk+j}^T \alpha_{sk} - \sum_{t=1}^{j-1}\left(\mathbb{I}_{sk+j}^T \mathbb{I}_{sk+t}\Delta\alpha_{sk+t}\right) + \mathbb{I}_{sk+j}^T y\Bigg)$$
  $x_{sk+s} = x_{sk} + \frac{1}{\lambda m} A^T \sum_{t=1}^{s}\left(\mathbb{I}_{sk+t}\Delta\alpha_{sk+t}\right)$
  $\alpha_{sk+s} = \alpha_{sk} + \sum_{t=1}^{s}\left(\mathbb{I}_{sk+t}\Delta\alpha_{sk+t}\right)$
Output $\alpha_{H'}$ and $x_{H'}$

for $j = 1, 2, \ldots, s$. Note that due to unrolling the recurrence we can compute $\Delta\alpha_{sk+j}$ from x_{sk} and α_{sk} which are the primal and dual solution vectors from the previous outer iteration. Since the solution vector updates require communication, the recurrence unrolling allows us to defer those updates for s iterations at the expense of additional computation. The solution vectors can be updated by

$$x_{sk+s} = x_{sk} + \frac{1}{\lambda m} A^T \sum_{t=1}^{s}\left(\mathbb{I}_{sk+t}\Delta\alpha_{sk+t}\right), \tag{3.18}$$

$$\alpha_{sk+s} = \alpha_{sk} + \sum_{t=1}^{s}\left(\mathbb{I}_{sk+t}\Delta\alpha_{sk+t}\right). \tag{3.19}$$

The resulting CA-BDCD algorithm is shown in Alg. 4.

3.3 Algorithm Analysis

From the derivations in Section 3.2, we can observe that BCD and BDCD perform computations on $A^T A$ and AA^T, respectively. This implies that, along with the convergence rates, the shape of A is a key factor in choosing between the two methods. Furthermore, the data partitioning scheme used to distribute A between processors may cause one method to have a lower communication cost

than the other. In this section we analyze the cost of BCD and BDCD under two data partitioning schemes: 1D-block column (feature partitioning) and 1D-block row (data point partitioning). In both cases, we derive the associated operational, storage, and communication costs. We perform a similar analysis of the CA-variants to illustrate that we provably avoid communication and describe tradeoffs. Since A is sparse the analysis of the computational cost includes passes over the sparse data structure instead of just the floating-point operations associated with the sparse matrix - sparse matrix multiplication (i.e. Gram matrix computation). Therefore, our analysis gives bounds on the local operations for each processor. We begin in Section 3.3 with the analysis of the BCD and BDCD algorithms and then analyze our new, communication-avoiding variants in Section 3.3.

Classical Algorithms

We begin with the analysis of the BCD algorithm with A stored in a 1D-block row layout and show how to extend this proof to BDCD with A in a 1D-block column layout.

Theorem 3.3.1. *H iterations of the Block Coordinate Descent (BCD) algorithm with the matrix $A \in \mathbb{R}^{m\times n}$ stored in 1D-block row partitions with a block size b, on P processors along the critical path costs*

$$F = O\left(\frac{Hb^2fm}{P} + Hb^3\right)\ ops,\ M = O\left(\frac{fmn+m}{P} + b^2 + n\right)\ words\ of\ memory.$$

Communication costs

$$W = O\left(Hb^2 \log P\right)\ words\ moved,\ L = O\left(H \log P\right)\ messages.$$

Proof. The BCD algorithm computes a $b \times b$ Gram matrix, Γ_h, solves a $b \times b$ linear system to obtain Δx_h, and updates the vectors x_h and z_h. Computing the Gram matrix requires that each processor locally compute a $b \times b$ block of inner-products and then perform an all-reduce (a reduction and broadcast) to sum the partial blocks. Since the $b \times m$ sub-matrix $\mathbb{I}_h^T A^T$ has bfm non-zeros, the parallel Gram matrix computation ($\mathbb{I}_h^T A^T A \mathbb{I}_h$) requires $O(\frac{b^2fm}{P})$ operations (there are b^2 elements of the Gram matrix each of which depend on fm non-zeros) and communicates $O\left(b^2 \log P\right)$ words, with $O\left(\log P\right)$ messages. In order to solve the subproblem redundantly on all processors, a local copy of the residual is required. Computing the residual requires $O\left(\frac{bfm}{P}\right)$ operations, and communicates $O\left(b \log P\right)$ words, in $O\left(\log P\right)$ messages. Once the residual is computed the subproblem can be solved redundantly on each processor in $O\left(b^3\right)$ flops. Finally, the vector updates to x_h and z_h can be computed without any communication in $O\left(b + \frac{bfm}{P}\right)$ flops on each processor. The critical path costs of H iterations of this algorithm are $O\left(\frac{Hb^2fm}{P} + Hb^3\right)$ flops, $O\left(Hb^2 \log P\right)$ words, and $O\left(H \log P\right)$ messages. Each processor requires enough memory to store x_h, Γ_h, Δx, $\mathbb{I}_h$ and $\frac{1}{P}$-th of A, z_h, and y. Therefore the memory cost of each processor is $n + b^2 + 2b + \frac{fmn+2m}{P} = O\left(\frac{fmn+m}{P} + b^2 + n\right)$ words per processor. □

If $\frac{fm}{P} > b$, then computing the Gram matrix dominates solving the subproblem. Furthermore, the storage cost of A dominates the cost of the Gram matrix.

Theorem 3.3.2. *H' iterations of the Block Dual Coordinate Descent (BDCD) algorithm with the matrix $A \in \mathbb{R}^{m\times n}$ stored in 1D-block column partitions with a block size b', on P processors along the critical path costs*

$$F = O\left(\frac{H'b'^2fn}{P} + H'b'^3\right)\ ops,\ M = O\left(\frac{fmn+n}{P} + b'^2 + m\right)\ words\ of\ memory.$$

Communication costs

$$W = O\left(H'b'^2 \log P\right)\ words\ moved,\ L = O\left(H' \log P\right)\ messages.$$

Proof. Proof is similar to that of Theorem 3.3.1 with appropriate change of variables for BDCD. □

If A is stored in a 1D-block column layout, then each processor stores a disjoint subset of the features of A. Since BCD selects b features at each iteration, 1D-block column partitioning could lead to load imbalance. In order to avoid load imbalance we re-partition the chosen b features into 1D-block row layout and proceed by using the 1D-block row BCD algorithm. Re-partitioning the b features requires communication, so we begin by bounding the maximum number of features assigned to a single processor. The bandwidth cost of re-partitioning is bounded by the processor with maximum load (i.e. maximum number of features). These bounds only holds with high probability since the features are chosen uniformly at random. To attain bounds on the bandwidth cost we assume that each sampled row of A has fn non-zeros.

Lemma 3.3.3. *Given a matrix $A \in \mathbb{R}^{m\times n}$ and P processors such that each processor stores $\Theta\left(\lfloor \frac{n}{P} \rfloor\right)$ features, if b features are chosen uniformly at random, then the worst case maximum number of features, $\eta(b,P)$, assigned to a single processor w.h.p. is:*

$$\eta(b,P) = \begin{cases} O\left(\frac{b}{P} + \sqrt{\frac{b\log P}{P}}\right) & \text{if } b > P\log P, \\ O\left(\frac{\log b}{\log\log b}\right) & \text{if } b = P, \\ O\left(\frac{\log P}{\log \frac{P}{b}}\right) & \text{if } b < \frac{P}{\log P}. \end{cases}$$

Proof. This is the well-known generalization of the balls and bins problem introduced by Gonnet [58] and extended by Mitzenmacher [84] and Raab et. al. [99]. □

A similar result holds for BDCD with A stored in a 1D-block row layout.

Theorem 3.3.4. *H iterations of the Block Coordinate Descent (BCD) algorithm with the matrix $A \in \mathbb{R}^{m\times n}$ stored in 1D-block column partitions with a block size b, on P processors along the critical path costs*

$$F = O\left(\frac{Hb^2fm}{P} + Hb^3\right)\ ops,\ M = O\left(\frac{fmn+m}{P} + b^2 + n\right)\ words\ of\ memory.$$

For small messages, communication costs w.h.p.

$$W = O\left(\left(b^2 + \eta(b,P)fm\right) H \log P\right) \ words\ moved,\ L = O\left(H \log P\right)\ messages.$$

For large messages, communication costs w.h.p.

$$W = O\left(Hb^2 \log P + H\eta(b,P)fm\right) \ words\ moved,\ L = O\left(HP\right)\ messages.$$

Proof. The 1D-block row partitioning scheme implies that the $b \times b$ Gram matrix, Γ_h, computation may be load imbalanced. Since we randomly select b columns, some processors may hold multiple columns while others hold none. In order to balance the computational load we perform an all-to-all to convert the $m \times b$ sampled matrix into the 1D-block row layout. The amount of data moved is bounded by the max-loaded processor, which from Lemma 3.3.3, stores $O\left(\eta(b,P)\right)$ rows w.h.p. in the worst-case. This requires $W = O\left(\eta(b,P)fm \log P\right)$ and $L = O\left(\log P\right)$ for small messages or $W = O\left(\eta(b,P)fn\right)$ and $L = O\left(HP\right)$ for large messages. The all-to-all requires additional storage on each processor of $M = O\left(\frac{bfn}{P}\right)$ words. Once the sampled matrix is converted, the BCD algorithm proceeds as in Theorem 3.3.1. By combining the cost of the all-to-all over H iterations and the costs from Theorem 3.3.1, we obtain the costs for the BCD algorithm with A stored in a 1D-block row layout. □

The additional storage for the all-to-all does not dominate since $b < n$ by definition.

Theorem 3.3.5. *H' iterations of the Block Dual Coordinate Descent (BDCD) algorithm with the matrix $A \in \mathbb{R}^{m \times n}$ stored in 1D-block row partitions with a block size b', on P processors along the critical path costs w.h.p.*

$$F = O\left(\frac{H'b'^2 fn}{P} + H'b'^3\right) \ ops,\ M = O\left(\frac{fmn + n}{P} + b'^2 + m\right) \ words\ of\ memory.$$

For small messages, communciation costs w.h.p.

$$W = O\left(\left(b'^2 + \eta(b',P)fn\right) H' \log P\right) \ words\ moved,\ L = O\left(H' \log P\right)\ messages.$$

For large messages, communication costs w.h.p.

$$W = O\left(H'b'^2 \log P + H'\eta(b',P)fn\right) \ words\ moved,\ L = O\left(H'P\right)\ messages.$$

Proof. Cost analysis similar to Thm. 3.3.4 proves this theorem. □

Communication-Avoiding Algorithms

In this section, we derive the computation, storage, and communication costs of our communication-avoiding BCD and BDCD algorithm under the 1D-block row and 1D-block column data layouts. In both cases we show that our algorithm reduces the latency costs by a factor of s, but increase flops and bandwidth by the same factor. Our experimental results will show that this tradeoff can lead to speedups. We begin with the CA-BCD algorithm in 1D-block row layout and, then show how this proof extends to CA-BDCD in 1D-block column layout.

Theorem 3.3.6. *H iterations of the Communication-Avoiding Block Coordinate Descent (CA-BCD) algorithm with the matrix $A \in \mathbb{R}^{m\times n}$ stored in 1D-block row partitions with a block size b, on P processors along the critical path costs*

$$F = O\left(\frac{Hb^2sfm}{P} + Hb^3\right)\ ops,\ M = O\left(\frac{fmn+m}{P} + b^2s^2 + n\right)\ words\ of\ memory.$$

Communication costs

$$W = O\left(Hb^2s\log P\right)\ words\ moved,\ L = O\left(\frac{H}{s}\log P\right)\ messages.$$

Proof. The CA-BCD algorithm computes the $sb \times sb$ Gram matrix, $G = \frac{1}{m}YY^T + \lambda I$, where $Y = \left[\mathbb{I}_{sk+1}, \mathbb{I}_{sk+2}, \cdots, \mathbb{I}_{sk+s}\right]^T A^T$, solves s ($b \times b$) linear systems to compute Δx_{sk+j} and updates the vectors x_{sk+s} and z_{sk+s}. Computing the Gram matrix requires that each processor locally compute a $sb \times sb$ block of inner-products and then perform an all-reduce (a reduction and broadcast) to sum the partial blocks. This operation requires $O\left(\frac{b^2s^2fm}{P}\right)$ operations (there are s^2b^2 elements of the Gram matrix each of which depends on fm non-zeros), communicates $O\left(s^2b^2\log P\right)$ words, and requires $O\left(\log P\right)$ messages. In order to solve the subproblem redundantly on all processors, a local copy of the residual is required. Computing the residual requires $O\left(\frac{bsfm}{P}\right)$ flops, and communicates $O\left(sb\log P\right)$ words, in $O\left(\log P\right)$ messages. Once the residual is computed the subproblem can be solved redundantly on each processor in $O\left(b^3s + b^2s^2\right)$ flops. Finally, the vector updates to x_{sk+s} and z_{sk+s} can be computed without any communication in $O\left(bs + \frac{bsfm}{P}\right)$ flops on each processor. Since the critical path occurs every $\frac{H}{s}$ iterations (every outer iteration), the algorithm costs $O\left(\frac{Hb^2sfm}{P} + Hb^3\right)$ flops, $O\left(Hb^2s\log P\right)$ words, and $O\left(\frac{H}{s}\log P\right)$ messages. Each processor requires enough memory to store x_{sk+j}, G, Δx_{sk+j}, $\mathbb{I}_{sk+j}$ and $\frac{1}{P}$-th of A, z_{sk+j}, and y. Therefore the memory cost of each processor is $n + s^2b^2 + 2sb + \frac{fmn+2m}{P} = O\left(\frac{fmn+m}{P} + b^2s^2 + n\right)$ words per processor. □

Theorem 3.3.7. *H' iterations of the Communication-Avoiding Block Dual Coordinate Descent (CA-BDCD) algorithm with the matrix $A \in \mathbb{R}^{m\times n}$ stored in 1D-block column partitions with a block size b', on P processors along the critical path costs*

$$F = O\left(\frac{H'b'^2sfn}{P} + H'b'^3\right)\ ops,\ M = O\left(\frac{fmn+n}{P} + b'^2s^2 + m\right)\ words\ of\ mem.$$

Communication costs

$$W = O\left(H'b'^2s\log P\right)\ words\ moved,\ L = O\left(\frac{H'}{s}\log P\right)\ messages.$$

Proof. Proof is similar to that of Theorem 3.3.6 with appropriate change of variables for CA-BDCD. □

Now we analyze the operational and communication costs of CA-variants of the 1D-block column BCD and 1D-block row BDCD algorithms.

Theorem 3.3.8. *H iterations of the Communication-Avoiding Block Coordinate Descent (CA-BCD) algorithm with the matrix $A \in \mathbb{R}^{m\times n}$ stored in 1D-block column partitions with a block size b, on P processors along the critical path costs*

$$F = O\left(\frac{Hb^2sfm}{P} + Hb^3\right)\ ops,\ M = O\left(\frac{(n+bs)fm+m}{P} + b^2s^2 + n\right) words.$$

For small messages, communication costs w.h.p.

$$W = O\left(\left(b^2s + \eta(sb,P)fm\right)H\log P\right)\ words\ moved, L = O\left(\frac{H}{s}\log P\right)\ messages.$$

For large messages, communication costs w.h.p.

$$W = O\left(Hb^2s\log P + H\eta(sb,P)fm\right)\ words\ moved,\ L = O\left(\frac{H}{s}P\right)\ messages.$$

Proof. The 1D-block column partitioning scheme implies that the $sb \times sb$ Gram matrix computation may be load imbalanced. Since we randomly select sb columns, some processors may hold multiple chosen columns while some hold none. In order to balance the computational load we perform an all-to-all to convert the $m \times sb$ sampled matrix into the 1D-block row layout. The amount of data moved is bounded by the max-loaded processor, which from Lemma 3.3.3, stores $O\left(\eta(sb,P)\right)$ rows w.h.p. in the worst-case. This requires $W = O\left(\eta(sb,P)fm\log P\right)$ and $L = O\left(\log P\right)$ for small messages or $W = O\left(\eta(sb,P)fm\right)$ and $L = O\left(HP\right)$ for large messages. The all-to-all requires additional storage on each processor of $M = O\left(\frac{bsfm}{P}\right)$ words. Once the sampled matrix is converted, the BCD algorithm proceeds as in Theorem 3.3.6. By combining the cost of the all-to-all over H iterations and the costs from Theorem 3.3.6, we obtain the costs for the CA-BCD algorithm with A stored in a 1D-block columns layout. □

Note that the additional storage for the all-to-all may dominate if $n < bs$. Therefore, b and s must be chosen carefully.

Theorem 3.3.9. *H iterations of the Communication-Avoiding Block Dual Coordinate Descent (CA-BDCD) algorithm with the matrix $A \in \mathbb{R}^{m\times n}$ stored in 1D-block row partitions with a block size b', on P processors along the critical path costs*

$$F = O\left(\frac{H'b'^2sfn}{P} + H'b'^3\right)\ ops, M = O\left(\frac{(m+b's)fn+n}{P} + b'^2s^2 + m\right)\ words.$$

For small messages, communication costs w.h.p.

$$W = O\left(\left(b'^2s + \eta(sb',P)fn\right)H'\log P\right)\ words\ moved, L = O\left(\frac{H'}{s}\log P\right)\ msgs.$$

Summary of datasets						
Name	Data Points (m)	Features (n)	f	σ_{min}	σ_{max}	Source
news20	$15,935$	$62,061$	0.0013	$1.7e{-6}$	$6.0e{+5}$	LIBSVM [77]
a9a	$32,561$	123	0.11	$4.9e{-6}$	$2.0e{+5}$	UCI [80]
real-sim	$72,309$	$20,958$	0.0024	$1.1e{-3}$	$9.2e{+2}$	LIBSVM [83]

Table 3.4: Properties of the LIBSVM datasets used in our experiments. We report the largest and smallest singular values (same as the eigenvalues) of A^TA.

For large messages, communication costs w.h.p.

$$W = O\left(H'b'^2 s \log P + H'\eta(sb', P)fn\right) \ words\ moved,\ L = O\left(\frac{H'}{s}P\right) \ messages.$$

Proof. A similar cost analysis to Theorem 3.3.8 proves this theorem. □

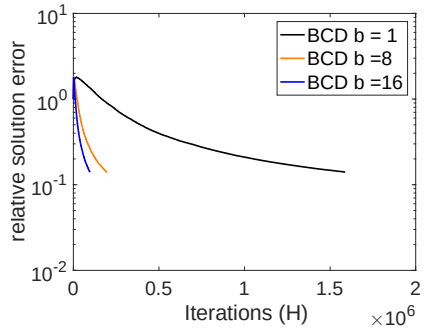
The communication-avoiding variants that we have derived require a factor of s fewer messages than their classical counterparts, at the cost of more computation, bandwidth and memory. This suggests that s must be chosen carefully to balance the additional costs with the reduction in the latency cost.

3.4 Convergence Behavior

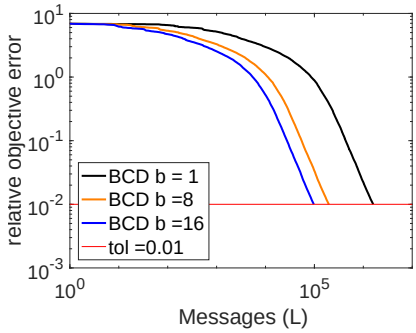
We proved in Section 3.3 that the CA-BCD and CA-BDCD algorithms reduce latency (the dominant cost) at the expense of additional bandwidth and computation. The recurrence unrolling we propose may also affect the numerical stability of CA-BCD and CA-BDCD since the sequence of computations and vector updates are different. In Section 3.4 we experimentally show that the communication-avoiding variants are numerically stable (in contrast to some CA-Krylov methods [20, 21, 22, 24, 23, 67]) and, in Section 3.5, we show that the communication-avoiding variants can lead to large speedups on a Cray XC30 supercomputer using MPI.

Numerical Experiments

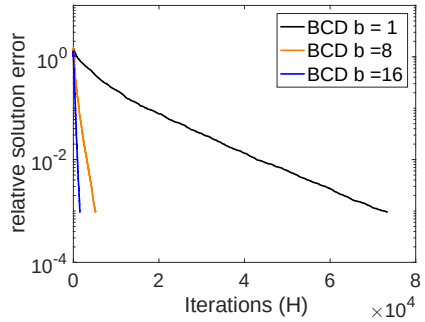
The algorithm transformations derived in Section 3.2 require that the CA-BCD and CA-BDCD operate on Gram matrices of size $sb \times sb$ instead of size $b \times b$ every outer iteration. Due to the larger dimensions, the condition number of the Gram matrix increases and may have an adverse affect on the convergence behavior. We explore this tradeoff between convergence behavior, flops, communication and the choices of b and s for the standard and communication-avoiding algorithms. All numerical stability experiments were performed in MATLAB version R2016b on a 2.3 GHz Intel i7 machine with 8GB of RAM with datasets obtained from the LIBSVM repository [26]. Datasets



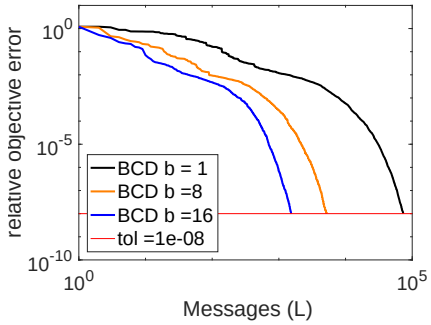
(a) news20



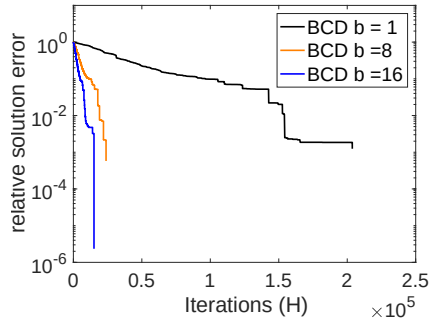
(b) news20



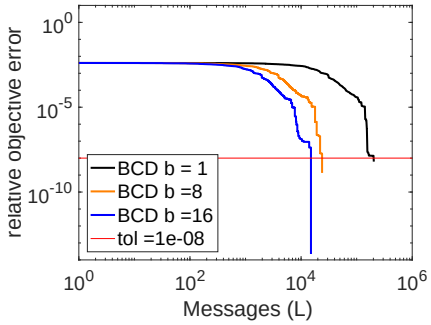
(c) a9a



(d) a9a



(e) real-sim



(f) real-sim

Figure 3.2: Convergence behavior of BCD for several block sizes, b, such that $1 \leq b < m$ on several machine learning datasets. We plot relative solution error (top row, Figs. 3.2a-3.2e) and relative objective error (bottom row, Fig. 3.2b-3.2f) with $\lambda = 1000\sigma_{min}$. We fix the relative objective error tolerance for news20 to $1e{-}2$ and $1e{-}8$ for a9a and real-sim. Note that the X-axis for Figures 3.2b-3.2f is equivalent to the number of iterations (modulo $\log_{10}$ scale).

were chosen so that all algorithms were tested on a range of shapes, sizes, and condition numbers. Table 3.4 summarizes the important properties of the datasets tested. For all experiments, we set the regularization parameter to $\lambda = 1000\sigma_{min}$. The regularization parameter reduces the

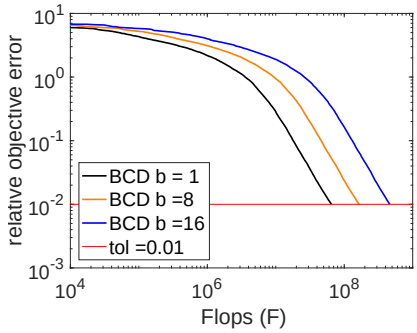
condition numbers of the datasets and allows the BCD and BDCD algorithms to converge faster. In practice, λ should be chosen based on metrics like prediction accuracy on the test data (or hold-out data). Smaller values of λ would slow the convergence rate and require more iterations, therefore we choose λ so that our experiments have reasonable running times. We do not explore tradeoffs among λ values, convergence rate and running times in this paper. In order to measure convergence behavior, we plot the relative solution error, $\frac{\|x_{opt}-x_h\|_2}{\|x_{opt}\|_2}$, where x_h is the solution obtained from the coordinate descent algorithms at iteration h and x_{opt} is obtained from conjugate gradients with $tol = 1e{-}15$. We also plot the relative objective error, $\frac{f(A,x_{opt},y)-f(A,x_h,y)}{f(A,x_{opt},y)}$, where $f(A,x,y) = \frac{1}{2m}\|Ax-y\|_2^2 + \frac{\lambda}{2}\|x\|_2^2$, the primal objective. We use the primal objective to show convergence behavior for BCD, BDCD and their communication-avoiding variants. We explore the tradeoff between the block sizes, b and b', and convergence behavior to test BCD and BDCD stability due to the choice of block sizes. Then, we fix the block sizes and explore the tradeoff between s, the recurrence unrolling parameter, and convergence behavior to study the stability of the communication-avoiding variants. Finally, for both sets of experiments we also plot the algorithm costs against convergence behavior to illustrate the theoretical performance tradeoffs due to choice of block sizes and choice of s. For the latter experiments we assume that the datasets are partitioned in 1D-block column for BCD and 1D-block row for BDCD. We plot the sequential flops cost for all algorithms, ignore the $\log P$ factor for the number of messages and ignore constants. We obtain the Gram matrix computation cost from the SuiteSparse [37] routine `ssmultsym`[1].

Block Coordinate Descent

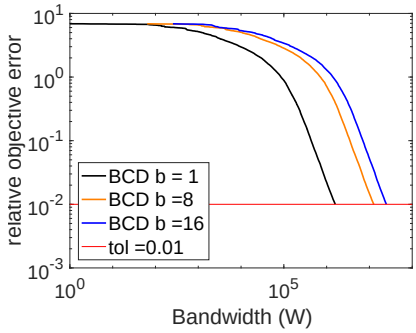
Recall that the BCD algorithm computes a $b \times b$ Gram matrix and solves a b-dimensional subproblem at each iteration. Therefore, one should expect that as b increases the algorithm converges faster but requires more flops and bandwidth per iteration. So we begin by exploring the block size vs. convergence behavior tradeoff for BCD with $1 \leq b < n$.

Figure 3.2 shows the convergence behavior of the datasets in Table 3.4 in terms of the relative solution error (Figs. 3.2a-3.2e) and relative objective error (Figs. 3.2b-3.2f). The x-axis for the latter figures are on $\log_{10}$ scale. Note that the number of messages is equivalent to the number of iterations, since BCD communicates every iteration. We observe that the convergence rates for all datasets improve as the block sizes increase.

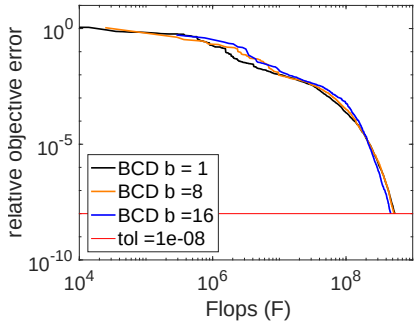
Figure 3.3 shows the convergence behavior (in terms of the objective error) vs. flops and bandwidth costs for each dataset. From these results, we observe that BCD with $b = 1$ is more flops and bandwidth efficient, whereas $b > 1$ is more latency efficient (from Figs. 3.2b-3.2f). This indicates the existence of a tradeoff between BCD convergence rate (which depends on the block size) and hardware-specific parameters (like flops rate, memory/network bandwidth and latency).



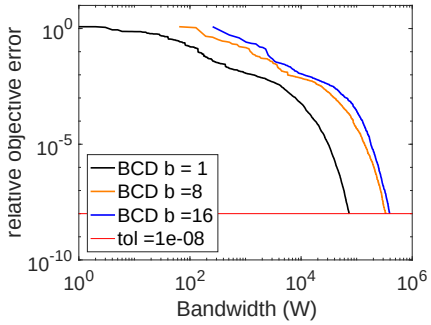
(a) news20



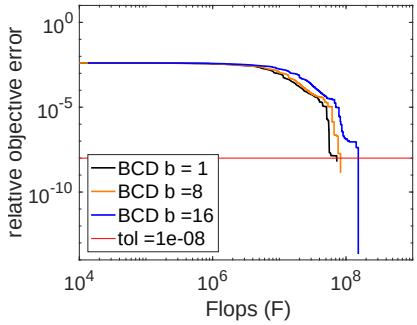
(b) news20



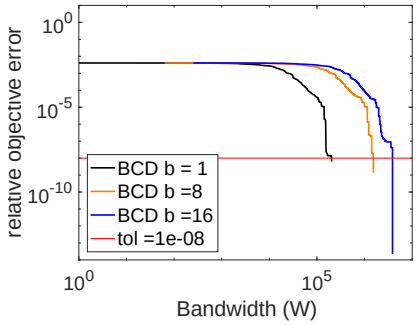
(c) a9a



(d) a9a



(e) real-sim



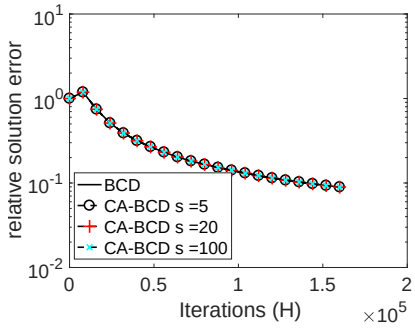
(f) real-sim

Figure 3.3: Convergence behavior of BCD for several block sizes, b, such that $1 \leq b < n$. We plot flops cost and bandwidth cost versus convergence with $\lambda = 1000\sigma_{min}$.

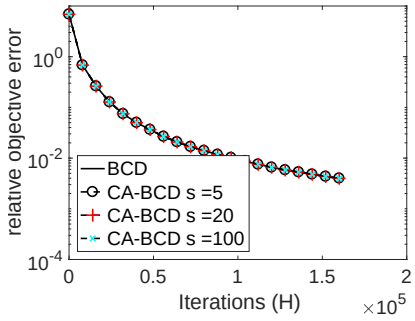
Communication-Avoiding Block Coordinate Descent

Our derivation of the CA-BCD algorithm showed that by unrolling the vector update recurrences we can reduce the latency cost of the BCD algorithm by a factor of s. However, this comes at the

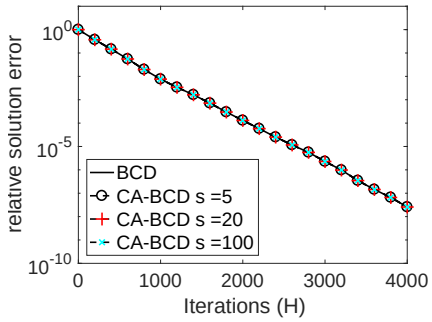
[1]Symbolically executes the sparse matrix - sparse matrix multiplication and reports an estimate of the flops cost (counting multiplications and additions).



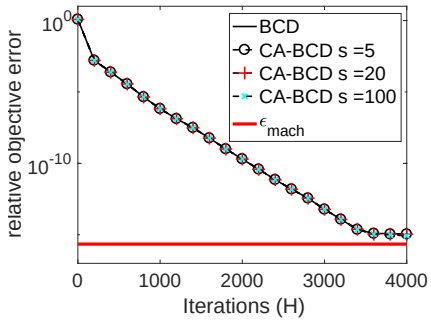
(a) news20



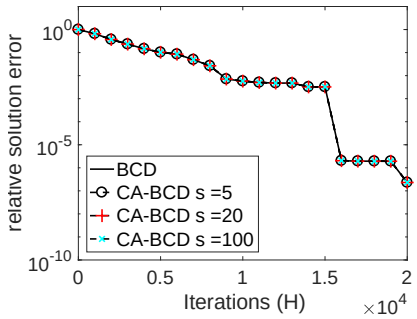
(b) news20



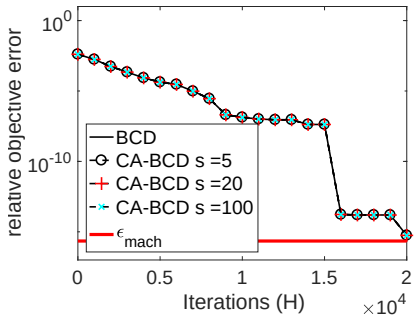
(c) a9a



(d) a9a



(e) real-sim

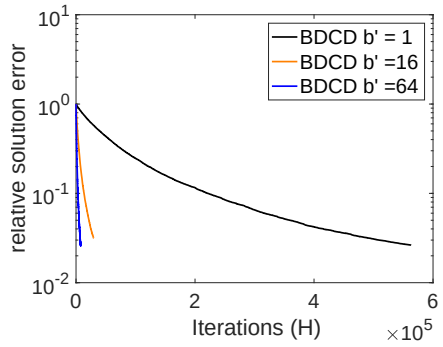


(f) real-sim

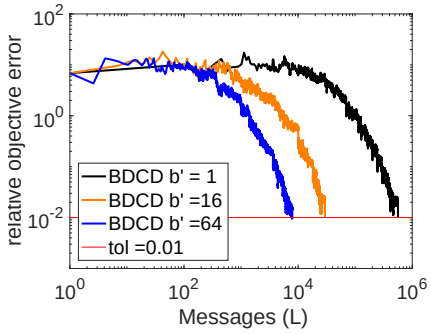
Figure 3.4: Convergence behavior of BCD and CA-BCD with several values of s. We plot relative solution error and relative objective error . The block size for each dataset is set to $b = 16$.

cost of computing a larger $sb \times sb$ Gram matrix whose condition number is larger than the $b \times b$ Gram matrix computed in the BCD algorithm. The larger condition number implies that the CA-BCD algorithm may not be stable for $s > 1$ due to round-off error. We begin by experimentally showing the convergence behavior of the CA-BCD algorithm on the datasets in Table 3.4 with fixed block sizes of $b = 16$ for news20, a9a, and real-sim, respectively.

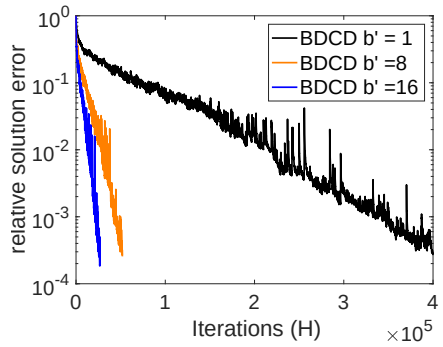
Figure 3.4 compares the convergence behavior of BCD and CA-BCD for $s > 1$. We plot the



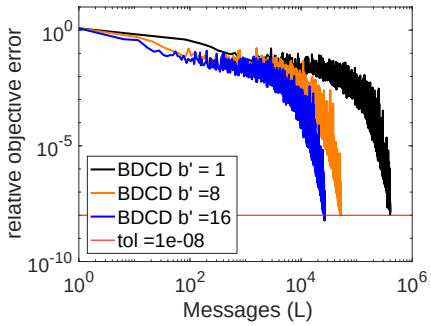
(a) news20



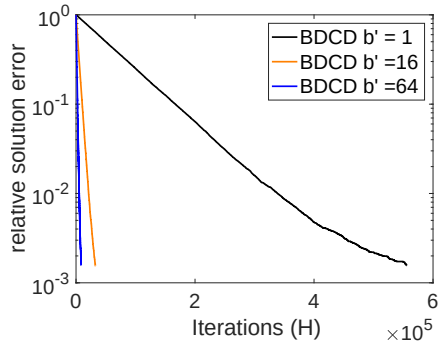
(b) news20



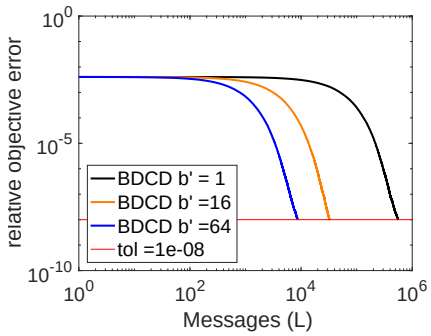
(c) a9a



(d) a9a



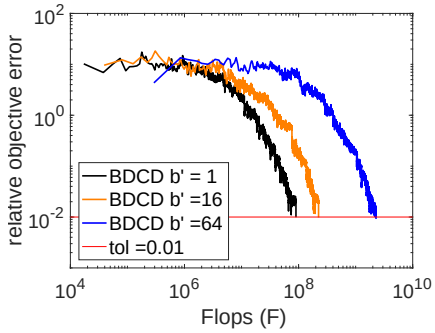
(e) real-sim



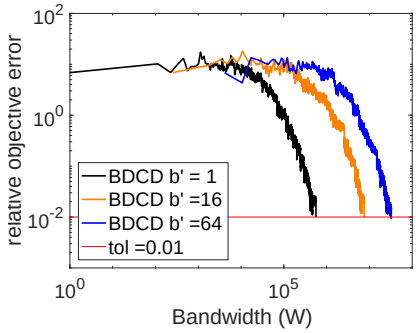
(f) real-sim

Figure 3.5: Convergence behavior of BDCD for several block sizes, b', such that $1 \leq b' < m$. We plot relative solution error and relative objective error with $\lambda = 1000\sigma_{min}$. Note that the X-axis is equivalent to the number of iterations (modulo $\log_{10}$ scale).

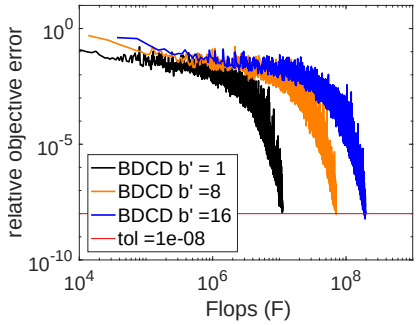
relative solution error, relative objective error and statistics of the Gram matrix condition numbers. The convergence plots indicate that CA-BCD shows almost no deviation from the BCD convergence. While the Gram matrix condition numbers increase with s for CA-BCD, those condition numbers are not so large as to significantly alter the numerical stability. Figures 3.4d and 3.4f



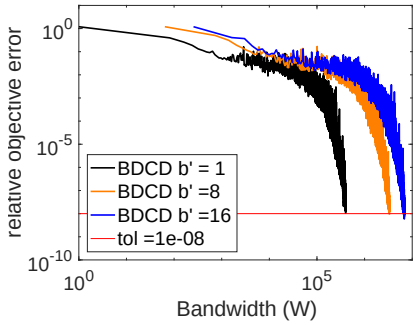
(a) news20



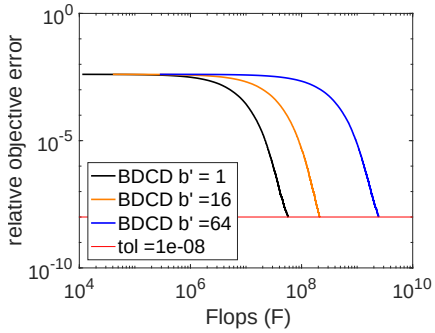
(b) news20



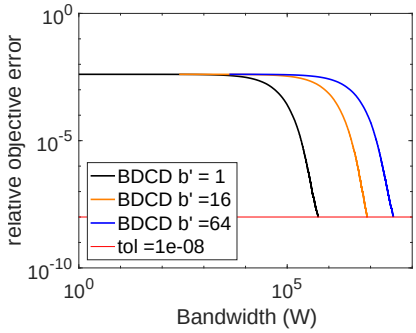
(c) a9a



(d) a9a



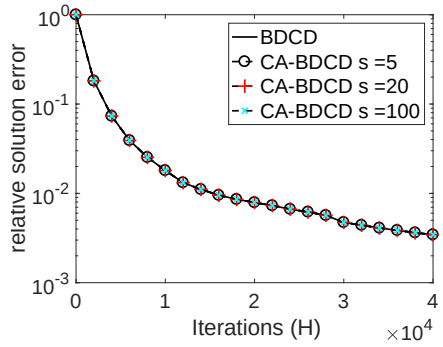
(e) real-sim



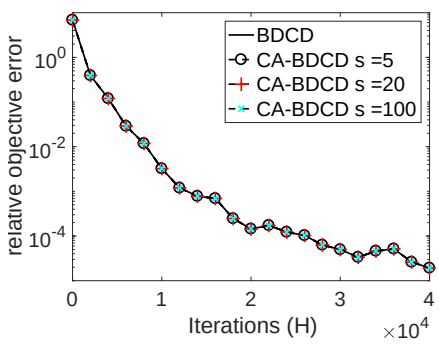
(f) real-sim

Figure 3.6: Convergence behavior of BDCD for several block sizes, b', such that $1 \leq b' < m$. We plot flops cost and bandwidth cost versus convergence with $\lambda = 1000\sigma_{min}$.

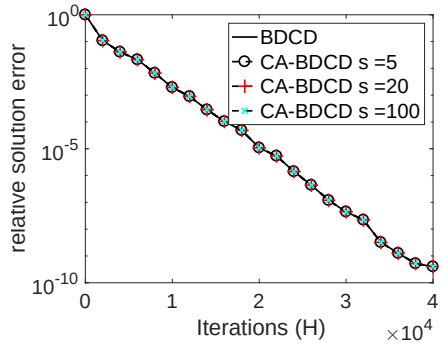
show that the objective error converges very close to machine precision, $\epsilon_{mach} \approx 1e\text{-}16$. The well-conditioning of the real-sim dataset in addition to the regularization and small block size (relative to n) makes the Gram matrices almost perfectly conditioned. Based on these results, it is likely that the factor of s increase in flops and bandwidth will be the primary bottleneck.



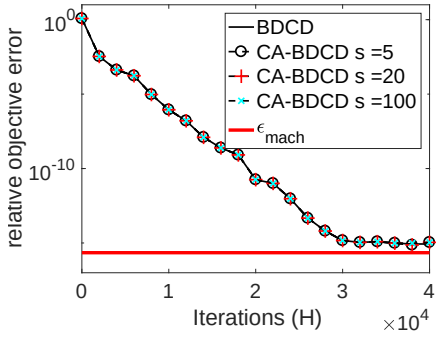
(a) news20



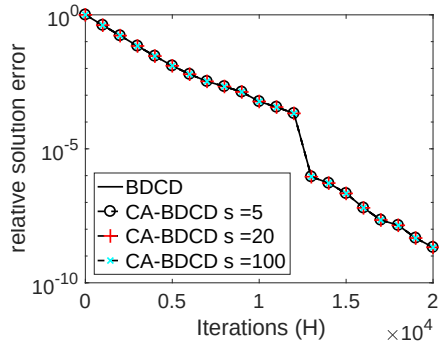
(b) news20



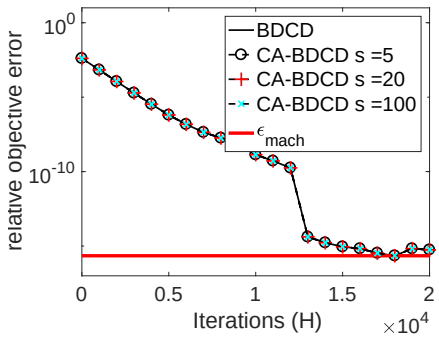
(c) a9a



(d) a9a



(e) real-sim



(f) real-sim

Figure 3.7: Convergence behavior of BDCD and CA-BDCD with several values of s. We plot relative solution error and relative objective error. The block sizes for each dataset are: news20 with $b' = 64$, a9a with $b' = 16$, and real-sim with $b' = 64$.

Block Dual Coordinate Descent

The BDCD algorithm solves the dual of the regularized least-squares problem by computing a $b' \times b'$ Gram matrix obtained from the rows of A (instead of the columns of A for BCD) and solves

a b'-dimensional subproblem at each iteration. Similar to BCD, we expect that as b' increases, the BDCD algorithm converges faster at the cost of more flops and bandwidth. We explore this tradeoff space by comparing the convergence behavior (solution error and objective error) and algorithm costs for BDCD with $1 \leq b' < m$.

Figure 3.5 shows the convergence behavior on the datasets in Table 3.4 for various block sizes and measures the relative solution error (Figs. 3.5a-3.5e) and relative objective error (Figs. 3.5b-3.5f). Similar to BCD, as the block sizes increase the convergence rates of each dataset improves. However, unlike BCD, the objective error does not immediately decrease for some datasets (news20 and a9a). This is expected behavior since BDCD minimizes the dual objective (see Section 3.2) and obtains the primal solution vector, x_h, by taking linear combinations of b' rows of A and x_{h-1}. This also accounts for the non-monotonic decrease in the primal objective and primal solution errors.

Figure 3.6 shows the convergence behavior (in terms of the objective error) vs. flops and bandwidth costs of BDCD for the datasets and block sizes tested in Figure 3.5. We see that small block sizes are more flops and bandwidth efficient while large block sizes are latency efficient (from Figs. 3.5b-3.5f). Due to this tradeoff it important to select block sizes that balance these costs based on machine-specific parameters.

Communication-Avoiding Block Dual Coordinate Descent

The CA-BDCD algorithm avoids communication in the dual problem by unrolling the vector update recurrences by a factor of s. This allows us to reduce the latency cost by computing a larger $sb' \times sb'$ Gram matrix instead of a $b' \times b'$ Gram matrix in the BDCD algorithm. The larger condition number implies that the CA-BDCD algorithm may not be stable, so we begin by experimentally showing the convergence behavior of the CA-BCD algorithm on the datasets in Table 3.4.

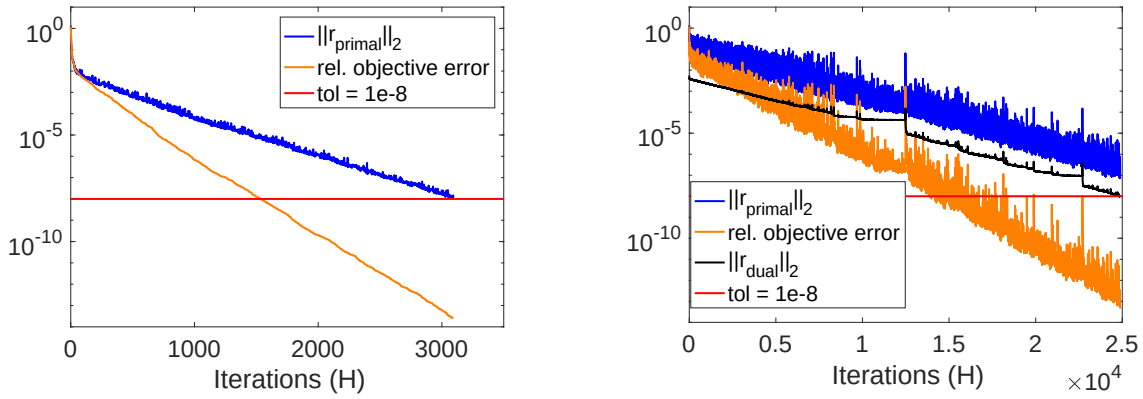
Figure 3.7 compares the convergence behavior of BDCD and CA-BDCD for $s > 1$ with block sizes of $b' = 64, 16$, and 64 for the news20, a9a and real-sim datasets, respectively. The results indicate that CA-BDCD is numerically stable for all tested values of s on all datasets. While the condition numbers of the Gram matrices increase with s, the numerical stability is not significantly affected. The well-conditioning of the real-sim dataset in addition to the regularization and small block size (relative to m) make the Gram matrices almost perfectly conditioned.

Stopping Criterion

At iteration h of BCD, we solve the subproblem

$$\Delta x_h = \left(\frac{1}{m}\mathbb{I}_h^T A^T A \mathbb{I}_h + \lambda \mathbb{I}_h^T \mathbb{I}_h\right)^{-1} \left(-\lambda \mathbb{I}_h^T x_{h-1} - \frac{1}{m}\mathbb{I}_h^T A^T z_{h-1} + \frac{1}{m}\mathbb{I}_h^T A^T y\right).$$

Note that the b-dimensional vector, $\left(-\lambda \mathbb{I}_h^T x_{h-1} - \frac{1}{m}\mathbb{I}_h^T A^T z_{h-1} + \frac{1}{m}\mathbb{I}_h^T A^T y\right)$, is the sub-sampled primal residual vector and is explicitly computed at every iteration. Therefore, a natural stopping criteria is to occasionally compute the full-dimensional residual to check for convergence. Figure 3.8a illustrates the convergence of the residual in comparison to the relative objective error (the



(a) BCD on a9a dataset ($b = 16$).

(b) BDCD on a9a dataset ($b' = 16$).

Figure 3.8: We plot the relative objective error, norm of the primal residual (for BCD Figure 3.8a), and norm of the dual residual (for BDCD Figure 3.8b) for the a9a dataset with block sizes $b = b' = 16$.

optimal objective value is obtained with Conjugate Gradients) for the a9a dataset with $b = 16$. Since the optimal objective value is, in general, unknown the residual can be used as an upper bound on the objective error.

At iteration h of BDCD, we solve the b'-dimensional subproblem

$$\Delta\alpha_h = -\frac{1}{m}\left(\frac{1}{\lambda m^2}\mathbb{I}_h^T AA^T\mathbb{I}_h + \frac{1}{m}\mathbb{I}_h^T\mathbb{I}_h\right)^{-1}\left(-\mathbb{I}_h^T Ax_{h-1} + \mathbb{I}_h^T\alpha_{h-1} + \mathbb{I}_h^T y\right)$$

This $b'-$dimensional vector, $\left(-\mathbb{I}_h^T Ax_{h-1} + \mathbb{I}_h^T\alpha_{h-1} + \mathbb{I}_h^T y\right)$, is the sub-sampled dual residual vector. One can similarly compute the full-dimensional dual residual occasionally to check for convergence. Figure 3.8b illustrates the convergence of the dual residual in comparison to the relative objective error, and the primal residual. We can observe that the dual residual is a lower bound on the primal residual, therefore, the dual problem should be solved to higher accuracy.

3.5 Performance and Scalability Results

In Section 3.4 we showed tradeoffs between convergence behavior and algorithm costs for several datasets. In this section, we explore the performance tradeoffs of standard vs. CA variants on datasets obtained from LIBSVM [26]. We implemented these algorithms in C/C++ using Intel

Algorithm	Name	Data Points (m)	Features (n)	f	residual tolerance (tol)
BCD	a9a	$32,561$	123	0.11	1e-2
	covtype	$581,012$	54	0.22	1e-1
	mnist8m	$8,100,000$	784	0.25	1e-1
BDCD	news20	$15,935$	$62,061$	0.0013	1e-2
	e2006	$3,308$	$150,360$	0.0093	1e-2
	rcv1	$3,000$	$47,236$	0.0017	1e-3

Table 3.5: LIBSVM datasets used in our performance experiments.

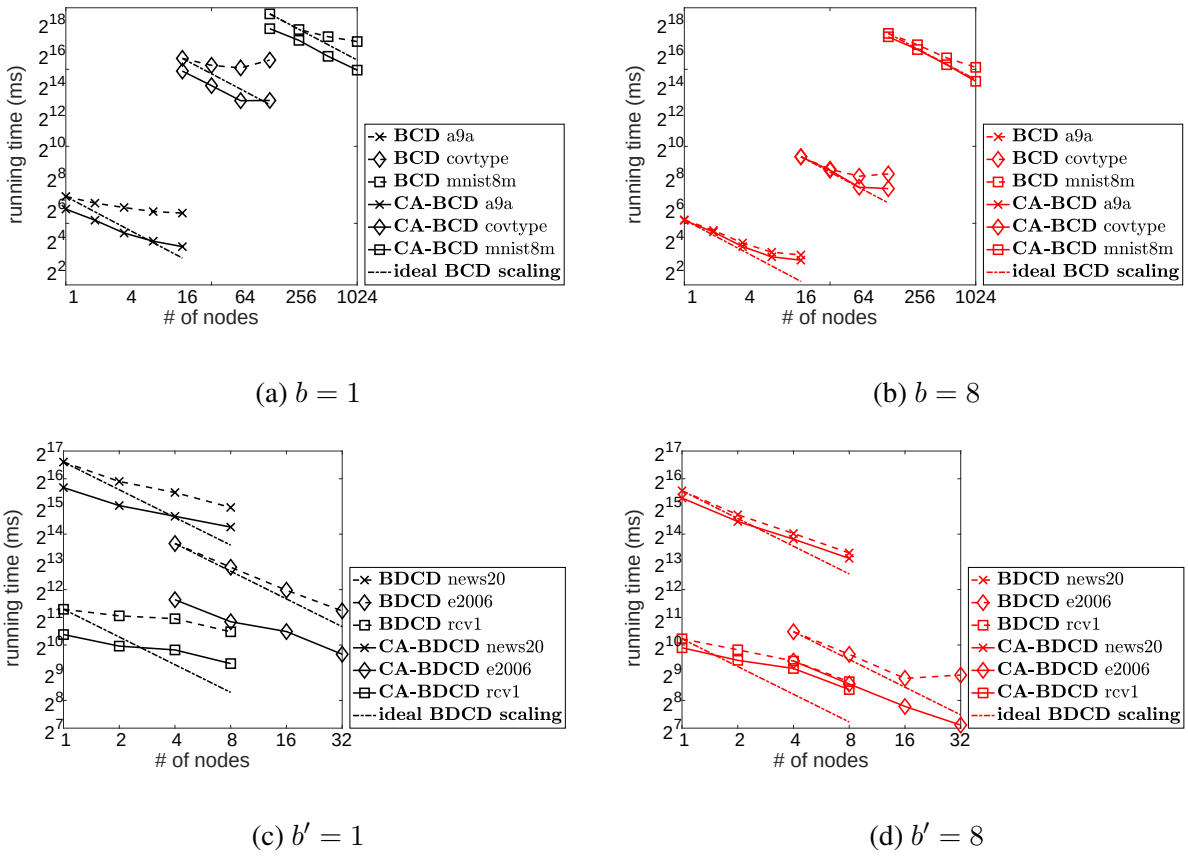
MKL for (sparse and dense) BLAS routines and MPI [62] for parallel processing. While Sections 3.3 and 3.4 assumed dense data for the theoretical analysis and numerical experiments, our parallel implementation stores the data in CSR (Compressed Sparse Row) format. We used a Cray XC30 supercomputer ("Edison") at NERSC [89] to run our experiments on the datasets shown in Table 3.5. We used a 1D-row layout for (CA-)BCD and 1D-column layout for (CA-)BDCD. We ensured that the parallel file I/O was load-balanced (i.e. each processor read roughly equal bytes) and found that the non-zero entries were reasonably well-balanced with less than 5% load imbalance[2]. We constrain the running time of (CA-)BCD and (CA-)BDCD by fixing the residual tolerance for each dataset to the values described in Table 3.5. We ran many of these datasets for smaller tolerances of $1e$-8 and found that our conclusions did not significantly change.

We compares the strong scaling behavior of the standard BCD and BDCD algorithms against their CA variants, shows the running time breakdown to illustrate the flops vs. communication tradeoff, and compares the speedups attained as a function of the number of processors, block size and recurrence unrolling parameter, s.

Strong Scaling

All strong scaling experiments were conducted with one MPI process per processor (flat-MPI) with one warm-up run and three timed runs. Each data point in Figure 3.9 reports the timing breakdown by averaging over three runs. Since there are many processors to choose from, we select the processor with the longest running time for each run and then report the average time spent computing the Gram matrix, solving the subproblem, and communicating. For each dataset in Figure 3.9 we plot the BCD running times, the fastest CA-BCD running times for $s \in \{2, 4, 8, 16, 32\}$, and the ideal scaling behavior. We show the scaling behavior of all datasets for $b \in \{1, 8\}$ to illustrate how the CA-BCD speedups are affected by the choice of block size, b. When the BCD algorithm is entirely latency dominated (i.e. Figure 3.9a), CA-BCD attains speedups of $3.6\times$ (mnist8m), $4.5\times$ (a9a) and $6.1\times$ (covtype). When the BCD algorithm is flops and bandwidth dominated (i.e. Figure 3.9b), CA-BCD attains modest speedups of $1.2\times$ (a9a), $1.8\times$ (mnist8m), and $1.9\times$ (covtype). The strong scaling behavior of the BDCD and CA-BDCD algorithms is shown in Figures 3.9c and 3.9d.

[2]For datasets with highly irregular sparsity structure, additional re-balancing is likely required but we leave this for future work.



(a) $b = 1$ (b) $b = 8$

(c) $b' = 1$ (d) $b' = 8$

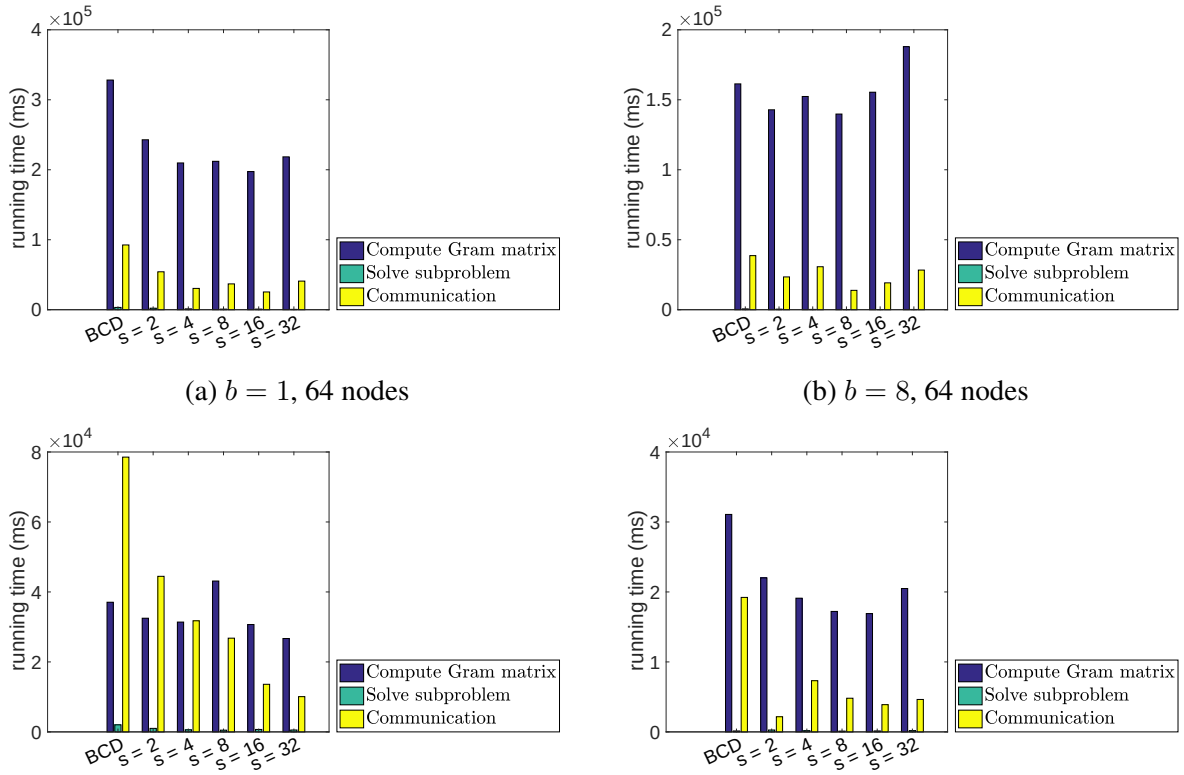
Figure 3.9: Strong scaling results for (CA-)BCD (top row, Figs. 3.9a-3.9b) and (CA-)BDCD (bottom row, Figs 3.9c-3.9d). Ideal strong scaling behavior for BCD and BDCD to illustrate the performance improvements of the CA-variants.

CA-BDCD attains speedups of $1.6\times$ (news20), $2.2\times$ (rcv1), and $2.9\times$ (e2006) when latency dominates and $1.1\times$ (news20), $1.2\times$ (rcv1), and $3.4\times$ (e2006) when flops and bandwidth dominated. The e2006 dataset achieves greater speedup for $b = 8$ than $b = 1$. This is due to machine noise, which caused larger latency times for $b = 8$.

While we did not experiment with weak scaling, we can observe from our analysis (in Section 3.3) that the BCD and BDCD algorithms achieve perfect weak scaling (in theory). It is likely that the CA-BCD and CA-BDCD algorithms would attain weak-scaling speedups by reducing the latency cost by a factor of s, if latency dominates.

Running Time Breakdown

Figure 3.10 shows the running time breakdown of BCD and CA-BCD for $s \in \{2, 4, 8, 16, 32\}$ on the mnist8m dataset. We plot the breakdowns for $b \in \{1, 8\}$ at scales of 64 nodes and 1024 nodes to illustrate CA-BCD tradeoffs for different flops vs. communication ratios. Figures 3.10a



(a) $b = 1$, 64 nodes

(b) $b = 8$, 64 nodes

(c) $b = 1$, 1024 nodes

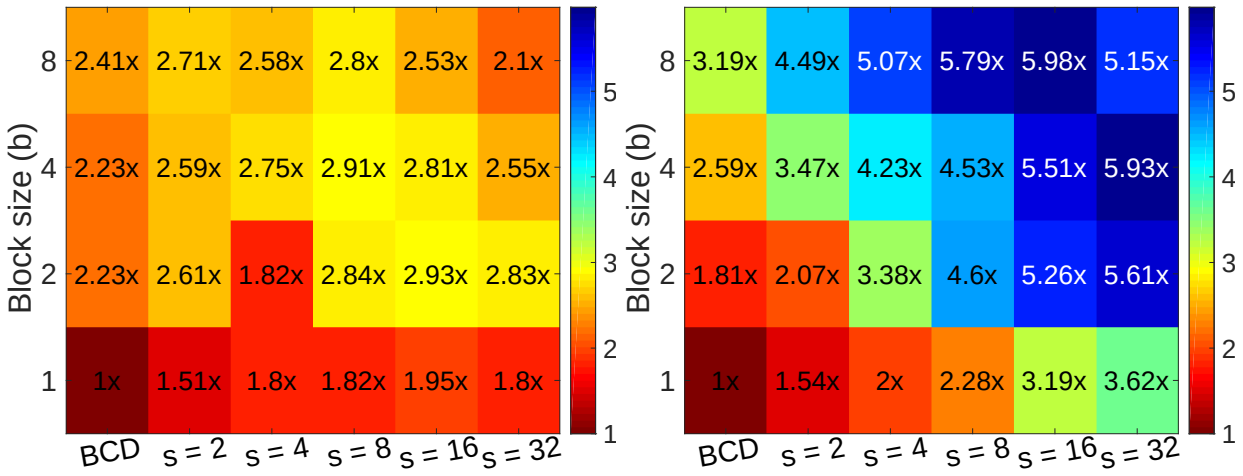
(d) $b = 8$, 1024 nodes

Figure 3.10: Running time breakdown for the mnist8m dataset for $b \in \{1, 8\}$ at scales of 64 and 1024 nodes. We plot the fastest timed run for each algorithm and setting.

and 3.10b show the running time breakdown at 64 nodes for $b = 1$ and $b = 8$, respectively. In both cases flops dominate and speedup for CA-BCD is from faster flops. CA-BCD with $s > 1$ increases the computational intensity and achieves higher flops performance through the use of BLAS-3 GEMM operations. Flops scaling continues until CA-BCD becomes CPU-bound. For $b = 8$, memory-bandwidth is saturated at $s < 8$. For $s \geq 8$ CA-BCD becomes CPU-bound and does not attain any speedup over BCD. Since communication is more bandwidth dominated, less communication speedup is expected. On 1024 nodes (Figs. 3.10c and 3.10d), where latency is more dominant, CA-BCD attains larger communication and overall speedups. These experiments suggest that appropriately chosen values of s, can attain large speedups when latency dominates.

Speedup Comparison

Figure 3.11 summarizes the speedups attainable on the mnist8m dataset at 64 nodes and 1024 nodes for several combinations of block sizes (b) and recurrence unrolling values (s). We normalize the speedups to BCD with $b = 1$. At small scale (Figure 3.11a) we see speedups of $1.95\times$ to $2.91\times$ since flops and bandwidth are the dominant costs. The speedup for larger block sizes is due to



(a) Speedup of mnist8m on 64 nodes.

(b) Speedup of mnist8m on 1024 nodes.

Figure 3.11: Speedups achieved for CA-BCD on mnist8m for various settings of b and s. We show speedups for 64 and 1024 nodes.

faster convergence (i.e. fewer iterations and messages) and due to the use of BLAS-3 matrix-matrix operations. At large scale, when latency dominates, (Figure 3.11b) we observe greater speedups of $3.62\times$ to $5.98\times$. Overall, CA-BCD is fastest for all block sizes and at all scales tested.

3.6 Conclusions and Future Work

In this chapter, we have shown how to extend the communication-avoiding technique of CA-Krylov subspace methods to block coordinate descent and block dual coordinate descent algorithms in machine learning. We showed that in some settings, BCD and BDCD methods may converge faster than traditional Krylov methods – especially when the solution does not require high-accuracy. We analyzed the computation, communication and storage costs of the classical and communication-avoiding variants under two partitioning schemes. Our experiments showed that CA-BCD and CA-BDCD are numerically stable algorithms for all values of s tested, experimentally showing the tradeoff between algorithm parameters and convergence. Finally, we showed that the communication-avoiding variants can attain large speedups of up to $6.1\times$ on a Cray XC30 supercomputer using MPI.

While CA-BCD and CA-BDCD appear to be stable, numerical analysis of these methods would be interesting directions for future work. Extending the CA-technique to other algorithms (SGD, L-BFGS, Newton's method, etc.) would be particularly interesting. Performance comparisons and analyzing the tradeoffs between the various CA methods is an important direction for future work.

Chapter 4

Avoiding Communication in L1-Regularized Least-Squares

The communication-avoiding accelerated block coordinate descent algorithm for solving the L1-regularized least-squares presented in this chapter was developed with co-authors James Demmel, Kimon Fountoulakis, and Michael W. Mahoney. This work was published under the title "Avoiding Synchronization in First-Order Methods for Sparse Convex Optimization" in the IPDPS 2018 conference proceedings and before publication as a technical report under the same title [46].

In this chapter, we are interested in solving regularized least-squares optimization problems, which takes the form

$$\arg\min_{x \in \mathbb{R}^n} \frac{1}{2}\|Ax - b\|_2^2 + \lambda||x||_1 \tag{4.1}$$

Given a data matrix $A \in \mathbb{R}^{m \times n}$ with m data points and n features, labels $y \in \mathbb{R}^m$ and regularization parameter $\lambda \in \mathbb{R}$, we would like to find a solution (or feature weights) $x \in \mathbb{R}^n$. Figure 4.1 illustrates the difference between the solution obtained by least-squares with L2-regularizer (ridge) and L1-regularizer (lasso). Unlike the ridge regularizer which uniformly shrinks x towards zero, lasso biases x towards a sparse solution (at the corners of the L1 ball), where some weights are set exactly to zero. The sparsity-inducing nature of the lasso problem is important when dealing with high-dimensional data and for data interpretability. In the case of high-dimensional and large-scale data, it becomes necessary to parallelize the computations in order to quickly and efficiently solve the lasso problem.

The barrier to efficiently scaling lasso methods on distributed machines is communication cost. Since these methods are iterative, communication is required at every iteration and is often the performance bottleneck. We address this issue by deriving a communication-avoiding variant of an existing lasso method which communicates once every s iterations, where s is a tuning parameter, without altering the convergence rates or numerical stability.

In this chapter, we present empirical results for accelerated and non-accelerated Block Coordinate Descent (BCD)[47] in terms of strong scaling, speedup, and convergence vs. running time

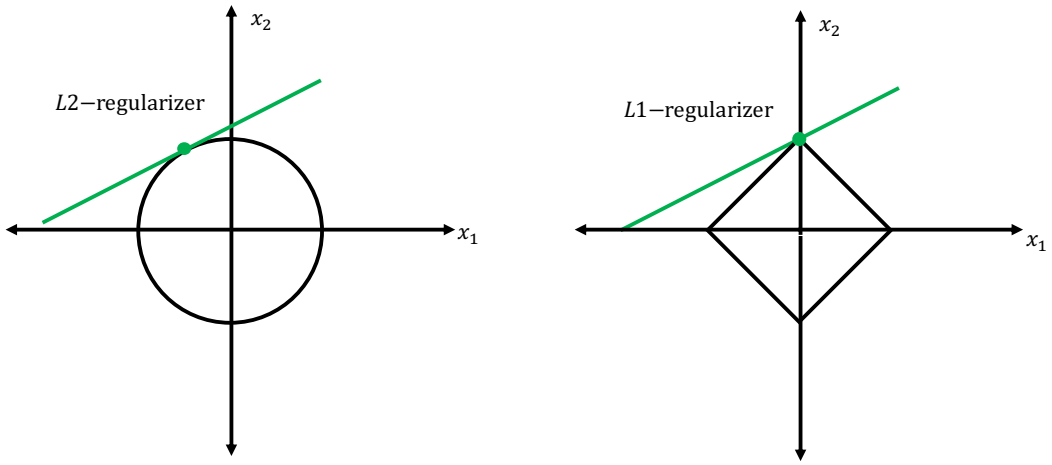


Figure 4.1: Solutions obtained by least-squares with L2-regularizer and L1-regularizer.

results for several LIBSVM datasets. We chose accelerated BCD since it has an optimal convergence rate among the class of first-order methods [47, 90, 101, 102]. The contributions of this chapter are:

- Derivation of a communication-avoiding variant of the accelerated BCD algorithm [47] for the L1-regularized least squares problem.
- Derivation of operational, bandwidth, and latency costs for the standard and communication-avoiding accelerated BCD which show that the communication-avoiding algorithm decreases latency cost by a factor of s at the expense of a factor of s more computation and bandwidth.
- Numerical stability and convergence behavior results which indicate our communication-avoiding algorithm is numerically stable for very large values of s.
- Implementation of the communication-avoiding accelerated BCD algorithm in C++ using the Message Passing Interface (MPI) [62] for distributed-memory parallelism.
- Experimental results which illustrate that our method can perform $1.5\times$-$5.1\times$ faster on up to $12,288$ cores of a Cray XC30 supercomputer [89] on datasets obtained from LIBSVM [26].

4.1 Communication-Avoiding Derivation

In this section, we derive a communication-avoiding version of the accelerated block coordinate descent (accBCD) algorithm for the Lasso problem. The derivation of CA-accBCD (Communication-avoiding accBCD) relies on unrolling the vector update recurrences by a factor of s and rearranging the updates and dependent computations in a communication-avoiding manner. We begin by describing the Lasso problem and then the communication-avoiding derivation. Given

Algorithm 5 Accelerated Block Coordinate Descent (accBCD) Algorithm [47, Alg. 2]

1: **Input**: $A \in \mathbb{R}^{m\times n}, y \in \mathbb{R}^m$, $H > 1$, $w_0 \in \mathbb{R}^n$, $z_0 \in \mathbb{R}^n$, $\lambda \in \mathbb{R}$, $b \in \mathbb{Z}_+$ s.t. $b \leq n$
2: $\theta_0 = b/n,\ \tilde{w}_0 = Aw_0,\ \tilde{z}_0 = Az_0 - b$
3: $q = \lceil n/b \rceil$
4: **for** $h = 1, 2, \cdots, H$ **do**
5: choose $\{i_l \in [n] | l = 1, 2, \ldots, b\}$ uniformly at random without replacement.
6: $\mathbb{I}_h = [e_{i_1}, e_{i_2}, \cdots, e_{i_b}]$
7: Let $\mathbb{A}_h = A\mathbb{I}_h$
8: $v =$ largest eigenvalue of $\mathbb{I}_h^T A^T A \mathbb{I}_h$
9: $\eta_h = \frac{1}{q\theta_{h-1}v}$
10: $r_h = \mathbb{I}_h^T A^T \left(\theta_{h-1}^2 \tilde{w}_{h-1} + \tilde{z}_{h-1}\right)$
11: $g_h = \mathbb{I}_{i_h}^T z_{h-1} - \eta_h r_h$
12: $\Delta z_h = S_{\lambda\eta_h}(g_h) - \mathbb{I}_h^T z_{h-1}$
13: $z_h = z_{h-1} + \mathbb{I}_h \Delta z_h$
14: $\tilde{z}_h = \tilde{z}_{h-1} + A\mathbb{I}_h \Delta z_h$
15: $w_h = w_{h-1} - \frac{1-q\theta_{h-1}}{\theta_{h-1}^2} \mathbb{I}_h \Delta z_h$
16: $\tilde{w}_h = \tilde{w}_{h-1} - \frac{1-q\theta_{h-1}}{\theta_{h-1}^2} A\mathbb{I}_h \Delta z_h$
17: $\theta_h = \frac{\sqrt{\theta_{h-1}^4 + 4\theta_{h-1}^2} - \theta_{h-1}^2}{2}$
18: **Output** $\theta_H^2 w_H + z_H$

a matrix $A \in \mathbb{R}^{m\times n}$ with m data points and n features and a vector of labels $y \in \mathbb{R}^m$, we would like to find the solution $x \in \mathbb{R}^n$ that solves the optimization problem (4.1). The Lasso problem can be solved using many iterative ML algorithms. In this section, we consider the accelerated block coordinate descent algorithm described in [47, Algorithm 2] (reproduced as Algorithm 5). Note that setting $\theta_0 = 0$ results in the non-accelerated block coordinate descent and coordinate descent algorithms. As a result, our communication-avoiding derivation technique in this section should be interpreted as a general technique that applies to accelerated or non-accelerated and coordinate or block coordinate descent methods for the Lasso problem (4.1) by setting b and θ_0 to appropriate values. Let $b \leq n$ be the blocksize and $q = \lceil n/b \rceil$ be the number of blocks. $S_\alpha(\beta)$ is the soft-thresholding operator defined as: $sign(\beta)\max(|\beta| - \alpha, 0)$. If we use the prox operator we would extend our CA-derivation to Newton-type methods. At each iteration we compute v, the Lipschitz constant, which is the largest eigenvalue of the $b \times b$ Gram matrix corresponding to the chosen column block of A. Our CA-technique also applies if the Lipschitz constant is approximately computed.

The recurrences in lines 10 – 16 can be unrolled to avoid communication. We begin the communication-avoiding derivation by changing the loop index from h to $sk + j$ where k is the outer loop index, s is the recurrence unrolling parameter, and j is the inner loop index. Let us assume that we are at iteration $sk + 1$ and have just computed the vectors z_{sk}, $\tilde{z}_{sk}$, w_{sk}, and $\tilde{w}_{sk}$.

Algorithm 6 Communication-Avoiding Accelerated Block Coordinate Descent (CA-accBCD) Algorithm

1: **Input**: $A \in \mathbb{R}^{m\times n}, y \in \mathbb{R}^m$, $H > 1$, $w_0 \in \mathbb{R}^n$, $z_0 \in \mathbb{R}^n$, $\lambda \in \mathbb{R}$, $b \in \mathbb{Z}_+$ s.t. $b \leq n$
2: $\theta_0 = b/n,\ \tilde{w}_0 = Aw_0,\ \tilde{z}_0 = Az_0 - y$
3: $q = \lceil n/b \rceil$
4: **for** $k = 1, 2, \cdots, \frac{H}{s}$ **do**
5: **for** $j = 1, 2, \cdots, s$ **do**
6: choose $\{i_l \in [n] | l = 1, 2, \ldots, b\}$ uniformly at random without replacement.
7: $\mathbb{I}_{sk+j} = [e_{i_1}, e_{i_2}, \cdots, e_{i_b}]$
8: Let $\mathbb{A}_{sk+j} = A\mathbb{I}_{sk+j}$
9: $\theta_{sk+j} = \frac{\sqrt{\theta^4_{sk+j-1} + 4\theta^2_{sk+j-1}} - \theta^2_{sk+j-1}}{2}$
10: Let $Y = A\left[\mathbb{I}_{sk+1}, \mathbb{I}_{sk+2}, \cdots, \mathbb{I}_{sk+s}\right]$.
11: Compute the Gram matrix, $G = Y^T Y$.
12: **for** $j = 1, 2, \cdots, s$ **do**
13: $v =$ large eigenvalue of $\mathbb{I}^T_{sk+j} A^T A \mathbb{I}_{sk+j}$.
14: $\eta_{sk+j} = \frac{1}{q\theta_{sk+j-1} v}$
15: $r_{sk+j} = \mathbb{I}^T_{sk+j} A^T \left(\theta^2_{sk+j-1}\tilde{w}_{sk} + \tilde{z}_{sk}\right)$
16: $g_{sk+j} = \mathbb{I}^T_{i_{sk+j}} z_{sk} - \eta_{sk+j} r_{sk+j} + \sum_{t=1}^{j-1} \mathbb{I}^T_{sk+j}\mathbb{I}_{sk+t}\Delta z_{sk+t}$
$-\eta_{sk+j}\sum_{t=1}^{j-1}\left(\theta^2_{sk+j-1}\frac{1-q\theta_{sk+t-1}}{\theta^2_{sk+t-1}} - 1\right)\mathbb{I}^T_{sk+j}A^T A\mathbb{I}_{sk+t}\Delta z_{sk+t}$
17: $\Delta z_{sk+j} = S_{\lambda\eta_{sk+j}}(g_{sk+j}) - \mathbb{I}^T_{sk+j} z_{sk} - \sum_{t=1}^{j-1}\mathbb{I}^T_{sk+j}\mathbb{I}_{sk+t}\Delta z_{sk+t}$
18: $z_{sk+s} = z_{sk} + \sum_{t=1}^{s}\mathbb{I}_{i_{sk+t}}\Delta z_{sk+t}$
19: $\tilde{z}_{sk+s} = \tilde{z}_{sk} + \sum_{t=1}^{s} A\mathbb{I}_{i_{sk+t}}\Delta z_{sk+t}$
20: $w_{sk+s} = w_{sk} - \sum_{t=1}^{s}\frac{1-q\theta_{sk+t-1}}{\theta^2_{sk+t-1}}\mathbb{I}_{i_{sk+t}}\Delta z_{sk+t}$
21: $\tilde{w}_{sk+s} = \tilde{w}_{sk} - \sum_{t=1}^{s}\frac{1-q\theta_{sk+t-1}}{\theta^2_{sk+t-1}} A\mathbb{I}_{i_{sk+t}}\Delta z_{sk+t}$
22: **Output** $\theta^2_H w_H + z_H$

From this Δz_{sk+1} can be computed by[1]

$$
\begin{aligned}
r_{sk+1} &= \mathbb{I}^T_{sk+1} A^T \left(\theta^2_{sk}\tilde{w}_{sk} + \tilde{z}_{sk}\right) \\
g_{sk+1} &= \mathbb{I}^T_{sk+1} z_{sk} - \eta_{sk+1} r_{sk+1} \\
\Delta z_{sk+1} &= S_{\lambda\eta_{sk+1}}(g_{sk+1}) - \mathbb{I}^T_{sk+1} z_{sk}
\end{aligned}
$$

By unrolling the vector update recurrences for z_{sk+1}, $\tilde{w}_{sk+1}$, and $\tilde{z}_{sk+1}$ (lines 13,14, and 16),

[1] We ignore scalar updates since they can be redundantly stored and computed on all processors.

we can compute r_{sk+2}, g_{sk+2}, and Δz_{sk+2} in terms of z_{sk}, $\tilde{z}_{sk}$, and $\tilde{w}_{sk}$

$$\begin{aligned}
r_{sk+2} &= \mathbb{I}_{sk+2}^T A^T \left(\theta_{sk+1}^2 \tilde{w}_{sk} - \theta_{sk+1}^2 \frac{1-q\theta_{sk}}{\theta_{sk}^2} A\mathbb{I}_{sk+1}\Delta z_{sk+1} + \tilde{z}_{sk} + A\mathbb{I}_{sk+1}\Delta z_{sk+1} \right) \\
&= \theta_{sk+1}^2 \mathbb{I}_{sk+2}^T A^T \tilde{w}_{sk} + \mathbb{I}_{sk+2}^T A^T \tilde{z}_{sk} - \left(\theta_{sk+1}^2 \frac{1-q\theta_{sk}}{\theta_{sk}^2} - 1 \right) \mathbb{I}_{sk+2}^T A^T A \mathbb{I}_{sk+1}\Delta z_{sk+1} \\
g_{sk+2} &= \mathbb{I}_{sk+2}^T z_{sk} + \mathbb{I}_{sk+2}^T \mathbb{I}_{sk+1}\Delta z_{sk+1} - \eta_{sk+2} r_{sk+2} \\
\Delta z_{sk+2} &= S_{\lambda\eta_{sk+2}}(g_{sk+2}) - \mathbb{I}_{sk+2}^T z_{sk} - \mathbb{I}_{sk+2}^T \mathbb{I}_{sk+1}\Delta z_{sk+1}
\end{aligned}$$

By induction we can show that r_{sk+j}, g_{sk+j}, and Δz_{sk+j} can be computed in terms of z_{sk}, $\tilde{z}_{sk}$, and $\tilde{w}_{sk}$

$$\begin{aligned}
r_{sk+j} &= \theta_{sk+j-1}^2 \mathbb{I}_{sk+j}^T A^T \tilde{w}_{sk} + \mathbb{I}_{sk+j}^T A^T \tilde{z}_{sk} \\
&\quad - \sum_{t=1}^{j-1} \left(\theta_{sk+j-1}^2 \frac{1-q\theta_{sk+t-1}}{\theta_{sk+t-1}^2} - 1 \right) \mathbb{I}_{sk+j}^T A^T A \mathbb{I}_{sk+t}\Delta z_{sk+t} \qquad (4.2) \\
g_{sk+j} &= \mathbb{I}_{sk+j}^T z_{sk} - \eta_{sk+j} r_{sk+j} + \sum_{t=1}^{j-1} \mathbb{I}_{sk+j}^T \mathbb{I}_{sk+t}\Delta z_{sk+t} \qquad (4.3) \\
\Delta z_{sk+j} &= S_{\lambda\eta_{sk+j}}(g_{sk+j}) - \mathbb{I}_{sk+j}^T z_{sk} - \sum_{t=1}^{j-1} \mathbb{I}_{sk+j}^T \mathbb{I}_{sk+t}\Delta z_{sk+t}
\end{aligned}$$

for $j = 1, 2, \ldots, s$. Notice that due to the recurrence unrolling we can defer the updates to z_{sk}, w_{sk}, $\tilde{z}_{sk}$, and $\tilde{w}_{sk}$ for s iterations. The summation in (4.2) computes the Gram-like matrices, $\mathbb{I}_{sk+j}^T A^T A \mathbb{I}_{sk+t}$, for $t = 1, 2, \cdots, j-1$. Communication can be avoided in these computations by computing the $sb \times sb$ Gram matrix $G = [\mathbb{I}_{sk+1}, \mathbb{I}_{sk+2}, \cdots, \mathbb{I}_{sk+s}]^T A^T A [\mathbb{I}_{sk+1}, \mathbb{I}_{sk+2}, \cdots, \mathbb{I}_{sk+s}]$ once before the inner loop[2] and redundantly storing it on all processors. Communication can be avoided in the summation in (4.3) by initializing the random number generator on all processors to the same seed. Finally, at the end of the s inner loop iterations we can perform the vector updates

$$\begin{aligned}
z_{sk+s} &= z_{sk} + \sum_{t=1}^{s} \mathbb{I}_{i_{sk+t}} \Delta z_{sk+t} \\
\tilde{z}_{sk+s} &= \tilde{z}_{sk} + \sum_{t=1}^{s} A\mathbb{I}_{i_{sk+t}} \Delta z_{sk+t} \\
w_{sk+s} &= w_{sk} - \sum_{t=1}^{s} \frac{1-q\theta_{sk+t-1}}{\theta_{sk+t-1}^2} \mathbb{I}_{i_{sk+t}} \Delta z_{sk+t} \\
\tilde{w}_{sk+s} &= \tilde{w}_{sk} - \sum_{t=1}^{s} \frac{1-q\theta_{sk+t-1}}{\theta_{sk+t-1}^2} A\mathbb{I}_{i_{sk+t}} \Delta z_{sk+t}
\end{aligned}$$

[2] G is symmetric so computing just the upper/lower triangular part reduces flops and message size by $2\times$.

Summary of operational and storage costs		
Algorithm	Ops cost (F)	Memory cost (M)
accBCD	$O\left(\frac{Hb^2fm}{P} + Hb^3\right)$	$O\left(\frac{fmn+m}{P} + b^2 + n\right)$
CA-accBCD	$O\left(\frac{Hb^2sfm}{P} + Hb^3\right)$	$O\left(\frac{fmn+m}{P} + b^2s^2 + n\right)$

Summary of communication costs		
Algorithm	Latency cost (L)	Message Size cost (W)
accBCD	$O\left(H \log P\right)$	$O\left(Hb^2 \log P\right)$
CA-accBCD	$O\left(\frac{H}{s} \log P\right)$	$O\left(Hsb^2 \log P\right)$

Table 4.1: Ops (F), Latency (L), Bandwidth (W) and Memory per processor (M) costs comparison along the critical path of classical accBCD and CA-accBCD. H is the number of iterations and we assume that $A \in \mathbb{R}^{m \times n}$ is sparse with fmn non-zeros that are uniformly distributed, $0 < f \leq 1$ is the density of A $\left(\text{i.e. } f = \frac{nnz(A)}{mn}\right)$, P is the number of processors and s is the recurrence unrolling parameter. fbm is the non-zeros of the $b \times m$ matrix with b sampled columns from A at each iteration. We assume that the $b \times b$ and Gram matrix computed at each iteration are dense.

The resulting communication-avoiding accelerated block coordinate descent (CA-accBCD) algorithm is shown in Algorithm 6. Since our communication-avoiding technique relies on rearranging the computations, the convergence rates and behavior of the standard accelerated BCD algorithm (Algorithm 5) is maintained (in exact arithmetic).

CA-accBCD computes a larger $sb \times sb$ Gram matrix every s iterations, which results in a computation-communication tradeoff where CA-accBCD increases the flops and message size in order to reduce the latency by s. If the latency cost is the most dominant term then CA-accBCD can attain s-fold speedup over accBCD. In general there exists a tradeoff between s and the speedups attainable. Table 4.1 summarizes the operational, storage, and communication costs of the CA and non-CA methods.

4.2 Algorithm Analysis

We analyze the operational, communication, and storage costs of the Accelerated Block Coordinate Descent (accBCD) and the Communication-Avoiding Accelerated Block Coordinate Descent (CA-accBCD) method to illustrate that CA-accBCD avoids communication. We assume that the matrix $A \in \mathbb{R}^{m \times n}$ is uniformly sparse with fmn non-zeros where $0 < f \leq 1$ is the density of A and $1 \leq b \leq n$ is the blocksize. We further assume that the Gram matrix computed at each iteration, the residual vectors, and the solution vectors are dense. We assume that A is stored in $1D$-block row layout so that the Gram matrix computation requires an all-reduce instead of a more expensive all-to-all when A is stored in $1D$-block column layout. Since A is sparse, we account for the cost of traversing the sparse data structure to store A (this cost may dominate the cost of floating-point

Summary of datasets			
Name	Features	Data Points	f
url	$3,231,961$	$2,396,130$	0.000036
covtype	54	$581,012$	0.22
news20	$62,061$	$15,935$	0.0013

Table 4.2: Properties of the LIBSVM datasets used in our experiments. Epsilon and url did not fit in DRAM of the local machine for the MATLAB experiments, so we use leu instead.

computations) in addition to the computational cost. The costs of accBCD and CA-accBCD are summarized in Table 4.1.

Classical Algorithm

We begin by analyzing the accBCD algorithm with the assumption that A is distributed row-wise (1D-block row layout), that vectors in $\mathbb{R}^m$ are distributed between the processors, and that vectors in $\mathbb{R}^n$ are replicated on all processors. We bound the amount of data moved by summing the per-iteration message sizes overall iterations of the algorithm.

Theorem 4.2.1. *H iterations of the Accelerated Block Coordinate Descent (accBCD) algorithm with the matrix $A \in \mathbb{R}^{m\times n}$ stored in 1D-block row partitions with a blocksize, b, on P processors along the critical path costs*

$$F = O\left(\frac{Hb^2fm}{P} + Hb^3\right) ops, \quad M = O\left(\frac{fmn+m}{P} + b^2 + n\right) \ words\ of\ memory.$$

Communication requires

$$W = O\left(Hb^2 \log P\right) \ words\ moved, \quad L = O\left(H \log P\right) \ messages.$$

Proof. The accBCD algorithm for scalar Lipschitz constants, that are computed on-the-fly, requires the computation of the Gram matrix $\mathbb{I}_h^T A^T A \mathbb{I}_h$, where $A\mathbb{I}_h$ is the subsampled, $m \times b$ matrix whose columns are chosen from A uniformly at random without replacement. Since the non-zeros are distributed uniformly, $A\mathbb{I}_h$ has bfm non-zeros. Communication is required for computing the Gram matrix and r_h (the former dominates). Due to the 1D-block row layout, each processor computes a $b \times b$ partial Gram matrix which is combined (by summing) and then stored redundantly on each processor at every iteration. The operational cost for computing the Gram matrix, G, is bounded by $O(\frac{b^2fm}{P})$ and also requires communication costs of $O\left(b^2 \log P\right)$ words and $O\left(\log P\right)$. Once the Gram matrix is computed the Lipschitz constant $\lambda_{max}(\mathbb{I}_h^T A^T A \mathbb{I}_h)$ (i.e. the max-eigenvalue of the block) can be directly computed or estimated. Since we assume G is dense, this costs $O(Hb^3)$ operations. Computing r_h, $\tilde{z}_h$, and z_h requires a matrix-vector products $\tilde{w}_h$ and $\tilde{z}_h$. This computation depends only on the non-zeros of $A\mathbb{I}_h$ and, hence, requires $O(\frac{Hbfm}{P})$ operations. Finally, computing the gradient update, g_h, the proximal solution, Δz_h, updating w_h and z_h all cost $O(b)$

Relative objective error			
	url	covtype	news20
CA-accCD	2.2176e-16	2.1514e-16	6.6324e-17
CA-CD	2.2204e-16	1.4203e-16	3.2567e-17
CA-accBCD	2.2204e-16	2.2616e-16	5.6153e-17
CA-BCD	2.2204e-16	2.6451e-16	8.8625e-17

Table 4.3: We show the relative objective error of the CA-methods compared to the non-CA methods (shown in Figure 4.2). Machine precision is 2.2204e-16 for the MATLAB experiments. We omit the url and epsilon datasets since they do not fit in the DRAM of the single-machine platform used for these MATLAB experiments.

and requires $O(b \log P)$ words to be moved and $O(\log P)$ messages. The remaining computations are scalar computations which do not dominate. Since we perform H iterations the total cost is $O\left(\frac{Hb^2fm}{P} + Hb^3\right)$ operations, $O(Hb^2 \log P)$ words moved, and $O(H \log P)$ messages. We require enough memory to store A, the Gram matrix and several vectors. Due to the $1D$-block row layout m-dimensional vectors are partitioned and n-dimensional vectors are replicated. This costs $O\left(\frac{fmn+m}{P} + b^2 + n\right)$ words of memory. □

Communication-Avoiding Algorithm

The CA-accBCD method (see Alg. 6) re-arranges the algebra of the accBCD algorithm into a form that allows us to avoid communication.

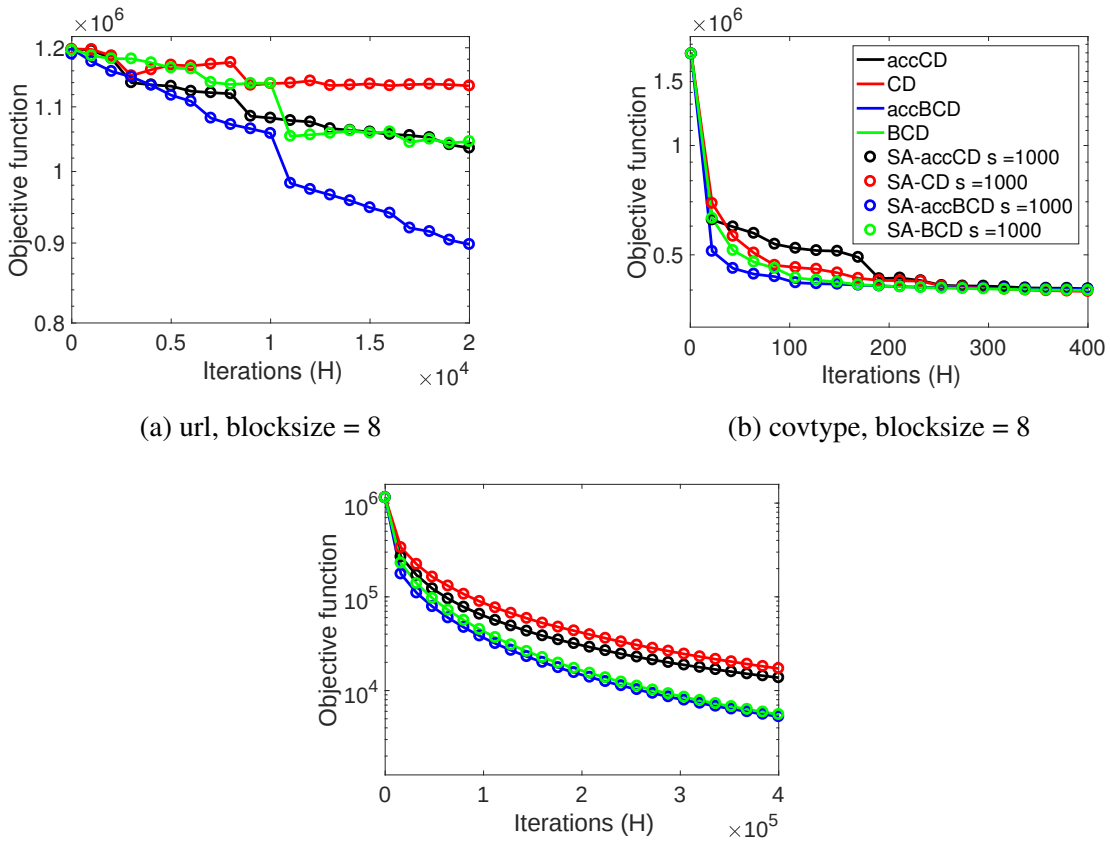
Theorem 4.2.2. *H iterations of the Communication-Avoiding Accelerated Block Coordinate Descent (CA-accBCD) algorithm with the matrix $A \in \mathbb{R}^{m \times n}$ stored in 1D-block row partitions with a blocksize b, on P processors along the critical path costs*

$$F = O\left(\frac{Hsb^2fm}{P} + Hb^3\right) ops, \quad M = O\left(\frac{fmn+m}{P} + b^2s^2 + n\right) \ words \ of \ memory.$$

Communication requires

$$W = O\left(Hsb^2 \log P\right) \ data \ moved, \quad L = O\left(\frac{H}{s} \log P\right) \ messages.$$

Proof. The CA-accBCD algorithm computes the $sb \times sb$ Gram matrix, $G = Y^TY$, where $Y = A\left[\mathbb{I}_{sk+1}, \mathbb{I}_{sk+2}, \cdots, \mathbb{I}_{sk+s}\right]$. Then it computes the Lipschitz constants for the $b \times b$ block-diagonals which correspond to the Gram matrices $\mathbb{I}_{sk+j}^T A^T A \mathbb{I}_{sk+j}$. Due to the 1D-block row layout, each



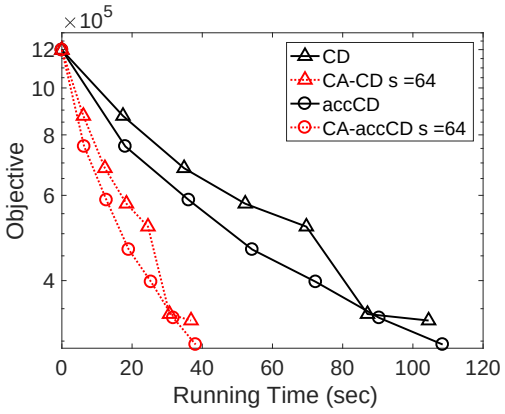
(a) url, blocksize = 8

(b) covtype, blocksize = 8

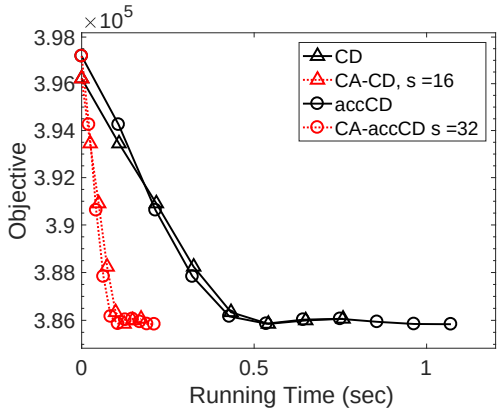
(c) news20, blocksize = 8

Figure 4.2: We compare the convergence of accelerated CD, CD, accelerated BCD, BCD against their communication-avoiding variants (with $s = 1000$) for datasets in Table 4.2 with $\lambda = 100\sigma_{min}$, where σ_{min} is the smallest singular value.

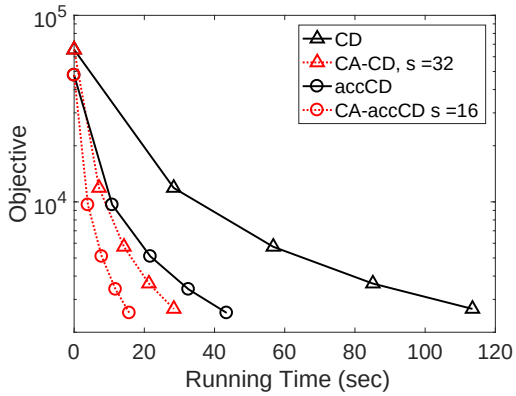
processor can compute a local, partial Gram matrix and then communicates to sum the results (using a reduction and then a broadcast). Since we have to compute a $sb \times sb$ Gram matrix, the operational cost is $O\left(\frac{s^2b^2fm}{P}\right)$ and requires $O\left(s^2b^2 \log P\right)$ data to be moved using $O\left(\log P\right)$ messages. Computing the residual r_{sk+j} for the next s iterations requires a matrix-vector product using Y, which costs $O\left(\frac{bsfm}{P}\right)$ operations, communicates $(bs \log P)$ data in $(O \log P)$ messages. Once these are computed the Lipschitz constants can be computed locally and requires on flops: $O\left(sb^3\right)$, for just the s diagonal $b \times b$ blocks. As in the proof of Theorem 4.2.1 the local vector updates do not dominate the flops and do not require communication (similarly for the scalar computations). Unlike accBCD, the critical path for CA-accBCD only occurs every *s iterations* with a total of $\frac{H}{s}$ critical path computations. Multiplying the dominant costs by the quantity $\frac{H}{s}$ gives the operational, bandwidth, and latency costs. The algorithm requires enough memory to store A, the $sb \times sb$ Gram matrix, and several vectors. This requires $O\left(\frac{fmn+m}{P} + s^2b^2 + n\right)$ words of memory. □



(a) url, Processors = 12288



(b) covtype, Processors = 3072

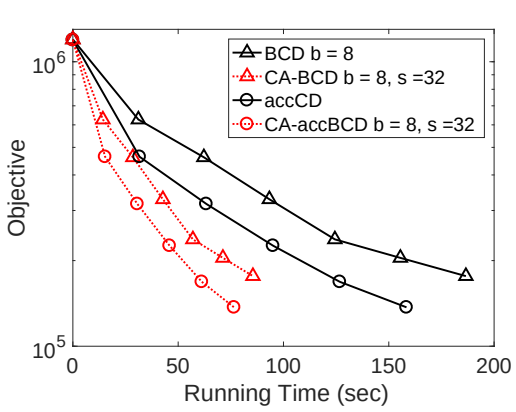


(c) news20, Processors = 768

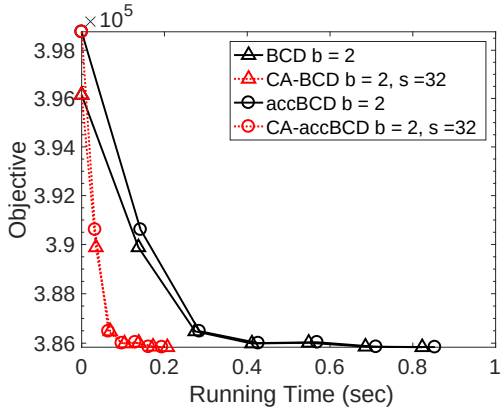
Figure 4.3: Running time vs. convergence of CA and non-CA variants of accelerated and non-accelerated CD.

4.3 Convergence Behavior

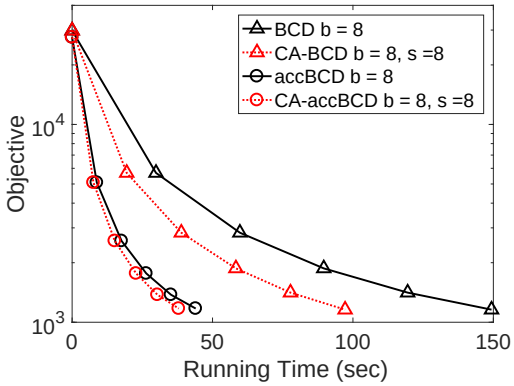
In this section we present experimental results for our communication-avoiding variant and explore the numerical and performance tradeoffs. The recurrence unrolling we propose requires computations of Gram-like matrices whose condition numbers may adversely affect the numerical stability of CA-accBCD. We also re-order the sequence of updates of the solution and residual vectors, which could also lead to numerical instability. We begin with experiments that illustrate that CA-accBCD is numerically stable.We explore the tradeoff between convergence behavior, block size, and s (the recurrence unrolling parameter) for the CA-accBCD algorithm and compare it to the behavior of the standard accBCD algorithm. All numerical stability experiments were performed in MATLAB version R2016b on a 2.3 GHz Intel i7 machine with 8GB of RAM. The datasets used in our experiments were obtained from the LIBSVM repository [26] and are summarized in Table 4.2. We measure the convergence behavior by plotting the objective function value



(a) url, Processors = 12288



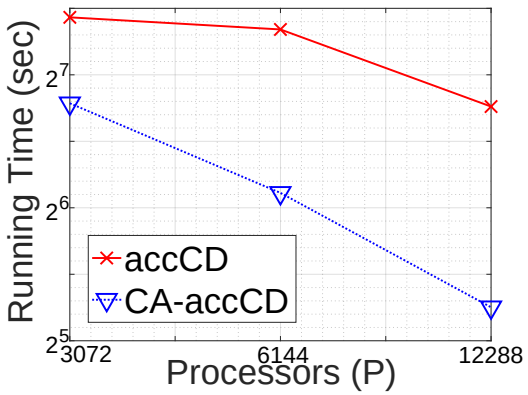
(b) covtype, Processors = 3072



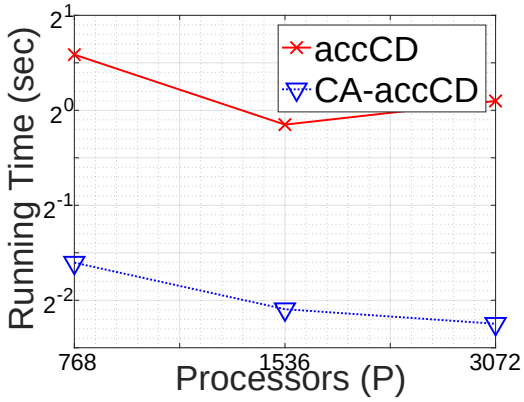
(c) news20, Processors = 768

Figure 4.4: Running time vs. convergence of CA and non-CA variants of accelerated and non-accelerated BCD.

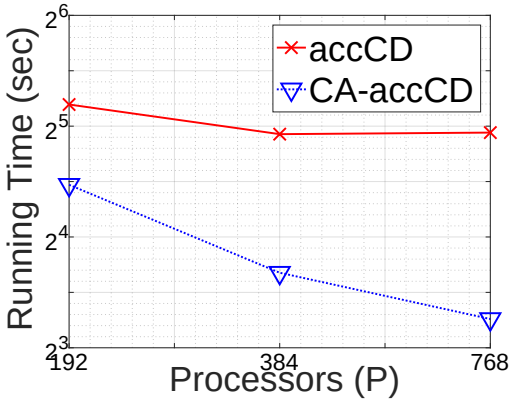
$\frac{1}{2}\|Ax_h - y\|_2^2 + \lambda\|x_h\|_1$ at each iteration. For all experiments, we set $\lambda = 100\sigma_{min}$. We report the convergence vs. iterations to illustrate any differences in convergence behavior. Figure 4.2 shows the convergence behavior of the datasets in Table 4.2 for several blocksizes and with $s = 1000$ for CA-accBCD. The results show that larger blocksizes converge faster than $b = 1$, but at the expense of more computation and larger message sizes. Comparing CA-accBCD and accBCD we observe no numerical stability issues for s as large as 1000 for all datasets tested (in theory we can avoid communication for 1000 iterations). Table 4.3 shows the relative objective error of the CA-methods compared to the non-CA methods. This suggests that the additional computation and message size costs are the performance limiting factors and not numerical instability.



(a) url strong scaling



(b) covtype strong scaling

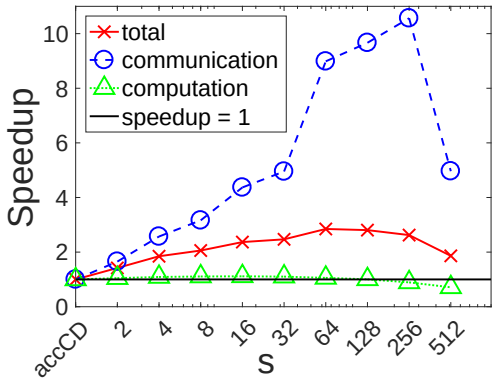


(c) news20 strong scaling

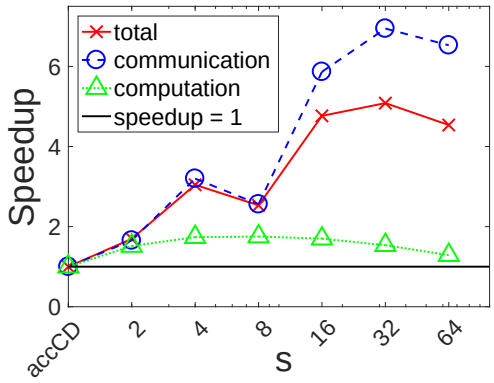
Figure 4.5: Strong scaling and speedups of CA and non-CA accCD.

4.4 Performance and Scalability Results

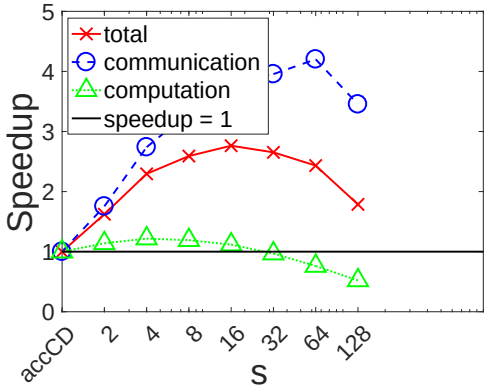
In this section, we present experimental results to show that the CA-methods are faster than their non-CA variants. We consider the datasets in Table 4.2 which were chosen to illustrate performance and speedups on over/under-determined, sparse and dense datasets to illustrate that speedups are independent of those factors. We implement the algorithms in C++ using the Message Passing Interface (MPI) [62] for high-performance, distributed-memory parallel processing. The local linear algebra computations are performed using the Intel MKL library for Sparse and Dense BLAS [78] routines. All methods were tested on a Cray XC30 supercomputer at NERSC which has 24 processors per node and 128GB of memory. The implementation divides the dataset row-wise, however, the CA-methods generalize to other data layout scheme. We choose row-wise since it results in the lowest per iteration communication cost of $O(\log P)$ [45, 116]. All datasets are stored using Compressed Sparse Row format (3-array variant). The solution vectors in $\mathbb{R}^n$ are replicated on all processors and the residual vectors in $\mathbb{R}^m$ are distributed between all processors.



(a) url speedup



(b) covtype speedup

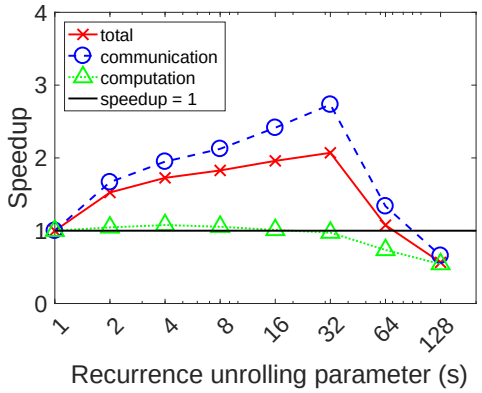


(c) news20 speedup

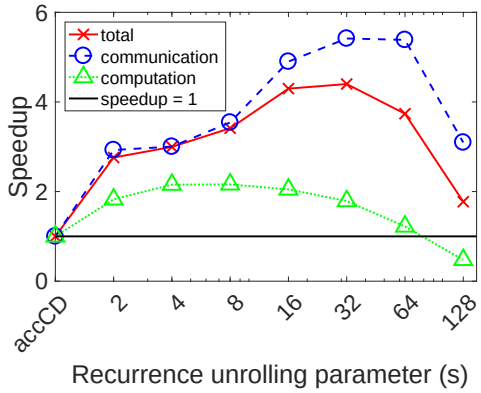
Figure 4.6: Computation and communication speedup results for various values of s. CA and non-CA accCD.

Convergence vs. Running Time

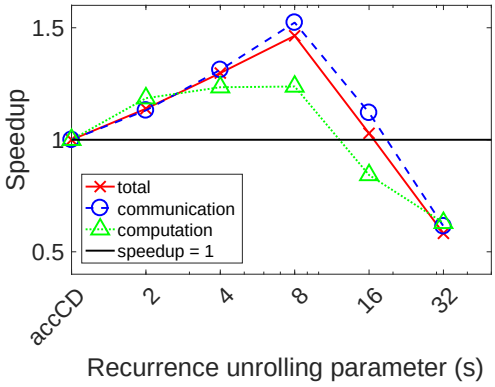
Figures 4.3 and 4.4 show the convergence vs. running time experiments for the datasets in Table 4.2. We present experiments on CD, accCD, BCD, accBCD and their communication-avoiding variants. In all plots, we can observe that the accelerated methods convergence faster than the non-accelerated methods. The BCD methods convergence faster than the CD methods as expected. Since the CA-methods do not alter the convergence rates they are faster per iteration. For the communication-avoiding methods, we plot two values of s, one value (in blue) where we observed the best speedups and a larger value (in red) where we observed less speedups. Note that this decrease in speedup for certain values of s is expected since the CA-methods tradeoff additional message size and computation for a decrease in latency cost. We see CA-accCD speedups of $2.8\times, 5.1\times$, and $2.8\times$ for url, covtype, and news20, respectively. For CA-accBCD the speedups decrease to $2.1\times, 4.4\times$, and $1.5\times$, for url, covtype, and new20, respectively. The decrease in



(a) url speedup



(b) covtype speedup



(c) news20 speedup

Figure 4.7: Computation and communication speedup results for various values of s on CA and non-CA accBCD with $b = 8$ for url and news20 and $b = 2$ for covtype.

speedup is expected since the CA-accBCD method becomes less latency dominant (where CA-methods are less efficient due to additional communication and message sizes costs).

Performance Scaling and Speedups

Figure 4.5 shows the performance strong scaling (problem size is fixed and number of processors is increased) of the accCD vs. CA-accCD methods for different ranges of processors for the datasets tested. We can observe that CA-accCD is faster for all datasets and for all processor ranges. Notice that the gap between accCD and CA-accCD increases with the number of processors.

Figure 4.6 shows the communication, computation, and total speedups attained by CA-accCD for several values of s over accCD. As expected, we see large communication speedups which eventually decreases when the larger message size cost dominate latency. CA-accCD also attains modest computation speedups over accCD. This is because computing the s^2 entries of the Gram

matrix (for CA-accCD) is more cache-efficient (uses a BLAS-3 routine) than computing s individual dot-products (uses a BLAS-1 routine). Once s becomes too large we see slowdowns compared to accCD.

Figure 4.7 shows the communication, computation, and total speedups attained by CA-accBCD for several values of s over accBCD. We use block sizes of $b = 8$ for news20 and url and $b = 2$ for covtype. The first thing we can observe is that the CA-accBCD speedups are less than for CA-accCD. With $b > 1$ the computational and bandwidth costs are higher for accBCD. Since CA-accBCD further increases computational and bandwidth costs by a factor of s, the reduced speedups are expected.

The total speedups achieved by CA-accCD and CA-accBCD are between the respective speedups for computation and communication. Depending on the ratio of computation to communication, the CA-variants can attain large speedups. The url and epsilon datasets are likely to scale to more processors, so additional speedups are be expected.

4.5 Conclusions and Future Work

We showed in this chapter that the CA technique used for L2-regularized least-squares can be extended to the, non-linear, L1-regularizer. In particular, we have derived the CA variants of accelerated BCD and illustrated that our variant attain the same convergence rates. We implemented the CA and non-CA methods in C++ with MPI and showed that the CA implementations attain speedups of $1.5\times$ - $5.1\times$ and can reduce communication by factors of $4.2\times$ - $10.9\times$ on a Cray XC30 supercomputer. While we did not explore other parallel environments, our methods would attain greater speedups on frameworks like Spark [56, 127] due to the large latency costs. Another interesting direction would be to extend the CA technique to higher-order methods (Newton, quasi-Newton, etc.) and frameworks (like proxCoCoA+). Finally, generalizing the CA technique to solve the elastic-net ($g(x) = \lambda||x||_2^2 + (1-\lambda)||x||_1$) and other non-linear regularizers is an important direction for future work.

Chapter 5

Avoiding Communication in Support Vector Machines

The communication-avoiding dual coordinate descent algorithm for solving the linear SVM problem presented in this chapter was developed with co-authors James Demmel, Kimon Fountoulakis, and Michael W. Mahoney. This work was published under the title "Avoiding Synchronization in First-Order Methods for Sparse Convex Optimization" in the IPDPS 2018 conference proceedings and before publication as a technical report under the same title [46].

In this chapter, we derive communication-avoiding coordinate descent algorithms for solving the support vector machines (SVM) problem [32]. Unlike the previous chapters where we solved regression problems, support vector machines primarily solve binary classification problems. Given a dataset that is linearly separable (as illustrated in Figure 5.1), SVM attempts to find a hyperplane (like H_3 in Figure 5.1) which maximizes the margins between the hyperplane and the two classes. The maximum-margin hyperplane can be obtained by selecting two parallel hyperplanes (one for each class) which run through data points closest to the other class. The region between the two parallel hyperplanes is called the *margin*. Clearly the maximum-margin hyperplane is the parallel hyperplane that lies halfway between the bounding hyperplanes. This approach is known as hard-margin SVM and assumes that the dataset is linearly separable (necessary to create the bounding hyperplanes). However, typical binary classification data is often noisy and unlikely to be linearly separable. In this situation soft-margin SVM can be used to obtain a reasonable solution to the binary classification problem by using the hinge-loss in order to balance misclassification and maximizing the margin. It should be noted that the soft-margin SVM problem can obtain the hard-margin solution (if desired) by setting the regularization parameter to a small value. We defer the mathematical description of the soft-margin SVM problem to Section 5.1. The SVM problem can be solved using many optimization algorithms, but we will limit the discussion to first-order methods like stochastic sub-gradient descent [107] and coordinate descent [68]. Both first-order methods have been shown to be computationally more efficient and have stronger convergence rates for large, sparse datasets than related quadratic programming and interior point methods. In practice, coordinate descent has been shown to outperform stochastic sub-gradient descent [68].

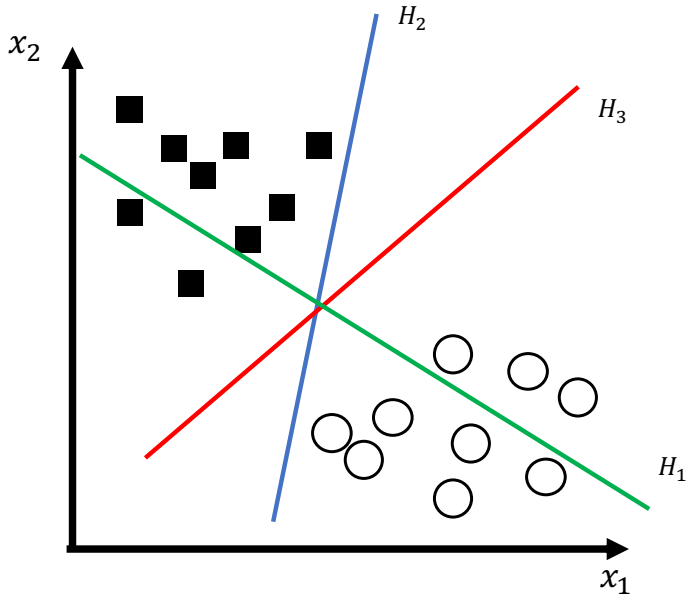


Figure 5.1: Illustration of binary classification using various hyperplanes. H_1 is a non-separating hyperplane and a poor classifier for the data depicted. H_2 is a valid separating hyperplane but does not maximize the margins between the two classes and the hyperplane. H_3 is a valid separating hyperplane and also maximizes the margins.

For this reason, this chapter will focus on deriving a communication-avoiding coordinate descent algorithm to solve the dual soft-margin SVM problem. We solve the dual problem since we can easily kernelize the dot-products and perform binary classification in a high-dimensional Hilbert space. The ability to use kernels makes the SVM problem more widely applicable through the development and use of domain or application specific kernels. Kernel methods are discussed in more detail in Chapter 6. Similar to previous chapters, we leverage the s-step technique from CA-Krylov and s-step Krylov methods [4, 20, 28, 67] in order to derive our communication-avoiding algorithm for SVM. The contributions of this chapter are:

- Communication-avoiding derivation of the coordinate descent algorithm for soft-margin SVM.
- Parallel computation and communication cost analysis of the standard algorithm and its communication-avoiding variant.
- Numerical experiments to confirm the stability of the CA algorithm for large values of s, where s is a tuning parameter that represents the number of coordinate descent iterations whose computations are re-arranged to avoid communiction.
- Parallel strong scaling results which show that the CA algorithm can attain speedups of up to $4\times$ over the standard algorithm on up to $3,072$ cores of a Cray XC30 supercomputer.

Algorithm 7 Dual Coordinate Descent (DCD) for Linear SVM [68]

Input: $A \in \mathbb{R}^{m \times n}, y \in \mathbb{R}^m$, $H > 1$, $\lambda \in \mathbb{R}$, $\alpha_0 \in \mathbb{R}^m$
$x_0 = \sum_{j=1}^{m} y_i \alpha_i A_i$
for $h = 1 \ldots H$ **do**
  $i_h \in [m]$, chosen uniformly at random.
  $e_{i_h} \in \mathbb{R}^m$, the i_h-th standard basis vector.
  $\eta_h = e_{i_h}^T A A^T e_{i_h} + \omega$
  $g_h = e_{i_h}^T y e_{i_h}^T A x_{h-1} - 1 + \omega e_{i_h}^T \alpha_{h-1}$
  $\tilde{g}_h = |\min(\max(e_{i_h}^T \alpha_{h-1} - g_h, 0), \nu) - e_{i_h}^T \alpha_{h-1}|$
  if $\tilde{g}_h \neq 0$ **then**
    $\theta_h = \min(\max(e_{i_h}^T \alpha_{h-1} - \frac{g_h}{\eta_h}, 0), \nu) - e_{i_h}^T \alpha_{h-1}$
  else
    $\theta_h = 0$
  $\alpha_h = \alpha_{h-1} + \theta_h e_{i_h}$
  $x_h = x_{h-1} + \theta_h e_{i_h}^T y A^T e_{i_h}$
Output: x_H

- Running time breakdown results that highlight the tradeoff between computation time and communication time for various values of s.

Our technique derives an alternate form for coordinate descent for soft-margin SVM by re-arranging the computations to obtain s solution updates per communication round. This allows us to obtain an algorithm whose convergence behavior and sequence of solution updates are equivalent to the original algorithm.

5.1 Communication-Avoiding Derivation

We are given a matrix $A \in \mathbb{R}^{m \times n}$, labels $y \in \mathbb{R}^m$ where y_i are binary labels $\{-1, +1\}$ for each observation A_i (i-th row of A). Support Vector Machines (SVM) solves the optimization problem:

$$\arg\min_{x \in \mathbb{R}^n} \frac{1}{2}||x||_2^2 + \lambda \sum_{i=1}^{m} F(A_i, y_i, x) \tag{5.1}$$

where $F(A_i, y_i, x)$ is a loss function and $\lambda > 0$ is the penalty parameter. In this work, we consider the two loss functions:

$$\max(1 - y_i A_i x, 0) \quad \text{and} \quad \max(1 - y_i A_i x, 0)^2. \tag{5.2}$$

We refer to the first as SVM-L1 and the second as SVM-L2. Recent work [68] has shown that both variants of SVM can be solved efficiently using the dual problem. Therefore, in this work we

Algorithm 8 Communication-Avoiding Linear SVM

1: **Input**: $A \in \mathbb{R}^{m\times n}, y \in \mathbb{R}^m$, $H > 1$, $\lambda \in \mathbb{R}$, $s \in \mathbb{Z}^+$ $\alpha_0 \in \mathbb{R}^m$
2: $x_0 = \sum_{j=1}^{m} y_i \alpha_i A_i$
3: **for** $k = 0, \ldots, \frac{H}{s}$ **do**
4: **for** $j = 1 \ldots s$ **do**
5: $i_{sk+j} \in [m]$, chosen uniformly at random.
6: $e_{i_{sk+j}} \in \mathbb{R}^m$, the i_{sk+j}-th standard basis vector.
7: $\mathbb{I}_k = \left[e_{i_{sk+1}}, \ldots, e_{i_{sk+s}}\right], \mathbb{A}_k = \mathbb{I}_k^T A$
8: $[r'_{sk+1}, \ldots, r'_{sk+s}]^T = \mathbb{A}_k x_{sk}$
9: $G_k = \mathbb{A}_k \mathbb{A}_k^T + \omega I_s$
10: $[\eta_{sk+1}, \ldots, \eta_{sk+s}]^T = \mathrm{diag}(G_k)$
11: **for** $j = 1, \ldots, s$ **do**
12: $\beta_{sk+j} = e_{i_{sk+j}}^T \alpha_{sk} + \sum_{t=1}^{j-1} e_{i_{sk+j}}^T e_{i_{sk+t}} \theta_{sk+t}$
13: $g_{sk+j} = e_{i_{sk+j}}^T y r'_{sk+j} - 1 + \omega \beta_{sk+j}$
$+ e_{i_{sk+j}}^T y \sum_{t=1}^{j-1} \theta_{sk+t} e_{i_{sk+t}}^T y e_{i_{sk+j}}^T A A^T e_{i_{sk+t}}$
14: $\tilde{g}_{sk+j} = |\min(\max(\beta_{sk+j} - g_{sk+j}, 0), \nu) - \beta_{sk+j}|$
15: **if** $\tilde{g}_{sk+j} \neq 0$ **then**
16: $\theta_{sk+j} = \min(\max(\beta_{sk+j} - \frac{g_{sk+j}}{\eta_{sk+j}}, 0), \nu)$
$- \beta_{sk+j}$
17: **else**
18: $\theta_{sk+j} = 0$
19: $\alpha_{sk+s} = \alpha_{sk} + \sum_{t=1}^{s} \theta_{sk+t} e_{sk+t}$
20: $x_{sk+s} = x_{sk} + \sum_{t=1}^{s} \theta_{sk+t} e_{i_{sk+t}}^T y A^T e_{i_{sk+t}}$
21: **Output:** x_H

will consider the dual optimization problem:

$$\arg\min_{\alpha \in \mathbb{R}^m} \frac{1}{2} \alpha^T \bar{Q} \alpha - 1^T \alpha \tag{5.3}$$

$$\text{subject to} \quad 0 \leq \alpha_i \leq \nu, \forall i, \tag{5.4}$$

where $\bar{Q} = Q + D$, where $D = \omega I_m$ and $Q_{ij} = y_i y_j A_i A_j^T$. For SVM-L1, $\omega = 0$ and $\nu = \lambda$ and for SVM-L2 $\omega = \frac{.5}{\lambda}$ and $\nu = \infty$. The dual problem can be solved efficiently by coordinate descent [68] and is shown in Algorithm 7.

The recurrences in lines 7-14 can be unrolled to avoid communication. We begin the CA derivation by changing the loop index from h to $sk + j$ where k is the outer loop index, s is the (tunable) recurrence unrolling parameter, and j is the inner loop index. Let us assume that we are at iteration $sk + 1$ and have just computed the vectors x_{sk} and α_{sk}. From this g_{sk+1} can be

computed by

$$
\begin{aligned}
g_{sk+1} &= e_{i_{sk+1}}^T y e_{i_{sk+1}}^T A x_{sk} - 1 + \omega e_{i_{sk+1}}^T \alpha_{sk}, \\
\tilde{g}_{sk+1} &= |\min(\max(e_{i_{sk+1}}^T \alpha_{sk} - g_{sk+1}, 0), \nu) - e_{i_{sk+1}}^T \alpha_{sk}|, \\
\theta_{sk+1} &= \begin{cases} \min(\max(e_{i_{sk+1}}^T \alpha_{sk} - \frac{g_{sk+1}}{\eta_{sk+1}}, 0), \nu) - e_{i_{sk+1}}^T \alpha_{sk} & , \tilde{g}_{sk+1} \neq 0 \\ 0 & , otherwise \end{cases} \\
\alpha_{sk+1} &= \alpha_{sk} + \theta_{sk+1} e_{i_{sk+1}}, \quad \text{and} \\
x_{sk+1} &= x_{sk} + \theta_{sk+1} e_{i_{sk+1}}^T y A^T e_{i_{sk+1}}.
\end{aligned}
$$

By unrolling the vector update recurrences for α_{sk+1} and x_{sk+1}, we can compute g_{sk+2}, $\tilde{g}_{sk+2}$, and θ_{sk+2} in terms of α_{sk} and x_{sk}. We will ignore the quantities $\tilde{g}_{sk+j}$ and θ_{sk+j} in the subsequent derivations for brevity. We introduce an auxiliary variable, β_{sk+j}, for notational convenience.

$$
\begin{aligned}
\beta_{sk+2} &= e_{i_{sk+2}}^T \alpha_{sk} + \theta_{sk+1} e_{i_{sk+2}}^T e_{i_{sk+1}}, \\
g_{sk+2} &= e_{i_{sk+2}}^T y e_{i_{sk+2}}^T A x_{sk} - 1 \\
&\quad + \theta_{sk+1} e_{i_{sk+1}}^T y e_{i_{sk+2}}^T y e_{i_{sk+2}}^T A A^T e_{i_{sk+1}} + \omega \beta_{sk+2},
\end{aligned}
$$

By induction we can show that g_{sk+j} can be computed in terms of α_{sk} and x_{sk} such that

$$
\beta_{sk+j} = e_{i_{sk+j}}^T \alpha_{sk} + \sum_{t=1}^{j-1} e_{i_{sk+j}}^T e_{i_{sk+t}} \theta_{sk+t}, \tag{5.5}
$$

$$
\begin{aligned}
g_{sk+j} &= e_{i_{sk+j}}^T y e_{i_{sk+j}}^T A x_{sk} - 1 + \omega \beta_{sk+j} \\
&\quad + e_{i_{sk+j}}^T y \sum_{t=1}^{j-1} \theta_{sk+t} e_{i_{sk+t}}^T y e_{i_{sk+j}}^T A A^T e_{i_{sk+t}},
\end{aligned} \tag{5.6}
$$

for $j = 1, 2, \ldots, s$. Due to the recurrence unrolling, we can defer updates to α_{sk} and x_{sk} for s iterations. The summation in (5.5) adds a previous update θ_{sk+t} if the coordinate chosen for update at iteration $sk + t$ is the same as iteration $sk + j$. Communication can be avoided in this step by initializing the random number generator on all processors to the same seed. The summation in (5.6) computes the inner-product $e_{i_{sk+j}}^T A A^T e_{i_{sk+t}}$. Communication can be avoided at this step by computing the Gram-like matrix, $\left[e_{i_{sk+1}}, \ldots, e_{i_{sk+s}}\right]^T A A^T \left[e_{i_{sk+1}}, \ldots, e_{i_{sk+s}}\right]$, upfront at the beginning of the outer loop. Note that the diagonal elements of the resulting matrix are the η_{sk+j}'s required in the inner loop. Finally, at the end of the s inner loop iterations we can perform the vector updates:

$$
\begin{aligned}
\alpha_{sk+s} &= \alpha_{sk} + \sum_{t=1}^{s} \theta_{sk+t} e_{sk+t} \\
x_{sk+s} &= x_{sk} + \sum_{t=1}^{s} \theta_{sk+t} e_{i_{sk+t}}^T y A^T e_{i_{sk+t}}.
\end{aligned}
$$

The resulting CA-SVM algorithm is shown in Alg. 8. The derivation we present in this section only rearranges the algebra. Hence, the convergence rates and behavior of SVM (Alg. 7) do not change (in exact arithmetic). However, in floating-point arithmetic this rearrangement may lead to numerical instability. We will empirically show in Section 5.3 that CA-SVM is numerically stable.

5.2 Algorithm Analysis

In this section, we derive the computation and communication costs of the standard dual coordinate descent and the communication-avoiding dual coordinate descent algorithms which solve the soft-margin SVM problem. From the derivation in Section 5.1 we can observe that dual coordinate descent performs computations on AA^T which requires n-dimensional dot-products. In order to parallelize the dot-products, we partition the data in $1D$-block column layout (feature partitioning). This layout requires a single all-reduce at every iteration whereas $1D$-block row layout requires a more expensive all-to-all. We will assume that A is sparse with fmn non-zeros, where $0 < f \leq 1$ is the density of A with uniform distribution of non-zeros (so that each column has fm non-zeros and each row has fn non-zeros. Since A is sparse, our analysis of the computational cost includes the cost of passing over the sparse data structure instead of just counting the floating-point operations associated with the sparse dot product.

Classical Algorithm

We begin with the analysis of the classical coordinate descent algorithm with A stored in $1D$-block column layout and then extend the proofs to the communication-avoiding algorithm.

Theorem 5.2.1. *H iterations of the Dual Coordinate Descent (DCD) algorithm with the matrix $A \in \mathbb{R}^{m \times n}$ stored in $1D$-block column partitions on P processors along the critical path costs*

$$F = O\left(\frac{Hfn}{P}\right) \ ops, \ M = O\left(\frac{fmn+n}{P} + m\right) \ words \ of \ memory.$$

Communication costs

$$W = O\left(H \log P\right) \ words \ moved, \ L = O\left(H \log P\right) \ messages.$$

Proof. The DCD algorithm computes the step-size η_h at every iteration by computing an inner-product with one randomly chosen row, A_i. Once the step-size is computed the vectors x_h and α_h are updated at every iteration. Computing the inner-product requires each processor to locally compute a partial inner-product with only the features stored locally and then perform an all-reduce (a reduction and broadcast) to sum the partial results from each processor. Since each row of A contains fn non-zeros, the parallel inner-product computation requires $O\left(\frac{fn}{P}\right)$ operations. This step also requires communicating a scalar (partial results from each processor) which requires $O(1)$ words to be moved using $O\left(\log P\right)$ messages. Computing the gradient requires an inner-product

between row A_i and the vector x. This requires $O\left(\frac{fn}{P}\right)$ operations, $O\left(\log P\right)$ word moved, and $O\left(\log P\right)$ messages. Once the gradient has been computed, solution vector update requires scalar operations which cost $O(1)$ and no communication. The critical path costs of H iterations of this algorithm are $O\left(\frac{Hfn}{P}\right)$ ops, $O\left(H \log P\right)$ words moved, and $O\left(H \log P\right)$ messages. Each processor requires enough memory to store the m-dimensional vectors α and b, $\frac{n}{P}$ columns of input matrix A and $\frac{n}{P}$ elements of the solution vector x which costs $\frac{fmn}{P} + 2m + \frac{n}{P} = O\left(\frac{fmn+n}{P} + m\right)$ words of memory per processor. □

Communication-Avoiding Algorithm

We derive the computation, communication, and storage costs of the communication-avoiding dual coordinate descent algorithm. We operate under the same assumptions as those used to analyze the costs of the classical algorithm. Note that in this section we will make use of s, the loop unrolling parameter as defined in Section 5.1. From the analysis we will illustrate the asymptotic behavior and tradeoffs due to avoiding communication. In addition to the analysis we will show experimental results to illustrate that the conclusions from the analysis hold in practice.

Theorem 5.2.2. *H iterations of the Communication-Avoiding Dual Coordinate Descent (CA-DCD) algorithm with the matrix $A \in \mathbb{R}^{m \times n}$ stored in $1D$-block column partitions on P processors along the critical path costs*

$$F = O\left(\frac{Hfns}{P} + Hs\right) \; ops, \; M = O\left(\frac{fmn+n}{P} + m + s^2\right) \; words \; of \; memory.$$

Communication costs

$$W = O\left(Hs \log P\right) \; words \; moved, \; L = O\left(\frac{H}{s} \log P\right) \; messages.$$

Proof. The CA-DCD algorithm begins by randomly selecting s rows of A and subsequently computing the $s \times s$ Gram matrix. Note that the diagonal elements are the step-sizes required for the next s iterations. Computing the Gram matrix requires $O\left(\frac{fns^2}{P}\right)$ operations since there are s^2 elements of the Gram matrix to compute, each of which requires $\frac{fn}{P}$multiplications. Once the Gram matrix is computed the inner loop requires updating the residual by taking linear combinations from previous solutions. This correction requires vector operations and costs $O\left(s^2\right)$ operations. Once the s gradients are computed, the solution vector can be updated. Since we perform s updates for every outer loop, this costs $O\left(\frac{sn}{P}\right)$. The critical path occurs every $\frac{H}{s}$ iterations so the total algorithm cost are $O\left(\frac{Hfn}{P} + Hs^2\right)$ ops, $O\left(Hs \log P\right)$ words moved, and $O\left(\frac{H}{s} \log P\right)$ messages. Each processor requires enough memory to store the m-dimensional vectors α and b, the input matrix A and the solution vector x which costs $\frac{fmn}{P} + s^2 + 2m + \frac{n}{P} = O\left(\frac{fmn+n}{P} + m + s^2\right)$ words of memory per processor. □

Summary of datasets			
Name	Features (n)	Data Points (m)	f
w1a	$2,477$	300	0.04
leu	$7,129$	38	1
duke	$7,129$	44	1

Table 5.1: Properties of the LIBSVM datasets used in our numerical stability experiments.

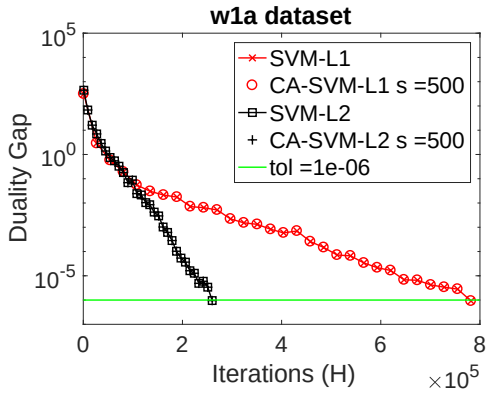
The communication-avoiding variant that we derived requires a factor of s fewer messages than their classical counterparts, at the cost of more computation, bandwidth and memory. Note that the computation and memory costs are unlikely to dominate since we only perform BLAS-1 operations on sparse data. While our CA-variants reduce the latency cost by s, they increase the flops and bandwidth costs by s (due to the $s \times s$ Gram matrix, G_k). Therefore, the best choice of s depends on the relative algorithmic flops, bandwidth, latency costs and their respective hardware parameters.

5.3 Convergence Behavior

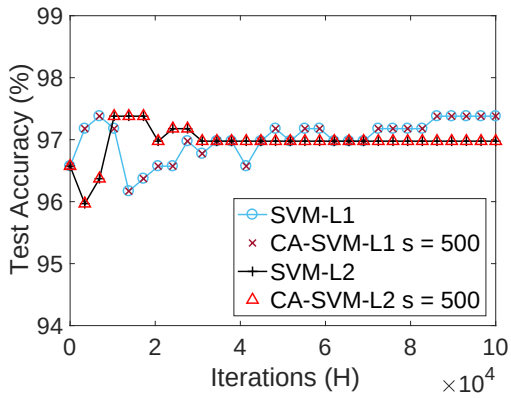
The recurrence unrolling results in the computation of an $s \times s$ Gram matrix, whose condition number may adversely affect numerical stability. So, we begin by verifying the stability of CA-SVM through MATLAB experiments and then illustrate the speedups attainable through our approach.

We conduct numerical stability experiments in MATLAB R2016b on a 2.3GHz Intel i7 machine with 8GB of DRAM. We use binary classification datasets (summarized in Table 6.1) from the LIBSVM [26] repository. We measure the convergence behavior by plotting the duality gap, $P(x) - D(\alpha)$, where $P(x)$ is the primal objective value and $D(\alpha)$ is the dual objective value. Due to strong convexity, primal and dual linear SVM have the same optimal function value [68, 97]. In addition to duality gap, we measure the test accuracy by randomly dividing A into a training set (with 80% of the rows) and a test set (with the remaining rows). We set $\lambda = 1$ for all experiments and show results for SVM-L1 and SVM-L2.

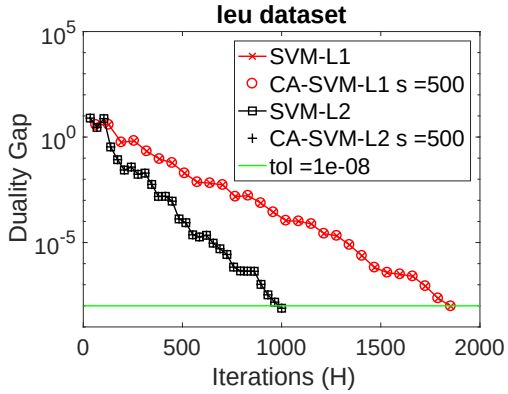
Figure 5.2 shows the convergence behavior and test accuracy of the datasets in Table 6.1. We set $s = 500$ for the CA-variants and plot their duality gap. The potential condition number of the Gram matrix grows with s, so we chose a large s for our experiments, to study the worst-case convergence behavior. Since SVM-L2 solves a smoothed variant of the loss function, it converges to the desired tolerance faster than SVM-L1. Depending on the dataset SVM-L1 may converge to a higher accuracy than SVM-L2. Note that the choice of loss function does not affect the CA-variants whose convergence matches their non-CA counterparts. We can observe for these experiments that the CA-variants are numerically stable. In practice, the additional flops and bandwidth costs are likely to be the limiting factor for the best choice of s.



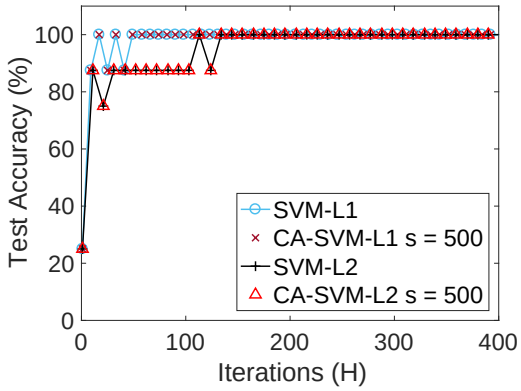
(a) w1a convergence behavior



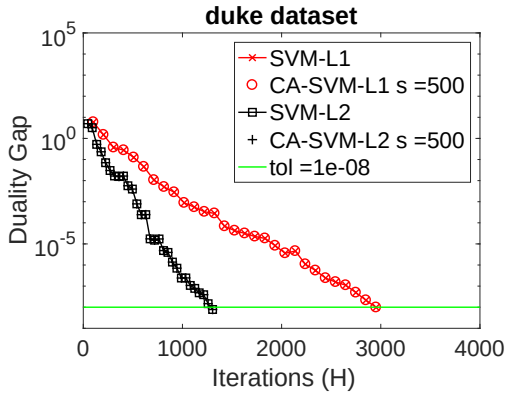
(b) w1a test accuracy



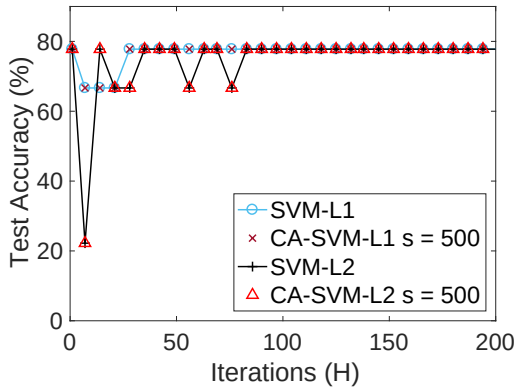
(c) leu convergence behavior



(d) leu test accuracy



(e) duke convergence behavior

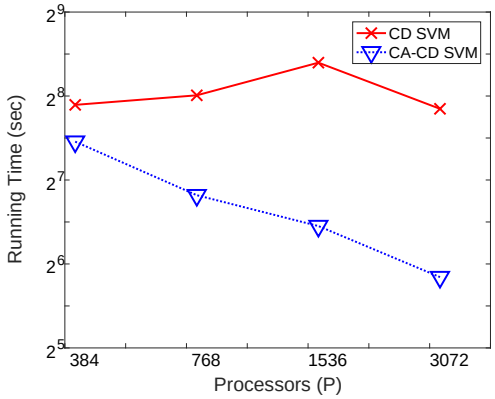


(f) duke test accuracy

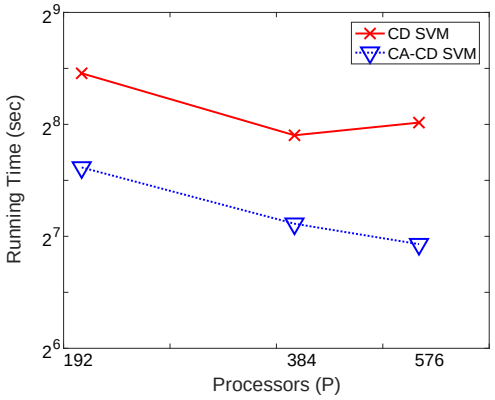
Figure 5.2: Duality gap vs. iterations and test error vs. iterations of SVM-L1, SVM-L2, and their CA variants with $s = 500$.

Summary of datasets			
Name	Features (n)	Data Points (m)	f
gisette	$5,000$	$6,000$	0.991
news20.binary	$1,355,191$	$19,996$	0.00034
rcv1.binary	$20,242$	$47,236$	0.0016

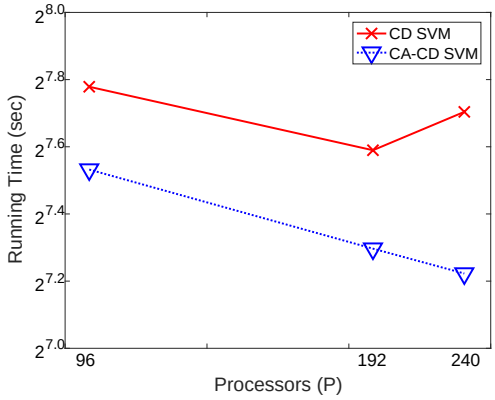
Table 5.2: Properties of the LIBSVM datasets used in our performance experiments.



(a) Gisette Strong Scaling



(b) News20-binary Strong Scaling



(c) RCV1 Strong Scaling

Figure 5.3: Strong scaling comparison of coordinate descent for SVM-L1 and its CA variant.

5.4 Performance and Scalability Results

In this section, we present experimental results to show that the CA-methods that we explored in Section 5.3 are faster than their non-CA variants. We consider the datasets in Table 5.2 which were chosen to illustrate performance and speedups on over/under-determined, sparse and dense datasets to illustrate that speedups are independent of those factors.

We implement the algorithms in C++ using the Message Passing Interface (MPI) [62] for high-performance, distributed-memory parallel processing. The local linear algebra computations are performed using the Intel MKL library for Sparse and Dense BLAS [78] routines. All methods were tested on a Cray XC30 supercomputer at NERSC which has 24 processors per node and 128GB of memory [89]. The implementation divides the dataset column-wise, however, the CA-methods generalize to other data layout schemes. We choose column-wise since it results in the lowest per iteration communication cost of $O(\log P)$ [45, 116]. All datasets are stored using Compressed Sparse Column format (3-array variant). The vectors in $\mathbb{R}^m$ are replicated on all processors and vectors in $\mathbb{R}^n$ are distributed between all processors. The experiments in this section solve the harder SVM-L1 problem with a duality gap tolerance of $1e$-1. Since the CA-variants introduce an additional tuning parameter (s), we search over powers of 2 from $s = 2$ to $s = 256$ and report the running times for the best value and tradeoffs over different values of s.

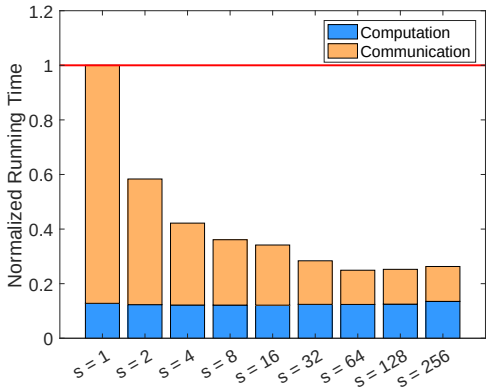
Strong Scaling

Figure 5.3 shows the strong scaling results of the datasets in Table 5.2. The first thing we note is that the CA-variant is faster at all processor settings for all datasets tested. As we increase the number of processors the speedup obtained by the CA-variant also increases. When communication is the dominant cost for the standard DCD-SVM algorithm, the CA-variant can obtain large speedups. For the gisette dataset where we were able to scale to 3072 cores, the CA-variant obtained a speedup of $4\times$. For the news20.binary and rcv1.binary we obtained speedups of $2.1\times$ and $1.4\times$, respectively. In addition to obtaining speedups, the CA-variant scales better and to more cores than the non-CA variant. Despite reducing the latency cost, the CA-variant does not achieve perfect strong scaling due to the additional bandwidth and flops cost.

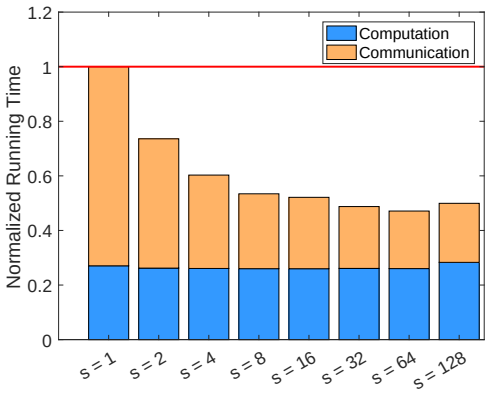
Due to load balancing issues we were unable to scale to large core counts for news20.binary and rcv1.binary and is the reason for non-powers of 2 core counts. The load balancer we currently use takes the input file (stored in row-major order), transposes it (since we use $1D$-block column layout), and reads roughly a $\frac{1}{P}$ fraction of the bytes (a proxy for the number of non-zeros) in the input file. If a processor begins in the middle of a column then we skip to a newline character (i.e. the beginning of the next column). If a processor ends in the middle of a column, then this processor reads and stores this additional column. If the non-zeros in each column are distributed non-uniformly, then this technique can lead to load imbalance. Another approach which counts the number of non-zeros per column and distributes so that each processor stores roughly the same number of non-zeros instead of bytes would likely perform better. This would also facilitate scaling to larger core counts where we would likely observe larger speedups since s can be set to a even larger values.

Timing Breakdown

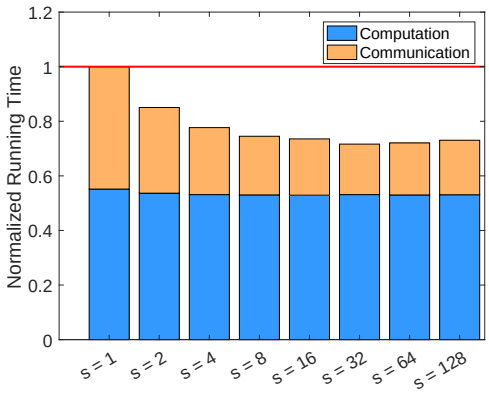
Figure 5.4 shows the communication and computation time for each dataset in Table 5.2. We normalize the running times relative to the non-CA algorithm ($s = 1$ in Figure 5.4). The running time breakdown is reported at several values of s to show the communication time improvements and



(a) Gisette Running Time



(b) News20-binary Running Time



(c) RCV1 Running Time

Figure 5.4: Normalized running time breakdown of coordinate descent for SVM-L1 with various settings for s. Note that $s > 1$ is the communication-avoiding algorithm.

tradeoff with computation time. For all datasets we can observe that as s increases, the communication time decreases. Note that since every iteration of coordinate descent for SVM requires a dot-product. This operation is latency-dominated since we communicate a scalar value and compute a single n-dimensional dot-product [1]. Since latency dominates, the CA algorithm can utilize larger values of s. For the gisette dataset, we can see that communication is the overwhelming cost and at $s = 64$ the CA algorithm can reduce the communication cost by a factor of 7. The news20.binary and rcv1.binary datasets use smaller core counts therefore the communication and total running time speedups are smaller. In some cases, we observe computation time speedups as s increases despite the flops cost increase by s. This is due to the better memory bandwidth utilization for the CA algorithm and through the use of BLAS-3 matrix multiply routines which attain higher peak performance. Note that the additional flops and bandwidth costs eventually dominate

[1] Note that flops cost is often even smaller since these vectors are sparse.

when s is large. In this case, we observe that both flops and communication times increase relative to the best value of s.

5.5 Conclusions and Future Work

In this chapter we derived a communication-avoiding dual coordinate descent algorithm to solve the soft-margin SVM problem. We proved asymptotic bounds on computation, communication, and storage. We illustrated with the analysis that the CA-variant reduces latency cost by a tunable factor s at the expense of additional bandwidth and computation cost. Prior work on s-step methods and CA-Krylov methods exhibited numerical instability for certain values of s. However, our numerical stability results showed that our CA-variant is stable even for unreasonably large values of s where the additional bandwidth and computational costs would likely dominate. Furthermore, we experimentally evaluated the algorithm and showed that the CA-variant scales better and can achieve speedups of up to $4\times$ on 3,072 cores of a Cray XC30 system.

There are several directions for future work for the Support Vector Machines problem. While we have derived CA-variants of the dual SVM problem, doing so for the primal problem is still an open area. The number of iterations required for primal problem does not depend on m, the number data points. Naturally, this is advantageous when the number of data points, m is larger than the number of features, n [107]. We focused on dual coordinate descent, however, deriving CA-variants of stochastic gradient/sub-gradient descent is also a fruitful direction. The theoretical computational cost analysis can be improved. We bounded the operational cost[2], however, a combination of flops and operational cost is required for a more accurate analysis. We also limited ourselves to the $1D$-block column partitioning. Exploring the performance and tradeoff space of different data layouts is an important direction in obtaining the fastest implementation. We used a trivial load balance to distribute the columns of the data matrix. This can likely be improved and would allow the algorithm to scale to more cores.

[2]Defined as the cost of passing over the sparse data structure.

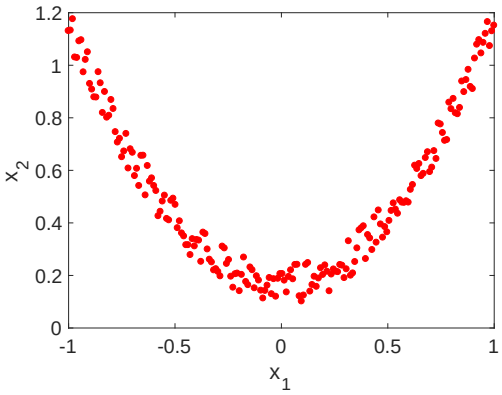
Chapter 6

Avoiding Communication in Kernel Methods

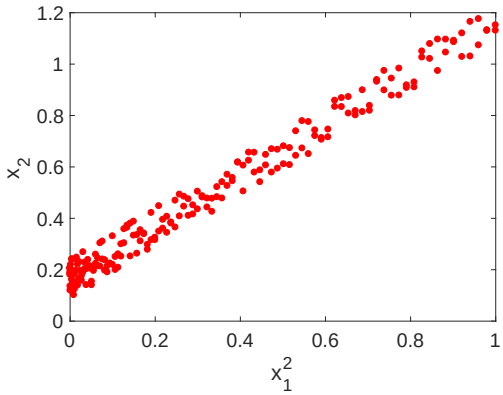
We have derived communication-avoiding block coordinate descent algorithms for several primal and dual machine learning problems. These results focus on regression and binary classification problems which inherently perform a linear fit (regression) or linear separation (binary classification) of the data. As one would expect non-linearity in the data cannot be fitted or separated linearly. It is also important to note that ridge/lasso regression and linear SVM preserve the feature dimension. Data which require higher than the input feature dimensions would obtain poor results with ridge/lasso and linear SVM. In these cases, kernels methods can project the finite-dimensional input features into higher, possibly infinite, dimensional feature spaces. As we have illustrated in previous chapters, ridge/lasso and SVM perform dot-products with A^TA (primal problem) and AA^T (dual problem) where $A \in \mathbb{R}^{m\times n}$ with rows being data points and columns being features. Introducing a kernel changes the features dimension such that the matrix A becomes another matrix $\Phi(A) \in \mathbb{R}^{m\times \xi}$ where $\xi \geq n$. After kernelizing, the primal and dual problems rely on the Gram matrices $\Phi(A)^T\Phi(A)$ and $\Phi(A)\Phi(A)^T$. Notice that for the primal problem we need to compute dot-products with features which requires explicitly storing the high-dimensional features and incurs large computational and storage costs. We can avoid this by instead working with the dual problem which takes dot-products between data points. By defining a kernel function we can perform regression and binary classification in high-dimensional feature spaces without actually computing the features. For example the kernel matrix obtained from the radial basis and polynomial functions (which we will focus on in this chapter)

$$\begin{aligned}\text{linear function: } K_{i,j} &= x_i^T x_j,\\ \text{Radial basis function (RBF): } K_{i,j} &= e^{-\gamma||x_i - x_j||_2^2},\\ \text{Polynomial function (Poly): } K_{i,j} &= \left(x_i^T x_j\right)^d,\end{aligned}$$

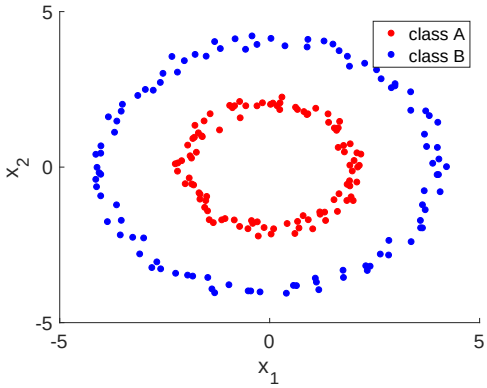
requires only $O(m^2)$ storage and $O(m^2n)$ computation. The kernel matrix dimensions can be reduced even further by using block coordinate descent where only b rows of the kernel matrix need to be computed, where b can be tuned to reduce the storage and computational costs for a



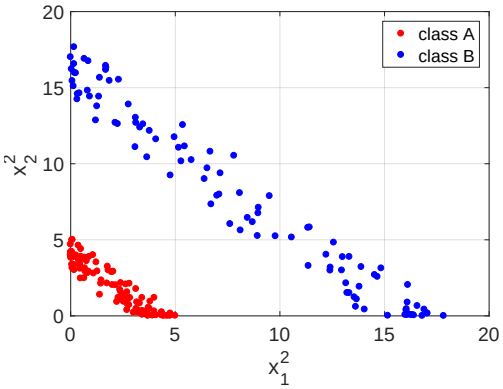
(a) Ridge regression input before kernelizing.



(b) Ridge regression input after kernelizing.



(c) SVM input before kernelizing.



(d) SVM input after kernelizing.

Figure 6.1: Problems where ridge regression and SVM obtain poor results without kernelizing.

given machine. Figure 6.1 illustrates input data which are ill-suited for ridge regression and SVM without a kernel function. For ridge regression, we can use the kernel $(x_1, x_2) \rightarrow (x_1^2, x_2)$ to obtain transformed data that is linear and where ridge regression can now obtain a good fit. For SVM, we can use the kernel $(x_1, x_2) \rightarrow (x_1^2, x_2^2)$ to obtain transformed data which can now be linearly classified. By using non-linear kernel functions and increasing the feature dimension, we can obtain good results from ridge regression and linear SVM. In this chapter, we will focus on solving the kernel ridge regression with Block Dual Coordinate Descent (BDCD) and kernel SVM with Dual Coordinate Descent (DCD). The contributions of this chapter are:

- Derivation of communication-avoiding variants of BDCD and DCD to solve the kernel ridge regression and kernel SVM problems, respectively.
- Derivation of operational, bandwidth and latency costs for the standard and communication-avoiding BDCD and DCD which show that communication-avoiding algorithms reduces latency by s, a tunable parameter, without changing the operational and bandwidth costs.

This is in contrast to previous chapters where operational and bandwidth costs increase by a factor of s.

- Numerical experiments to confirm the stability of communication-avoiding BDCD and DCD for large values of s.
- Modeled strong scaling and speedup results on a predicted Exascale system (using MPI [62] and Spark[127]) and an existing cloud system (using Spark). We leave actual implementation for future work.

6.1 Communication-Avoiding Derivation

In this section, we introduce the kernel ridge regression problem and how to solve it using a computationally-efficient block dual coordinate descent (BDCD) algorithm. Once we derive BDCD, we show how to avoid communication by unrolling the BDCD vector updates and re-arranging the computations. The resulting CA-KRR algorithm produces the same sequence of vector updates and is mathematically equivalent (in exact arithmetic) to the existing algorithm.

Block Dual Coordinate Descent for Kernel Ridge Regression

Given a matrix $A \in \mathbb{R}^{m\times n}$ with m data points and n features and labels $y \in \mathbb{R}^m$ (one for each data point), we are interested in solving the kernel ridge regression (KRR) problem:

$$\arg\min_{\alpha\in\mathbb{R}^m} \frac{\lambda}{2}\left\|\frac{1}{\lambda m}\Phi(A)^T\alpha\right\|_2^2 + \frac{1}{2m}\left\|\alpha - y\right\|_2^2. \tag{6.1}$$

Note that $\Phi(A) \in \mathbb{R}^{m\times\xi}$ with $\xi \geq n$ is the matrix where each data point from the matrix A is represented by a ξ-dimensional feature vector. Since the ξ-dimensional feature space can be infinitely large, explicitly forming $\Phi(A)$ is prohibitively expensive. Fortunately, it is not necessary to form $\Phi(A)$ explicitly by observing that the closed-form solution for (6.1),

$$\alpha = \left(\frac{1}{\lambda m^2}\Phi(A)\Phi(A)^T + \frac{1}{m}I_m\right)^{-1}\left(\frac{1}{m}y\right), \tag{6.2}$$

requires computation of the matrix $K = \Phi(A)\Phi(A)^T \in \mathbb{R}^{m\times m}$. Notice that K is finite-dimensional and only requires dot-products between the high-dimensional feature vectors. As a result, we only need access to the inner-product space[1] associated with $\Phi(A)$. By defining an appropriate kernel for the inner-product space, we can compute K without explicitly accessing the high-dimensional feature space. While this technique reduces the computational and storage complexity, it still requires computing and storing the kernel matrix, K. In order to further reduce this complexity, we consider solving the kernel ridge regression problem using block dual coordinate descent.

[1]Note that we assume that the kernel function satisfies Mercer's condition.

Algorithm 9 Block Dual Coordinate Descent (BDCD) Algorithm for KRR

1: **Input**: $A = [a_1, a_2, \dots a_m]^T \in \mathbb{R}^{m \times n}$, $y \in \mathbb{R}^m$, $H > 1$, $\alpha_0 \in \mathbb{R}^m$, $b \in \mathbb{Z}_+$ s.t. $b \leq m$
2: **for** $h = 1, 2, \cdots, H$ **do**
3: choose $\{i_l \in [m] \mid l = 1, 2, \dots, y\}$ uniformly at random without replacement
4: $\mathbb{I}_h = [e_{i_1}, e_{i_2}, \cdots, e_{i_b}]$
5: $\Theta_h = \frac{1}{\lambda m^2}\mathbb{I}_h^T \Phi(A)\Phi(A)^T \mathbb{I}_h + \frac{1}{m}\mathbb{I}_h^T \mathbb{I}_h$
6: $r_h = \frac{1}{m}\left(-\mathbb{I}_h^T \Phi(A)\Phi(A)^T \alpha_{h-1} + \mathbb{I}_h^T \alpha_{h-1} + \mathbb{I}_h^T y\right)$
7: $\Delta\alpha_h = \Theta_h^{-1} r_h$
8: $\alpha_h = \alpha_{h-1} + \mathbb{I}_h \Delta\alpha_h$
9: **Output** α_H

BDCD solves (6.1) by iteratively solving a b-dimensional sub-problem by sampling b rows from the matrix $\Phi(A)$ and updating the solution α_h (where h is the iteration counter) according to the rule:

$$\alpha_h = \alpha_{h-1} + \mathbb{I}_h \Delta\alpha_h, \tag{6.3}$$

where $\mathbb{I}_h \in \mathbb{R}^{m \times b}$ is the matrix obtained by sub-sampling b columns from the identity matrix, I_m. The solution $\Delta\alpha_h$ can be obtained by solving the following (smaller) optimization problem:

$$\underset{\Delta\alpha_h \in \mathbb{R}^b}{\arg\min} \frac{\lambda}{2}\left\|\frac{1}{\lambda m}\Phi(A)^T \alpha_{h-1} + \frac{1}{\lambda m}\Phi(A)^T \mathbb{I}_h \Delta\alpha_h\right\|_2^2 + \frac{1}{2m}\left\|\alpha_{h-1} + \mathbb{I}_h \Delta\alpha_h - y\right\|_2^2. \tag{6.4}$$

The optimization problem (6.4) has the following closed-form solution

$$\Delta\alpha_h = \left(\frac{1}{\lambda m^2}\mathbb{I}_h^T \Phi(A)\Phi(A)^T \mathbb{I}_h + \frac{1}{m}\mathbb{I}_h^T \mathbb{I}_h\right)^{-1}\left(\frac{1}{m}\mathbb{I}_h^T y - \frac{1}{m}\mathbb{I}_h^T \alpha_{h-1} - \frac{1}{\lambda m^2}\mathbb{I}_h^T \Phi(A)\Phi(A)^T \alpha_{h-1}\right).$$

Note that unlike (6.2), solving for $\Delta\alpha_h$ only requires computation and storage of b rows of the kernel matrix, $\Phi(A)\Phi(A)^T$. The resulting BDCD algorithm is shown in Algorithm 9.

Derivation of CA-KRR

The recurrences in lines 6 and 8 of Algorithm 9 can be unrolled in order to avoid communication. We begin by changing the loop index from h to $sk + j$ where k is the outer loop index, s is the recurrence unrolling parameter and j is the inner loop index. Assume that we are at the beginning of iteration $sk + 1$ and w_{sk} and α_{sk} were just computed. Then $\Delta\alpha_{sk+1}$ can be computed by

$$\Delta\alpha_{sk+1} = \left(\frac{1}{\lambda m^2}\mathbb{I}_{sk+1}^T \Phi(A)\Phi(A)^T \mathbb{I}_{sk+1} + \frac{1}{m}\mathbb{I}_{sk+1}^T \mathbb{I}_{sk+1}\right)^{-1} \left(\frac{1}{m}\mathbb{I}_{sk+1}^T y - \frac{1}{m}\mathbb{I}_{sk+1}^T \alpha_{sk} - \frac{1}{\lambda m^2}\mathbb{I}_{sk+1}^T \Phi(A)\Phi(A)^T \alpha_{sk}\right).$$

Algorithm 10 Communication-Avoiding BDCD for KRR (CA-KRR) Algorithm

Input: $A = [a_1, a_2, \dots a_m]^T \in \mathbb{R}^{m\times n}$, $y \in \mathbb{R}^m$, $H > 1$, $\alpha_0 \in \mathbb{R}^m$, $b \in \mathbb{Z}_+$ s.t. $b \leq m$
for $k = 0, 1, \cdots, \frac{H}{s}$ **do**
  for $j = 1, 2, \cdots, s$ **do**
    choose $\{i_l \in [m] \mid l = 1, 2, \dots, b\}$ uniformly at random without replacement
    $\mathbb{I}_{sk+j} = [e_{i_1}, e_{i_2}, \cdots, e_{i_b}]$
  compute sb rows of the kernel matrix by $G = [\mathbb{I}_{sk+1}, \mathbb{I}_{sk+2}, \dots, \mathbb{I}_{sk+s}]^T \Phi(A)\Phi(A)^T$.
  for $j = 1, 2, \cdots, s$ **do**
    Extract elements from G required to compute $\Delta\alpha_{sk+j}$.
    Compute $\Delta\alpha_{sk+j}$ according to (6.5).
  $\alpha_{sk+j} = \alpha_{sk} + \sum_{t=1}^{s} \mathbb{I}_{sk+t}\Delta\alpha_{sk+t}$
Output α_H

After computing $\Delta\alpha_{sk+1}$, α_{sk+1} can be computed according to (6.3). However, if we were to defer the update and avoid forming α_{sk+1}, then the solution for the next iteration, $sk+2$, is

$$\Delta\alpha_{sk+2} = \left(\frac{1}{\lambda m^2}\mathbb{I}_{sk+2}^T\Phi(A)\Phi(A)^T\mathbb{I}_{sk+2} + \frac{1}{m}\mathbb{I}_{sk+2}^T\mathbb{I}_{sk+2}\right)^{-1} \left(\frac{1}{m}\mathbb{I}_{sk+2}^T y - \frac{1}{m}\mathbb{I}_{sk+2}^T\alpha_{sk} - \frac{1}{\lambda m^2}\mathbb{I}_{sk+2}^T\Phi(A)\Phi(A)^T\alpha_{sk} - \frac{1}{\lambda m^2}\mathbb{I}_{sk+2}^T\Phi(A)\Phi(A)^T\mathbb{I}_{sk+1}\Delta\alpha_{sk+1} - \frac{1}{m}\mathbb{I}_{sk+2}^T\mathbb{I}_{sk+1}\Delta\alpha_{sk+1}\right).$$

By induction we can show that

$$\Delta\alpha_{sk+j} = \left(\frac{1}{\lambda m^2}\mathbb{I}_{sk+j}^T\Phi(A)\Phi(A)^T\mathbb{I}_{sk+j} + \frac{1}{m}\mathbb{I}_{sk+j}^T\mathbb{I}_{sk+j}\right)^{-1} \left(\frac{1}{m}\mathbb{I}_{sk+j}^T y - \frac{1}{m}\mathbb{I}_{sk+j}^T\alpha_{sk} - \frac{1}{\lambda m^2}\mathbb{I}_{sk+j}^T\Phi(A)\Phi(A)^T\alpha_{sk} - \sum_{t=1}^{j-1}\left(\frac{1}{\lambda m^2}\mathbb{I}_{sk+j}^T\Phi(A)\Phi(A)^T\mathbb{I}_{sk+t}\Delta\alpha_{sk+t} - \frac{1}{m}\mathbb{I}_{sk+j}^T\mathbb{I}_{sk+t}\Delta\alpha_{sk+t}\right)\right), \quad (6.5)$$

for $j = 1, 2, \dots, s$. By unrolling the recurrences, we can compute $\Delta\alpha_{sk+j}$ from α_{sk}. Unlike previous chapters where used an auxiliary vector, x, to implicitly represent $\mathbb{I}_{sk+j}AX^T\alpha_{sk+j-1}$. However, since AA^T is replaced with its kernelized version $\Phi(A)\Phi(A)^T$ we can no longer use an auxiliary vector. As a result, the communication-avoiding derivation differs from the one used in Chapter 3. Communication can be avoided for s computations of $\Delta\alpha_{sk+j}$ by computing $[\mathbb{I}_{sk+1}, \dots, \mathbb{I}_{sk+j}]^T\Phi(A)\Phi(A)^T$ with a single round of communication. Once the sb rows of

the kernel matrix have been communicated, the remaining computations can be performed without communication. After the sequence of s solutions have been computed, the solution vector, α_{sk+s}, can be updated by

$$\alpha_{sk+s} = \alpha_{sk} + \sum_{t=1}^{s} \left(\mathbb{I}_{sk+t} \Delta \alpha_{sk+t} \right). \tag{6.6}$$

The resulting CA-KRR algorithm is shown in Alg. 10. It is worth noting that the communication-avoiding derivation just presented holds for any kernel function. However, the maximum attainable value of s (the recurrence unrolling parameter) will depend on the computational complexity associated with the chosen kernel function and the hardware parameters of the target parallel machine.

Dual Coordinate Descent for Kernel Support Vector Machines

In this section, we introduce the support vector machines problem and show how to solve it using a computationally-efficient dual coordinate descent (DCD) algorithm. Once we derive DCD, we show how to avoid communication by unrolling the DCD vector updates and re-arranging the computations. The resulting CA-KSVM algorithm produces the same sequence of vector updates and is mathematically equivalent (in exact arithmetic) to the existing algorithm. We are given a matrix $A \in \mathbb{R}^{m \times n}$, labels $y \in \mathbb{R}^m$ where y_i are binary labels $\{-1, +1\}$ for each observation A_i (i-th row of A). Support Vector Machines (SVM) solves the optimization problem:

$$\arg\min_{x \in \mathbb{R}^n} \frac{1}{2} ||x||_2^2 + \lambda \sum_{i=1}^{m} F(A_i, y_i, x) \tag{6.7}$$

where $F(A_i, y_i, x)$ is a loss function and $\lambda > 0$ is the penalty parameter. In this work, we consider the two loss functions:

$$\max(1 - y_i A_i x, 0) \quad \text{and} \quad \max(1 - y_i A_i x, 0)^2. \tag{6.8}$$

We refer to the first as SVM-L1 and the second as SVM-L2 (with a smoothed loss function). Since we are interested in solving the kernel SVM problem we will instead consider solving the dual optimization problem:

$$\arg\min_{\alpha \in \mathbb{R}^m} \frac{1}{2} \alpha^T \bar{Q} \alpha - 1^T \alpha \tag{6.9}$$

$$\text{subject to} \quad 0 \leq \alpha_i \leq \nu, \forall i, \tag{6.10}$$

where $\bar{Q} = Q + D$, where $D = \omega I_m$ and $Q_{ij} = y_i y_j \Phi(A_i) \Phi(A_j)^T$. For SVM-L1, $\omega = 0$ and $\nu = \lambda$ and for SVM-L2 $\omega = \frac{.5}{\lambda}$ and $\nu = \infty$. The matrix $Q \in \mathbb{R}^{m \times m}$ requires the computation of the kernel matrix, $\Phi(A)\Phi(A)^T$. In our derivation, we avoid using the matrix Q and instead use $\text{diag}(y)\Phi(A)\Phi(A)^T \text{diag}(y)$ in order to make the computational cost clear. We use the operator $\text{diag}(.)$ to represent the transformation of $y \in \mathbb{R}^m$ into a diagonal matrix $\bar{D} \in \mathbb{R}^{m \times m}$ such that

Algorithm 11 Dual Coordinate Descent (DCD) Algorithm for Kernel SVM

1: **Input**: $A \in \mathbb{R}^{m \times n}, y \in \mathbb{R}^m$, $H > 1$, $\lambda \in \mathbb{R}$, $\alpha_0 \in \mathbb{R}^m$
2: **for** $h = 1 \dots H$ **do**
3: $\quad i_h \in [m]$, chosen uniformly at random.
4: $\quad e_{i_h} \in \mathbb{R}^m$, the i_h-th standard basis vector.
5: $\quad \eta_h = e_{i_h}^T \Phi(A)\Phi(A)^T e_{i_h} + \omega$
6: $\quad g_h = e_{i_h}^T y \Phi(A)\Phi(A)^T \operatorname{diag}(y)\alpha_{h-1} - 1 + \omega e_{i_h}^T \alpha_{h-1}$
7: $\quad \tilde{g}_h = |\min(\max(e_{i_h}^T \alpha_{h-1} - g_h, 0), \nu) - e_{i_h}^T \alpha_{h-1}|$
8: $\quad$ **if** $\tilde{g}_h \neq 0$ **then**
9: $\quad\quad \theta_h = \min(\max(e_{i_h}^T \alpha_{h-1} - \frac{g_h}{\eta_h}, 0), \nu) - e_{i_h}^T \alpha_{h-1}$
10: $\quad$ **else**
11: $\quad\quad \theta_h = 0$
12: $\quad \alpha_h = \alpha_{h-1} + \theta_h e_{i_h}$
13: **Output:** α_H

$D_{i,i} = y_i$ for $i = 1, 2, \dots, m$. For large values of m computing and storing Q is prohibitively expensive. As a result, we solve the Kernel SVM problem using dual coordinate descent [2, 68] which is shown in Algorithm 11.

The recurrences in lines 6 and 12 can be unrolled to avoid communication. In the CA derivation of linear SVM we use the auxiliary vector, x, to replace $A^T \operatorname{diag}(y)\alpha_{h-1}$ in line 6 (first term) of Alg. 11. However, we cannot do so with the kernel matrix $\Phi(A)\Phi(A)^T$. As a result, the derivation of kernel SVM differs from the derivation presented in Chapter 5. We begin the CA derivation by changing the loop index from h to $sk + j$ where k is the outer loop index, s is the (tunable) recurrence unrolling parameter, and j is the inner loop index. Let us assume that we are at iteration $sk + 1$ and have just computed the vector α_{sk}. From this g_{sk+1} can be computed by

$$
\begin{aligned}
g_{sk+1} &= e_{i_{sk+1}}^T y e_{i_{sk+1}}^T \Phi(A)\Phi(A)^T diag(y)\alpha_{sk} - 1 + \omega e_{i_{sk+1}}^T \alpha_{sk}, \\
\tilde{g}_{sk+1} &= |\min(\max(e_{i_{sk+1}}^T \alpha_{sk} - g_{sk+1}, 0), \nu) - e_{i_{sk+1}}^T \alpha_{sk}|, \\
\theta_{sk+1} &= \begin{cases} \min(\max(e_{i_{sk+1}}^T \alpha_{sk} - \frac{g_{sk+1}}{\eta_{sk+1}}, 0), \nu) - e_{i_{sk+1}}^T \alpha_{sk}, & \text{if } \tilde{g}_{sk+1} \neq 0 \\ 0 \quad , otherwise \end{cases} \\
\alpha_{sk+1} &= \alpha_{sk} + \theta_{sk+1} e_{i_{sk+1}},
\end{aligned}
$$

where i_{sk+1} is a randomly chosen index from $[m]$ and $e_{i_{sk+1}} \in \mathbb{R}^m$ is the standard basis vector with a 1 in the i_{sk+1} position. By unrolling the vector update recurrences for α_{sk+1}, we can compute g_{sk+2}, $\tilde{g}_{sk+2}$, and θ_{sk+2} in terms of α_{sk}. We will ignore the quantities $\tilde{g}_{sk+j}$ and θ_{sk+j} in the subsequent derivations for brevity and since they depend on g_{sk+j} which requires a kernel computation.

Algorithm 12 Communication-Avoiding DCD for Kernel SVM (CA-KSVM) Algorithm

Input: $A \in \mathbb{R}^{m \times n}, y \in \mathbb{R}^m$, $H > 1$, $\lambda \in \mathbb{R}$, $s \in \mathbb{Z}^+$ $\alpha_0 \in \mathbb{R}^m$
$x_0 = \sum_{i=1}^{m} y_i \alpha_i A_i$
for $k = 0, \ldots, \frac{H}{s}$ **do**
  for $j = 1 \ldots s$ **do**
    $i_{sk+j} \in [m]$, chosen uniformly at random.
    $e_{i_{sk+j}} \in \mathbb{R}^m$, the i_{sk+j}-th standard basis vector.
  $\mathbb{I}_k = \left[e_{i_{sk+1}}, \ldots, e_{i_{sk+s}}\right]$
  Compute s rows of the kernel matrix by $G = \left(\mathbb{I}_k^T y\right)^T \mathbb{I}_k^T \Phi(A)\Phi(A)^T \operatorname{diag}(y)$
  for $j = 1, \ldots, s$ **do**
    Extract elements from G to compute η_{sk+j} and g_{sk+j}
    Compute β_{sk+j} according to (6.11)
    Compute g_{sk+j} according to (6.12)
    $\tilde{g}_h = |\min(\max(\beta_{sk+j} - g_{sk+j}, 0), \nu) - \beta_{sk+j}|$
    if $\tilde{g}_h \neq 0$ **then**
      $\theta_{sk+j} = \min(\max(\beta_{sk+j} - \frac{g_{sk+j}}{\eta_{sk+j}}, 0), \nu) - \beta_{sk+j}$
    else
      $\theta_{sk+j} = 0$
  $\alpha_{sk+j} = \alpha_{sk} + \sum_{t=1}^{s} \theta_{sk+t} e_{i_{sk+t}}$
Output: α_H

We introduce an auxiliary variable, β_{sk+j}, for notational convenience.

$$\begin{aligned}
\beta_{sk+2} &= e_{i_{sk+2}}^T \alpha_{sk} + \theta_{sk+1} e_{i_{sk+2}}^T e_{i_{sk+1}}, \\
g_{sk+2} &= e_{i_{sk+2}}^T y e_{i_{sk+2}}^T \Phi(A)\Phi(A)^T \operatorname{diag}(y)\alpha_{sk} - 1 \\
&\quad + \theta_{sk+1} e_{i_{sk+1}}^T y e_{i_{sk+2}}^T \Phi(A)\Phi(A)^T \operatorname{diag}(y) e_{i_{sk+1}} + \omega \beta_{sk+2},
\end{aligned}$$

By induction we can show that g_{sk+j} can be computed in terms of α_{sk} and x_{sk} such that

$$\beta_{sk+j} = e_{i_{sk+j}}^T \alpha_{sk} + \sum_{t=1}^{j-1} e_{i_{sk+j}}^T e_{i_{sk+t}} \theta_{sk+t}, \tag{6.11}$$

$$\begin{aligned}
g_{sk+j} &= e_{i_{sk+j}}^T y e_{i_{sk+j}}^T \Phi(A)\Phi(A)^T \operatorname{diag}(y)\alpha_{sk} - 1 + \omega \beta_{sk+j} \\
&\quad + e_{i_{sk+j}}^T y \sum_{t=1}^{j-1} \theta_{sk+t} e_{i_{sk+j}}^T \Phi(A)\Phi(A)^T \operatorname{diag}(y) e_{i_{sk+t}},
\end{aligned} \tag{6.12}$$

for $j = 1, 2, \ldots, s$. Due to the recurrence unrolling, we can defer updates to α_{sk} for s iterations. The summation in (6.11) adds a previous update θ_{sk+t} if the coordinate chosen for update at iteration $sk + t$ is the same as iteration $sk + j$. Communication can be avoided in this step by

initializing the random number generator on all processors to the same seed. In (6.12) we compute the i_{sk+j}-th row of the kernel matrix, $e_{i_{sk+j}}^T \Phi(A)\Phi(A)^T$, and compute an inner product with α_{sk} and extract elements to be used in the summation. Note that the summations in (6.11) and (6.12) essential perform an update to the residual. Communication can be avoided at this step by computing s rows of the kernel matrix, $\left[e_{i_{sk+1}}, \ldots, e_{i_{sk+s}}\right]^T \Phi(A)\Phi(A)^T$, upfront at the beginning of the outer loop. Note that the η_{sk+j} term can be extracted from the j-th row of the kernel matrix. Finally, at the end of the s inner loop iterations we can perform the vector updates:

$$\alpha_{sk+s} = \alpha_{sk} + \sum_{t=1}^{s} \theta_{sk+t} e_{sk+t}.$$

The resulting CA-KSVM algorithm is shown in Algorithm 12. The derivation we present in this section only rearranges the algebra. Hence, the convergence rates and behavior of KSVM (Alg. 11) do not change (in exact arithmetic). However, in floating-point arithmetic this rearrangement may lead to numerical instability. We explore the numerically stable of CA-KSVM in the next section.

6.2 Algorithm Analysis

In this section, we derive the computation and communication costs of the block dual coordinate descent and the communication-avoiding block dual coordinate descent algorithms which solve the kernel ridge regression and kernel SVM problems. Note that when $b = 1$ we essentially obtain the costs for DCD. As a result, we will derive the cost for BDCD/CA-BDCD and set $b = 1$ to obtain costs for DCD/CA-DCD for kernel SVM. From the derivation we can observe that BDCD performs computations on AA^T (for the linear kernel) which requires n-dimensional dot-products. In order to parallelize the dot-products, we partition the data in $1D$-block column layout (feature partitioning). This layout requires a single all-reduce at every iteration whereas $1D$-block row layout requires a more expensive all-to-all. We will assume that A is sparse with fmn non-zeros, where $0 < f \leq 1$ is the density of A with uniform distribution of non-zeros (so that each column has fm non-zeros and each row has fn non-zeros. Since A is sparse, our analysis of the computational cost includes the cost of passing over the sparse data structure instead of just counting the floating-point operations associated with the sparse dot product. Note that the radial basis function kernel requires the $exp(\cdot)$ instruction and the polynomial requires the $pow(\cdot)$ instruction. Both instructions are more expensive than a multiply instruction, so we introduce a new parameter,

$$\mu := \frac{\text{cycles for } \{exp, pow, sin, etc.\}}{\text{cycles for } multiply},$$

to accurately capture the operational cost. Note that μ essentially counts the number of multiply operations that can be computed in place of an expensive exp instruction, for example. On most platform instructions like exp require an order of magnitude more cycles to complete than $multiply$ instructions.

Classical Algorithm

We begin with the analysis of the classical BDCD algorithm with A stored in $1D$-block column layout and then extend the proofs to the communication-avoiding algorithm.

Theorem 6.2.1. *H iterations of the Block Dual Coordinate Descent (BDCD) algorithm for kernel ridge regression with the matrix $A \in \mathbb{R}^{m \times n}$ stored in $1D$-block column partitions on P processors along the critical path costs*

$$F = O\left(\frac{Hfbmn}{P} + H\mu bm + Hb^3\right) \ ops, \ M = O\left(\frac{fmn}{P} + bm\right) \ words \ of \ memory.$$

Communication costs

$$W = O\left(Hbm \log P\right) \ words \ moved, \ L = O\left(H \log P\right) \ messages.$$

Proof. The BDCD algorithm computes a $b \times m$ block of the kernel matrix, $\mathbb{I}_h^T \Phi(A)\Phi(A)^T$, at every iteration by computing pair-wise vector operations (RBF: vector subtraction, Poly: dot-product) with $\mathbb{I}_h^T A_{i_l}$ for $l = 1, 2, \ldots, b$ and A_j^T for $j = 1, 2, \ldots, m$. Note that there are bm total vector combinations to be computed and each pair requires $\frac{n}{P}$ operations per processor since the features are distributed. Combining the two costs gives us $\frac{fbmn}{P}$ operations. Note that we also need an all-reduce to aggregate each processor's contribution, which costs $\log P$ messages and bm words (each processor computed a partial $b \times m$ matrix)[2]. After communication we need to kernelize (RBF: multiply by $-\gamma$ then exp operation, Poly: pow operation) the bm entries which costs μbm operations followed by a matrix-vector product to compute $\mathbb{I}_h^T \Phi(A)\Phi(A)^T \alpha_{h-1}$ which costs bm flops. Computing r_h costs m flops and Θ_h can be computed by extracting relevant entries from the matrix, $\mathbb{I}_h^T \Phi(A)\Phi(A)^T$, and solving the linear-system $\Delta\alpha_h = \Theta_h^{-1} r_h$, which costs b^3 operations and no communication. Finally, the solution vector α_h can be updated in b operations (since only b entries are updated) and no communication. Since we perform H iterations overall to reach a desired tolerance the BDCD algorithms costs: $O\left(H\frac{fbmn}{P} + H\mu bm + Hb^3\right)$ operations, $O\left(Hbm \log P\right)$ words moved, and $O\left(H \log P\right)$. Each processor requires enough memory to store the m-dimensional vectors α_h, the labels vector y, $\frac{fmn}{P}$ entries of input matrix A, and bm entries of the kernel matrix, $\mathbb{I}_h^T \Phi(A)\Phi(A)^T$. Storing these quantities costs $\frac{fmn}{P} + 2m + bm = O\left(\frac{fmn}{P} + bm\right)$ words of memory per processor. □

We will now present the analysis for the classical Dual Coordinate Descent (DCD) algorithm for kernel SVM with A, once again, stored in $1D$-block column layout and extend the proofs to CA-DCD for kernel SVM.

[2]For the radial basis function kernel each processor performs pairwise vector subtractions, entry-wise squaring (which requires an additional $\frac{fbmn}{P}$ operations), and summation over the squared vector elements to obtain one partial entry of the $b \times m$ matrix. Since $||A_i - A_j||_2^2 = A_i A_i^T - 2A_i A_j^T + A_j A_j^T$, if we pre-compute and redundantly store $A_i A_i^T$ for $i = 1, 2, \ldots, m$ on all processors, then we can avoid the vector subtractions, element-wise squaring, and perform only the matrix multiply $-2\mathbb{I}_h^T AA^T$ in parallel.

Theorem 6.2.2. *H iterations of the Dual Coordinate Descent (DCD) algorithm for kernel SVM with the matrix $A \in \mathbb{R}^{m \times n}$ stored in $1D$-block column partitions on P processors along the critical path costs*

$$F = O\left(\frac{Hfmn}{P} + H\mu m\right) \; ops, \; M = O\left(\frac{fmn}{P} + m\right) \; words \, of \, memory.$$

Communication costs

$$W = O\left(Hm \log P\right) \; words \, moved, \; L = O\left(H \log P\right) \; messages.$$

Proof. The DCD algorithm computes one row of the kernel matrix, $e_{i_h}^T \Phi(A)\Phi(A) \in \mathbb{R}^{1 \times m}$, at every iteration by computing pair-wise vector operations (operations that are appropriately chosen for the kernel function being used). Since we only compute one row of the kernel matrix there are m vector combinations to be computed and each pair requires $\frac{n}{P}$ operations per processor due to the $1D$-column partitioning. Note that computing one row of the kernel matrix requires $\frac{fmn}{P}$ operations. Since each processor computes a partial row, we much communicate using an all-reduce with summation. Communicating requires the movement of $m \log P$ words in $\log P$ messages. Then we must kernelize the matrix row by applying an entry-wise operation (RBF: multiply by $-\gamma$ then exp operation, Poly: pow operation) which costs μm operations. Once the kernel matrix row is computed we update it by performing an entry-wise multiplication with the label vector y^T, which costs m operations and no communication. After which η_h can be computed easily by extracting one entry. Computing g_h requires a dot-product between the updated kernel matrix row and the solution vector, α_h, which also requires m operations. Finally, the remaining computations only require scalar quantities which are dominated by the previous matrix row and kernelizing computation and vector updates. Since we perform H iteration of DCD, this costs $O\left(H\frac{fmn}{P} + H\mu m\right)$ operations, $O\left(Hm \log P\right)$ words moved, and $O\left(H \log P\right)$ messages. Each processor requires enough memory to store the m-dimensional vector α_h, a row of the kernel matrix, the labels vector y, and $\frac{fmn}{P}$ entries of the input matrix A. Storing these quantities costs $\frac{fmn}{P} + 3m = O\left(\frac{fmn}{P} + m\right)$ words of memory per processor. □

Unlike previous chapters, kernel methods require additional computation and bandwidth when computing a block of b rows of the kernel matrix[3]. This suggests that latency is relatively less dominant and resulting speedups from the CA-BDCD and CA-DCD algorithms to be less than those observed in previous chapters.

Communication-Avoiding Algorithm

In this section, we derive the computation, communication, and storage costs of CA-BDCD for ridge regression and CA-DCD for kernel SVM. We operate under the same assumptions as those used to analyze the costs of the classical algorithm. Note that in this section we will make use of s, the loop unrolling parameter. From the analysis we will illustrate the asymptotic behavior

[3]Note that for kernel ridge regression $b \geq 1$ and for kernel SVM $b = 1$.

and tradeoffs due to avoiding communication. In addition to the analysis we will show predicted experimental results.

Theorem 6.2.3. *H iterations of the Communication-Avoiding Block Dual Coordinate Descent (CA-BDCD) algorithm for kernel ridge regression with the matrix $A \in \mathbb{R}^{m\times n}$ stored in $1D$-block column partitions on P processors along the critical path costs*

$$F = O\left(\frac{Hfbmn}{P} + H\mu bm + Hb^3\right)\ ops,\ M = O\left(\frac{fmn}{P} + sbm\right)\ words\ of\ memory.$$

Communication costs

$$W = O\left(Hbm\log P\right)\ words\ moved,\ L = O\left(\frac{H}{s}\log P\right)\ messages.$$

Proof. The CA-BDCD algorithm begins by randomly selecting sb rows of A and subsequently computing the $sb \times m$ kernel matrix, G. Computing G requires $\frac{fsbmn}{P}$ pair-wise vector operations. Each processor computes a partial kernel matrix which requires communication to aggregate. The communication costs sbm words communicated in $\log P$ messages. Once the sb rows of the kernel matrix are communicated, we kernelize it by performing entry-wise operations, which requires μsbm operations. Once the kernel matrix is computed, we can compute r_{sk+1} by performing a matrix-vector multiply with bm entries from the kernel matrix and α_{sk}, which requires bm operations. After which $\Delta\alpha_{sk+1}$ can be computed with b^3 computations. Once the first inner iteration is computed, the residual r_{sk+j} for $j = 2,\dots s$ becomes stale and requires correction. Correcting the residual requires the matrix vector products $\left(\sum_{t=1}^{j-1}\mathbb{I}_{sk+j}^T\Phi(A)\Phi(A)^T\mathbb{I}_{sk+t}\Delta\alpha_{sk+t}\right)$. When $j = s$, we will have performed $s(s-1)$ matrix vector products each of which cost b^2 operations. In total, the corrections cost an extra $s(s-1)b^2$ operations. Finally, the solution vector α_{sk+j} can be updated with sb operations. Note, however, that CA-BDCD requires $\frac{H}{s}$ outer iterations in order to obtain the same solution as BDCD. Therefore, $\frac{H}{s}$ iteration of CA-BDCD cost $O\left(\frac{Hfbmn}{P} + H\mu bm + Hb^3\right)$ operation, $O\left(Hbm\log P\right)$ words moved in $O\left(\frac{H}{s}\log P\right)$ messages. Each processor requires enough memory to store the m-dimensional vectors α and the labels vector y, the input matrix A, and sbm entries of the kernel matrix. This costs costs $\frac{fmn}{P} + sbm + 2m = O\left(\frac{fmn}{P} + sbm\right)$ words of memory per processor. □

Theorem 6.2.4. *H iterations of the Communication-Avoiding Dual Coordinate Descent (CA-DCD) algorithm for kernel SVM with the matrix $A \in \mathbb{R}^{m\times n}$ stored in $1D$-block column partitions on P processors along the critical path costs*

$$F = O\left(\frac{Hfmn}{P} + H\mu m\right)\ ops,\ M = O\left(\frac{fmn}{P} + sm\right)\ words\ of\ memory.$$

Communication costs

$$W = O\left(Hm \log P\right) \ words\ moved,\ L = O\left(\frac{H}{s} \log P\right) \ messages.$$

Proof. The CA-DCD algorithm begins by randomly selecting b rows of A and subsequently computing the $s \times m$ kernel matrix, G. Computing G requires $\frac{fsmn}{P}$ pair-wise vector operations to compute partial entries on each processor. The entries can be aggregate through an all-reduce with summations which costs sm words communicated in $\log P$ messages. Once the s rows of the kernel matrix are communicated, we kernelize it by performing entry-wise operations, which requires μsm operations. Note that we then need to perform entry-wise multiplications by the label vector, y, which costs sm operations. The gradient, g_{sk+1}, can be computed by performing an inner-product with m entries from the kernel matrix and α_{sk}, which requires m operations. After which computing θ_{sk+1} only requires scalar computations. Once the first inner iteration is computed, the solution vector α_{sk} becomes stale and requires correction. Correcting the solution vector requires scalar products $\left(\sum_{t=1}^{j-1} \mathbb{I}_{sk+j}^T \Phi(A)\Phi(A)^T \mathbb{I}_{sk+t} \Delta\alpha_{sk+t}\right)$. When $j = s$, we will have performed $s(s-1)$ scalar products. In total, the corrections cost an extra $s(s-1)$ operations. Finally, the solution vector α_{sk+j} can be updated with s operations. Note, however, that CA-DCD requires $\frac{H}{s}$ outer iterations in order to obtain the same solution as DCD. Therefore, $\frac{H}{s}$ iteration of CA-DCD cost $O\left(\frac{Hfmn}{P} + H\mu m\right)$ operation, $O\left(Hm \log P\right)$ words moved in $O\left(\frac{H}{s} \log P\right)$ messages. Each processor requires enough memory to store the m-dimensional vectors α and the labels vector y, the input matrix A, and sm entries of the kernel matrix. This costs costs $\frac{fmn}{P} + sm + 2m = O\left(\frac{fmn}{P} + sm\right)$ words of memory per processor. □

The communication-avoiding variant that we derived requires a factor of s fewer messages than their classical counterparts, but the same computation and bandwidth costs. This is in contrast to the previous chapters where the computational and bandwidth costs increase by a factor of s. Note, however, that the storage costs increase for the CA-BDCD and CA-DCD algorithms. When computing $\mathbb{I}_h^T A A^T \alpha_{h-1}$ we can introduce an auxiliary vector $x \in \mathbb{R}^n$ such that we now require two computations: the matrix-vector product $\mathbb{I}_h^T A x_{h-1}$ and a vector update for x_h. However, kernel methods cannot take advantage of this because $\Phi(A)$ has a large (possibly infinite) number of features. Due to this we must compute $\mathbb{I}_h^T \Phi(A)\Phi(A)^T$ explicitly at the cost of additional computation and bandwidth. CA-BDCD and CA-DCD simply increase the rows by a factor of s and computing $G = \left[\mathbb{I}_{sk+1}^T, \mathbb{I}_{sk+1}^T, \ldots, \mathbb{I}_{sk+s}^T\right] \Phi(A)\Phi(A)$. No additional computation nor bandwidth is required to compute $\mathbb{I}_{sk+j}^T \Phi(A)\Phi(A)^T \mathbb{I}_{sk+t}$ (used to correct the residual/gradient), since we can extract them from G. The additional computation and bandwidth costs for kernel methods suggest that latency is less dominant.

6.3 Convergence Behavior

The recurrence unrolling results in the computation of an $sb \times m$ kernel matrix, whose condition number may adversely affect numerical stability. So, we begin by verifying the stability of CA-BDCD for kernel ridge regression and CA-DCD for kernel SVM through MATLAB experiment.

Summary of datasets					
Name	Features (n)	Data Points (m)	f	σ_{min}	σ_{max}
abalone	$4,177$	8	1	$4.3e{-5}$	$2.3e4$
w1a	$2,477$	300	0.04	$1.6e{-4}$	$6.2e3$
breast-cancer	683	10	1	$9.5e{-16}$	$1.1e15$

Table 6.1: Properties of the LIBSVM datasets used in our numerical stability experiments.

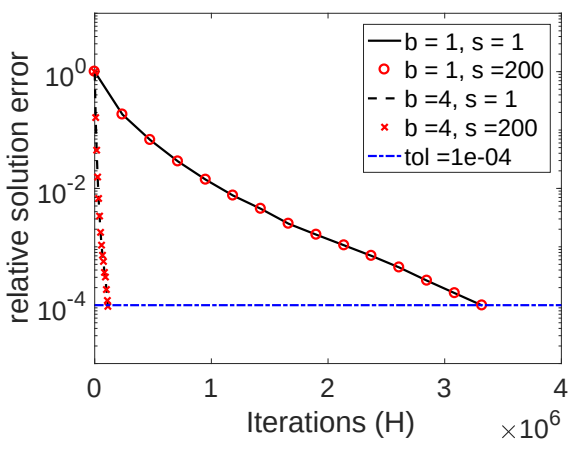
The numerical stability experiments are conducted using MATLAB R2016b on a 2.3GHz Intel i7 machine with 8GB of DRAM. We use datasets (summarized in Table 6.1) from the LIBSVM [26] repository. We measure the convergence behavior for kernel ridge regression by plotting the solution error, $\frac{||\alpha_{opt}-\alpha_h||_2}{||\alpha_{opt}||_2}$. The optimal solution, α_{opt}, is obtained by computing the full $(m \times m)$ kernel matrix and solving for α exactly. For kernel SVM we compute the duality gap, $P(\alpha_h) - D(\alpha_h)$, where $P(\alpha_h)$ is the primal objective value and $D(\alpha_h)$ is the dual objective value. Due to strong convexity, primal and dual linear SVM have the same optimal function value [68, 97]. We set $\lambda = 10\sigma_{min}$ for kernel ridge regression and $\lambda = 1$ for kernel SVM. We show results using the polynomial kernel with $d = 2$ and the radial basis function. Figure 6.2 illustrates the convergence behavior of CA-BDCD and BDCD for the abalone and w1a datasets for two block sizes $b = 1$ and $b = 4$. For CA-BDCD we use $s = 200$ which is large enough that we would expect numerical instability to occur. Regardless of which kernel function is used, we can observe that $b = 4$ converges faster than $b = 1$, as expected. However, the faster convergence is at the expense of additional computation and bandwidth. Therefore, b should be chosen to balance the computation and communication costs. CA-BDCD with $s = 200$ matches the convergence of BDCD for both settings of b. This shows that CA-BDCD is numerical stable and suggests that the computation and communication tradeoff is the limiting factor for large values of s. Figure 6.3 shows the convergence behavior of CA-DCD and DCD for the breast-cancer and w1a datasets in Table 6.1. We show the convergence for the SVM-L1 and the smoothed SVM-L2 loss functions. Note that SVM-L2 does not converge faster than SVM-L1 as observed in Chapter 5 for the linear kernel (however, both loss functions converge to the desired tolerance). Once again we set $s = 200$ for CA-DCD and plot the convergence. As expected, the convergence behavior of CA-DCD matches that of DCD and is numerically stable at $s = 200$. Therefore, the value of s should chosen to balance the computation and communication costs.

6.4 Predicted Performance Results

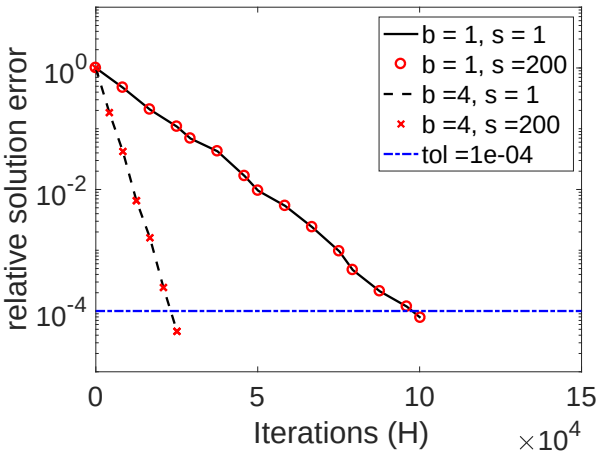
In this section, we show the predicted performance of BDCD, DCD, and their CA variants. We use the running time model (introduced in Chapter 1),

$$T = \gamma F + \beta W + \alpha L,$$

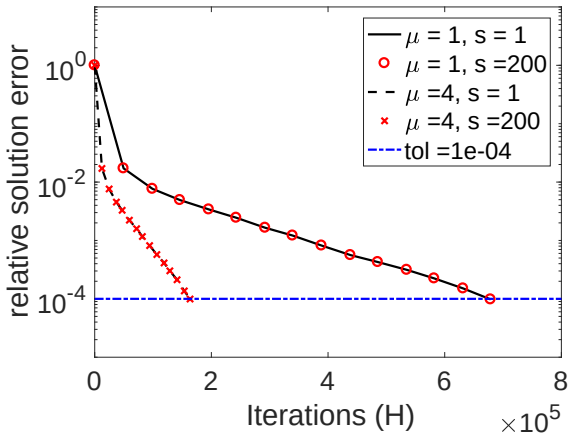
where T is the running time, F is the computational cost, W is the bandwidth cost, and L is the



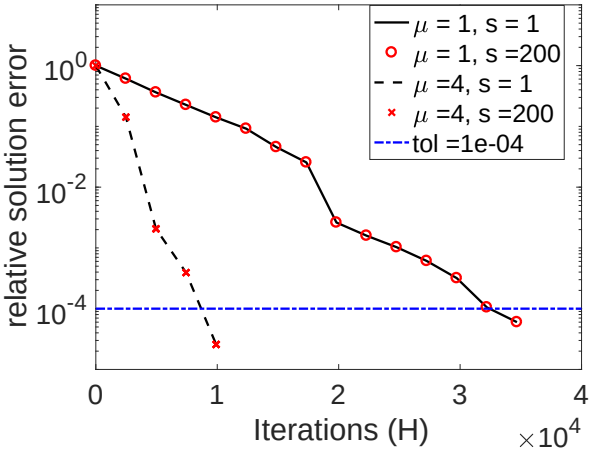
(a) abalone, Poly. kernel, $d = 2$.



(b) abalone, RBF kernel, $\gamma = 1$.



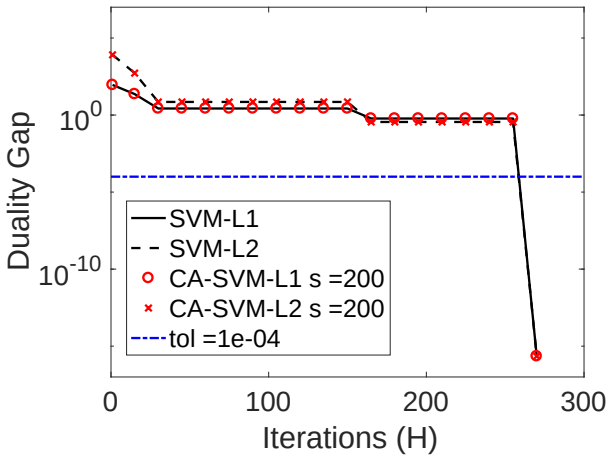
(c) w1a, Poly. kernel, $d = 2$.



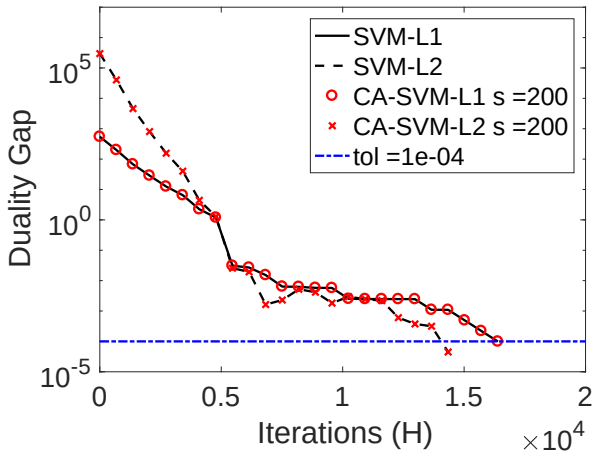
(d) w1a, RBF kernel, $\gamma = 1$.

Figure 6.2: Relative solution error vs. iterations of solving the kernel ridge regression problem using BDCD and CA-BDCD with $s = 200$.

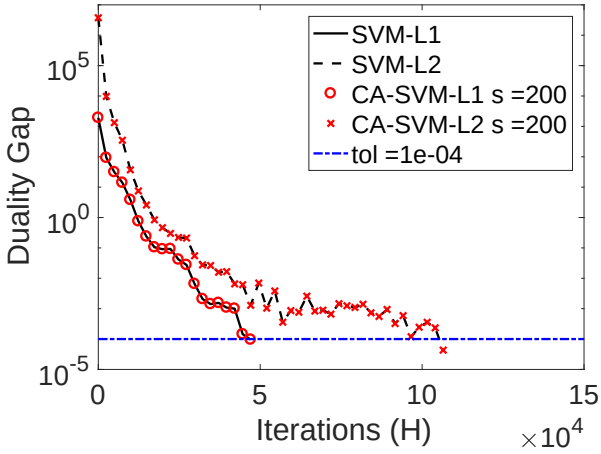
number of messages of the algorithm obtained. The parameters γ (seconds per flop) is the inverse of the flops speed of the hardware, β (seconds per byte) is the inverse bandwidth of the network, and α (seconds per message) is the latency of the network. We use the algorithm costs derived in Section 6.3 when modeling the running time. The results we obtain show the per iteration running time and speedups. We assume that kernel ridge regression and kernel SVM use the radial basis function. This requires an exp operation which has $\mu = 3$ (i.e. $3\times$ slower than a multiply instruction) [50, Knights Landing]. We show results for three platforms: Exascale with MPI,



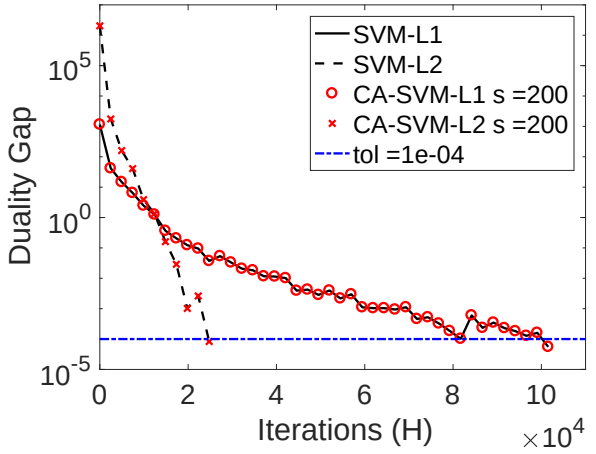
(a) breast-cancer, Poly. kernel, $d = 2$.



(b) breast-cancer, RBF kernel, $\gamma = 0.1$.



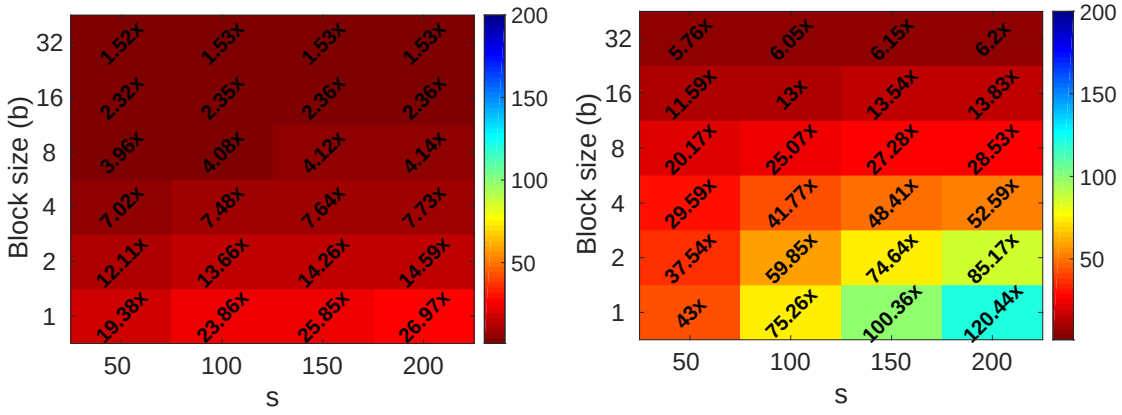
(c) w1a, Poly. kernel, $d = 2$.



(d) w1a, RBF kernel, $\gamma = 0.1$.

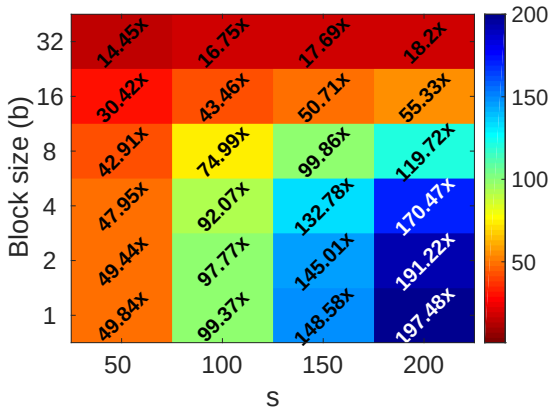
Figure 6.3: Duality gap vs. iterations of DCD and CA-DCD with $s = 200$.

Exascale with Spark, and Cloud with Spark. The hardware costs for the Exascale platform are $\gamma = 10e{-}13$, $\beta = 3.7e{-}10$, and $\alpha = 5e{-}7$ [25]. For Exascale with Spark we assume that the computation and bandwidth speeds remain the same but increase the latency cost to $\alpha = 5e{-}6$ where Spark was shown to be an order of magnitude slower than MPI in terms of latency for iterative algorithms [56]. Finally, for Spark on Cloud we assume that the CPUs are the same as Exascale, but replace the network with gigabit ethernet which has $\beta = 8e{-}9$ and $\alpha = 1.25e{-}3$. Note that we assume that the processors can attain peak γ performance and neglect sequential costs (i.e. communication between DRAM and caches, etc.).



(a) MPI on Exascale.

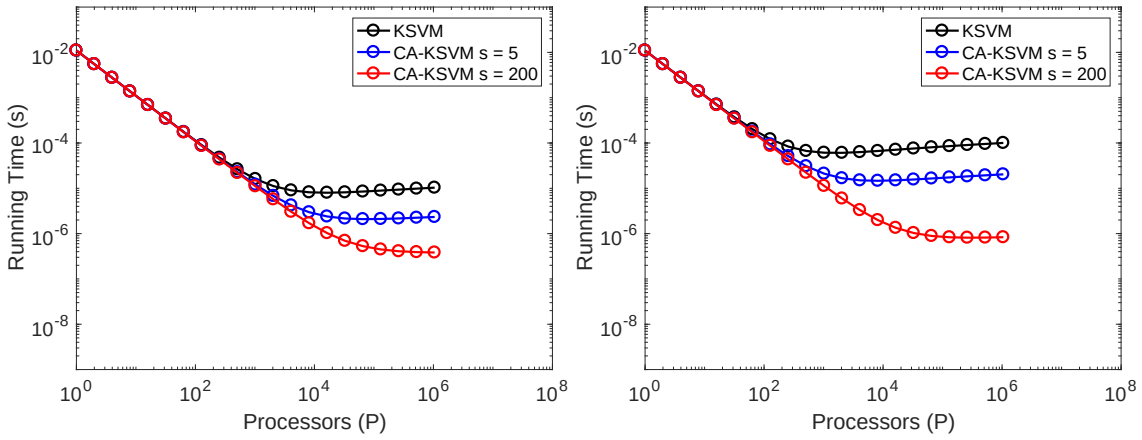
(b) Spark on Exascale.



(c) Spark on Cloud.

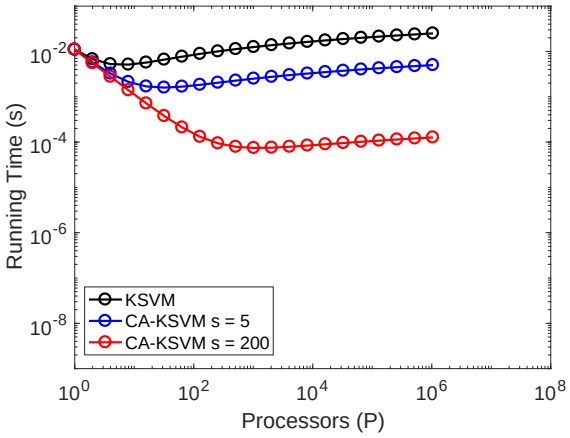
Figure 6.4: Speedups obtained for BDCD and CA-BDCD for various settings of b and s.

For kernel ridge regression we use a matrix with $m = 2^{20}$ data points, $n = 2^{20}$ features and run on $P = 2^{20}$ processors, such that each processor stores one feature. The matrix is assumed to be sparse with $f = 0.1$. We explore the performance of several values of block size, $b = 1, 2, 4, 8, 16, 32$ and $s = 50, 100, 150, 200$. Since convergence depends on the condition of the matrix, we normalize the running times of CA-BDCD relative to BDCD with matching block sizes and omit BDCD (i.e. the $s = 1$ column) since the speedups are $1\times$. Figure 6.4 shows the modeled speedups for the three platform and programming model combinations. For Exascale, where latency times are fast we observe speedups of $26\times$ for $b = 1$ and $s = 200$. Speedups decrease as b increases since computation and bandwidth costs become more dominant. Exascale with Spark attains speedups of up to $120\times$ due to the higher latency costs. Finally, Cloud with Spark achieves the highest speedups of all configurations with $s = 200$ performing up to $197\times$



(a) MPI on Exascale.

(b) Spark on Exascale.



(c) Spark on Cloud.

Figure 6.5: Strong scaling of DCD for kernel SVM and CA-DCD with $s = 20$, and $s = 200$.

faster on a gigabit ethernet network.

For kernel SVM we show modeled strong scaling results using the same matrix for the number of processor ranging from $P = 1$ to $P = 2^{20}$ by doubling. Note that the strong scaling and speedup results are independent of the loss functions, SVM-L1 and SVM-L2, because we show per iteration running times. The two loss functions only differ in scalar quantities so they have the same algorithm costs. At $P = 2^{20}$ we observe speedups of $26\times$, $120\times$, and $197\times$ for Exascale with MPI, Exascale with Spark, and Cloud with Spark, respectively. As expected, larger latency costs result in DCD losing scalability at smaller numbers of processors. On the other hand, CA-DCD scales much further and attains larger speedups due to large latency costs.

6.5 Conclusions and Future Work

In this chapter we derived communication-avoiding algorithms to solve the kernel ridge regression and kernel SVM problems. We derived asymptotic bounds on computation, communication, and storage. We illustrated with the analysis that the CA-variant reduces latency cost by a tunable factor s while maintaining the same computation and bandwidth costs. We showed that these methods are numerically stable even for very large values of s. Furthermore, we showed modeled performance results that showed that the CA-variants scale better and can achieve large modeled speedups up to $197\times$ on three different platforms and programming model combinations.

There are several directions for future work. While we have derived CA-variants, we have only explored modeled performance. As a result, implementation of these algorithms and exploring the practical tradeoffs is important. While we focused on (block) dual coordinate descent deriving CA-variants of stochastic gradient/sub-gradient descent is also a fruitful direction. Comparing the performance of CA-BDCD and CA-DCD with other approaches of reducing communication in kernel methods would be fruitful. Introducing caching schemes to reduce the amount of computation required to computing b rows of the kernel matrix is worth exploring. Caching could potentially eliminate the computation and bandwidth costs for some iterations. In this situation latency dominates and the CA-variants could yield greater speedups.

Chapter 7

Adaptive Batch Sizes for Deep Neural Networks

The AdaBatch technique and experimental results in this chapter are a product of joint work with co-authors Maxim Naumov and Michael Garland during an internship at NVIDIA. This work has appeared as a technical report on arXiv [44].

The optimization problems explored thus far in this thesis were limited to the convex case. However, many of the problems solved by deep neural networks are non-convex. In this chapter, we introduce an adaptive batch size (AdaBatch) technique that reduces the cost of communication when training deep neural networks. The contributions of this chapter are:

- Illustrating that adaptively increasing batch size can augment or replace learning rate decay.
- Single-GPU and multi-GPU performance evaluation of a PyTorch implementation of our technique compared to fixed batch size techniques on up to 4 NVIDIA P100 GPUs. Our results show that we can attain speedups of $3.54\times$ on VGG19 network and $6.25\times$ on ResNet-20 network using the CIFAR-100 dataset without changing the test error.
- Training the ImageNet dataset on the ResNet-50 network with a batch size of up to $524,228$, which would allow neural network training to attain a higher fraction of peak GPU performance than training with smaller batch sizes.

Deep neural networks (DNNs) have a long and well-documented history. Their resurgence due, in part, to hardware improvements has led to rapid progress in several research areas; most notably, in speech recognition and computer vision. While modern hardware has made the use of deep neural networks practical, running time remains a bottleneck. Graphics Processing Units (GPUs) have become particularly attractive in training DNNs due to their high computational throughput and high flops to watts ratio. The main computational kernels for DNNs are dense convolutions and dense matrix multiplications, which are well-suited for the GPU architecture.

Neural networks differ from the models (i.e. regularized least-squares, SVM, logistic regression, etc.) described in previous chapters in that neural networks utilize additional “hidden" layers

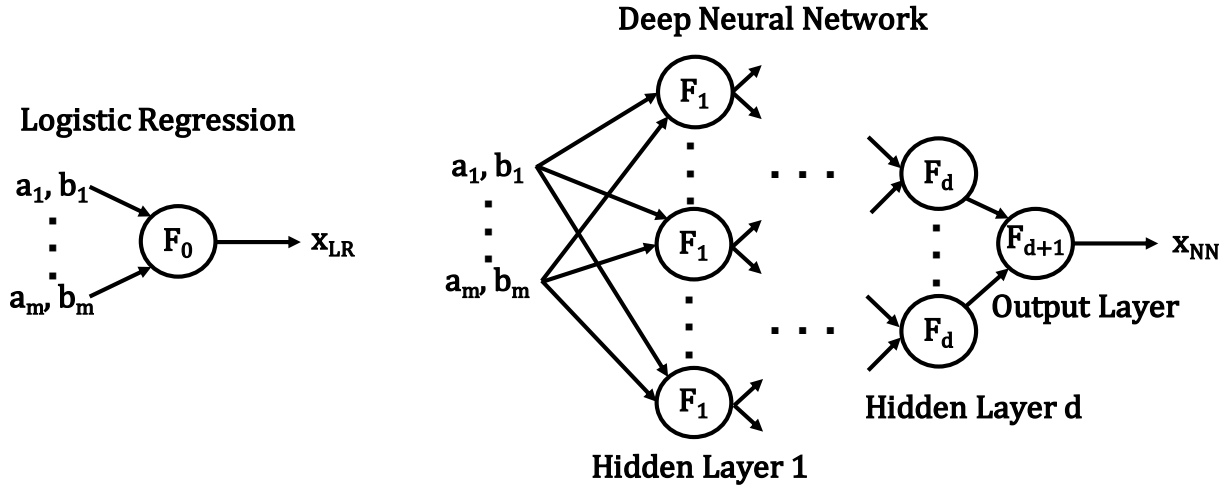


Figure 7.1: Comparison of the input-output transformation between logistic regression and a generic deep neural network with d hidden layers. a_i is the i-th sample from the input dataset,A, and b_i is its corresponding label. F_0 is the nonlinear logistic function, F_j for $1 \leq j \leq d$ are the nonlinear activation functions for the j-th hidden layer, and F_{d+1} is the nonlinear output function. x_{LR} and x_{NN} are the solutions obtained from the logistic regression and DNN model, respectively. Each node in the hidden layer is known as a hidden unit or a neuron.

between the input and output layers. Figure 7.1 illustrates the input-output transformations for logistic regression and a simple, deep neural network[1] with d hidden layers. Suppose that Stochastic Gradient Descent (SGD) is used to solve the logistic regression (LR) problem. In this setting, the gradient of the loss function for LR is computed using a randomly chosen mini-batch of samples and then the solution, x_{LR}, is updated. This process is repeated until a termination criterion is met.

On the other hand, solving the DNN requires a more complicated optimization technique known as *backpropagation* [104]. The hidden layers introduce intermediate inputs and outputs, therefore computing the gradient of the DNN requires two passes over the network. A *forward pass* is required, which begins with a mini-batch of samples (at the input layer) and transforms them into the output of hidden layer 1 using nonlinear function F_1. The output from hidden layer $d-1$ is similarly transformed into the output of hidden layer d until the result of the output layer is computed. A *backward pass* is then performed which begins at the output layer by computing the gradient of an error function (e.g. mean-squared error) with respect to the true labels and the computed labels. By applying the chain-rule, the gradient at each hidden layer can be computed and the hidden layer weights can be updated using the SGD update rule. Once the weights at hidden layer 1 are updated, the next forward pass can begin. This process is repeated until a termination criterion is met.

[1]The DNN depicted is a feedforward neural network. Also known as a Multi-Level Perceptron (MLP).

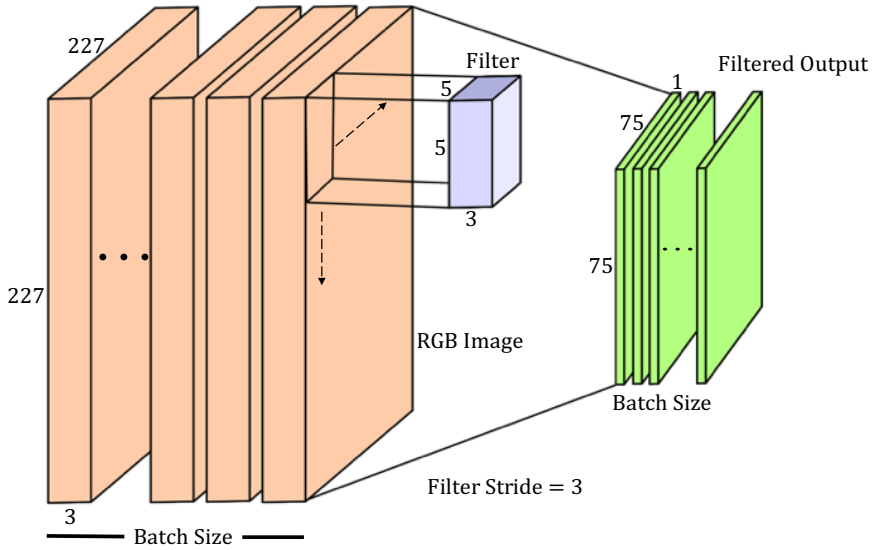


Figure 7.2: This convolution layer applies a $5 \times 5 \times 3$ convolution filter to each $227 \times 227 \times 3$ image and results in a batch of $75 \times 75 \times 1$ filtered output. “Filter Stride" is the number of pixels the filter is shifted up or down before applying the filter. In general, convolution layers apply several filters so the filters and filtered output are typically 4-D tensors.

Convolutional Neural Networks

Feedforward neural networks (FNNs), as illustrated in Figure 7.1, are characterized by fully-connected layers. In other words, the output of each hidden unit is an input to every hidden unit in the next hidden layer. In addition to fully-connected layers, convolutional neural networks (CNNs) also consist of convolutional layers, pooling layers, and normalization layers. The use of convolutional layers makes CNNs particularly well-suited for image classification problems in computer vision. Figure 7.2 illustrates a typical convolutional or pooling layer which takes a mini-batch of images[2] and performs convolutions or a pooling operation (like max-pooling) using a series of filters to obtain a series of filtered output. The filtered output is then passed to the next layer as the input and so on until the output layer. In general, several filters are applied at each layer resulting in 4-D tensors as filtered output.

Adaptive Batch Sizes

Stochastic Gradient Descent (SGD) and its variants are the most widely used optimization methods for training deep neural networks. Training a neural network requires large amounts of data, therefore, implementations typically divide the training set into a series of batches of some fixed batch size. Each batch is processed in sequence during training; however, the individual training

[2]We will use “batch size" to refer to the number of images in a mini-batch.

samples within a single batch may be processed in parallel [12, 59]. One pass over the batches (i.e. the entire dataset) is referred to as an epoch.

The training process typically uses a static batch size, r, which is held constant throughout training. However, static batch sizes force the user to resolve an important conflict. On one hand, small batch sizes are desirable since they tend to perform better in terms of test accuracies [35, 70]. On the other hand, large batch sizes offer more data-parallelism which in turn improves computational efficiency and scalability [60, 123].

This trade-off between small and large batch sizes can be resolved by *adaptively* increase the batch size during training. We begin with a batch size r, typically the same batch size used for fixed batch size training, and progressively increase the batch size between selected epochs as training proceeds. We double the batch size at specific intervals and simultaneously adapt the learning rate α so that the ratio α/r remains constant. The adaptive batch size technique has several advantages. It delivers the accuracy of training with small batch sizes, while improving performance by increasing the amount of work available per processor in later epochs. Furthermore, the large batches used in later epochs expose sufficient parallelism to create the opportunity for distributing work across many processors, where those are available. Our approach can also be combined with other existing techniques for constructing learning rate decay schedules to increase batch sizes even further.

We have applied our adaptive batch size method to training standard AlexNet, ResNet, and VGG networks on the CIFAR-10, CIFAR-100, and ImageNet datasets. The experimental results, detailed in Section 7.3, demonstrate that training with adaptive batch sizes attains similar test accuracies with faster running times compared with training using fixed batch sizes. Furthermore, we experimentally show that adaptive batch sizes can be combined with other large batch size techniques to yield speedups of up to $6.25\times$ while leaving test accuracies relatively unchanged.

7.1 Related Work

Previous work introduced gradual learning rate warmup and linear learning rate scaling in order to attain batch sizes of 8192 for ImageNet CNN training in a large, distributed GPU cluster setting [60]. More recently, the use of a layer-wise learning rate scaled by the norms of the gradients allowed even higher batch sizes of 32,768 [123]. Both results use a fixed batch size throughout training whereas our work changes the batch sizes during training. Furthermore, we show that our work is complementary to these existing results.

The relationship between adaptive batch sizes and learning rates is well-known. In particular, this relationship was illustrated for strongly-convex, unconstrained optimization problems by Friedlander & Schmidt [53]. This work showed that batch size increases can be used instead of learning rate decreases. On the other hand, a batch size selection criterion based on estimates of gradient variance, which are used to adaptively increase batch sizes, was introduced by Byrd *et al.* [19]. Both approaches consider second-order, Newton-type methods. Our work complements and adds to this research by exploring adaptive batch sizes for various neural network architectures.

Several authors have also studied adaptively increasing batch size with fixed schedules from the context of accelerating the optimization method through variance reduction [34, 63]. Both works study the theoretical and empirical convergence behavior of their adaptive batch size optimization methods on convex optimization problems. In contrast an adaptive criterion to control the batch size increases and illustration of their convergence on convex problems and convolutional neural networks is developed by De *et al.* [38] and Balles *et al.* [6]. While both illustrate the practicality of coupling adaptive batch sizes with learning rates, they do not explore the performance benefits that can be gained through the use of adaptive batch sizes when coupled with existing large batch size CNN training work [60, 123].

A very recent work illustrates that learning rate decay can be replaced with batch size increase [110]. The aforementioned study shows that the batch size increase in lieu of learning rate decay works on several optimization algorithms: SGD, SGD with momentum, and Adam. Furthermore, it experiments with altering the momentum term with batch size increases and explores the effects on convergence.

In addition, our research explores the performance tradeoffs from using an adaptive batch size technique on popular CNNs and illustrates that our technique is complementary to existing fixed, large batch size training techniques. Our results also independently verify that adaptive batch size can practically replace learning rate decay and lead to performance improvements. In particular, we show that adaptive batch size schedules can yield speedups that learning rate schedules alone cannot achieve. Finally, by combining adaptive batch sizes with large batch size techniques we show that even larger speedups can be achieved with similar test error performance.

7.2 Adaptive Batch Sizing and its Effects

The batch size and learning rate are intimately related tuning parameters [123, 60]. Recent work has shown that adapting the learning rate either layerwise or over several iterations can enable training with large batch sizes without sacrificing accuracy. We will illustrate that the relationship between batch size and learning rate extends even further to learning rate decay. The following analysis provides the basis for our adaptive batch size technique.

Learning Rate

In supervised learning, the training of neural networks consists of repeated forward and backward propagation passes over a labelled data set. The data set is often partitioned into training, validation and test parts, where the first is used for fitting the neural network function to the data and the others for verification of the results.

Let a training data set be composed of data samples $\{(\mathbf{x}, \mathbf{z}^*)\}$, which are pairs of known inputs $\mathbf{x} \in \mathbb{R}^n$ and outputs $\mathbf{z}^* \in \mathbb{R}^m$. Further, let these pairs be ordered and partitioned into q disjoint batches of size r. For simplicity of presentation, we assume that the number of pairs is qr. In cases where r does not divide the number of pairs evenly, implementations must in practice either pad the last batch or correctly handle truncated batches.

Training a neural network with weights W can be interpreted as solving an optimization problem

$$\arg\min_{W} \mathcal{L} \tag{7.1}$$

for some choice of loss function $\mathcal{L}$. This optimization problem is often solved with a stochastic gradient descent algorithm, where the weight updates at i-th iteration are performed using the following rule

$$W_{i+1} = W_i - \frac{\alpha}{r}\Delta W_i \tag{7.2}$$

for an update matrix ΔW_i computed with batch-size r and learning rate α.

The batch size and learning rate are independent tuning parameters. However, notice that factor $\frac{\alpha}{r}$ is proportional to batch size and inversely proportional to the learning rate. This suggests that we may be able to augment learning rate decay with batch size increases. We will now illustrate that augmenting learning rate decay with batch size increase is feasible (in the sense that convergence behavior does not change significantly). According to Equation (7.2) after q iterations (i.e., one epoch) with a learning rate α and batch size r we have

$$W_{i+q} = W_i - \frac{\alpha}{r}\sum_{i=1}^{q}\Delta W_i \tag{7.3}$$

Suppose that we instead train with larger batch sizes by grouping $\beta > 1$ batches. Note that this results in an effective batch size of βr and results in $\tilde{q} = q/\beta$ iterations for one epoch of training. Under this setting we can write

$$W_{i+\tilde{q}} \;=\; W_i - \frac{\tilde{\alpha}}{\beta r}\sum_{j=1}^{\tilde{q}}\Delta\widetilde{W}_j \tag{7.4}$$

for an update matrix $\widetilde{W}_j$ computed with batch size βr and learning rate $\tilde{\alpha}$. Notice that this can be re-written as an accumulation of β gradients with batch size r as follows:

$$W_{i+\tilde{q}} \;=\; W_i - \frac{\tilde{\alpha}}{\beta r}\sum_{j=1}^{\tilde{q}}\left(\sum_{k=1}^{\beta}\Delta W_{i'}\right) \tag{7.5}$$

where $\widetilde{W}_j = \sum_{k=1}^{\beta}\Delta W_{i'}$ and index $i' = (j-1)\beta + k$. Notice that W_{i+q} might be similar to $W_{i+\tilde{q}}$ only if we set the learning rate $\alpha = \tilde{\alpha}/\beta$ and assume that updates $\Delta W_i \approx \Delta W_{i'}$ are similar in both cases. This assumption was empirically shown to hold for fixed large batch size training with gradual learning rate warmup [60] after the first few epochs of training.

Notice that the factor $1/\beta$ can be interpreted as a learning rate decay, when comparing Equation (7.3) and (7.5). This relationship has been used to justify linearly scaling the learning rate for large batch size training [60]. On the other hand, we take advantage of this relationship to illustrate that increasing the batch size can mimic learning rate decay. Naturally, the two approaches can be combined as we will show in Section 7.3. In our experiments, we will consider adaptive batch sizes

which increase according to a fixed schedule. We ensure that the effective learning rates ($\alpha = \tilde{\alpha}\beta$) for fixed batch size vs. adaptive batch size experiments are fixed throughout the training process for fair comparison (see Section 7.3 for details).

Test Accuracy and Performance

Training with large batch sizes is attractive due to its performance benefits on modern deep learning hardware. For example, on GPUs larger batch sizes allow us to better utilize GPU memory bandwidth and improve computational throughput [93, 94]. Larger batch sizes are especially important when distributing training across multiple GPUs or even multiple nodes since they can hide communication cost more effectively than small batch sizes.

However, the performance benefits of large batch sizes come at the cost of lower test accuracies since large batches tend to converge to sharper minima [35, 70]. Through the use of learning rate scaling [60] and layer-wise adaptive learning rates [123], larger batch sizes can attain better accuracies. While both approaches increase the batch size, the batch size they use remains fixed throughout training. We propose an approach that adaptively changes the batch size and progressively exposes more parallelism. This approach also allows one to progressively add GPUs, if available, to the training process.

Work per Epoch

Fixed batch sizes (small or large) require a fixed number of floating point operations (flops) per iteration throughout the training process. Since our technique adaptively increases the batch size, the flops per iteration progressively increases. Despite this increase, we can show that the flops per *epoch* remains fixed as long as the computation required for forward and backward propagation is a linear function of the batch size r.

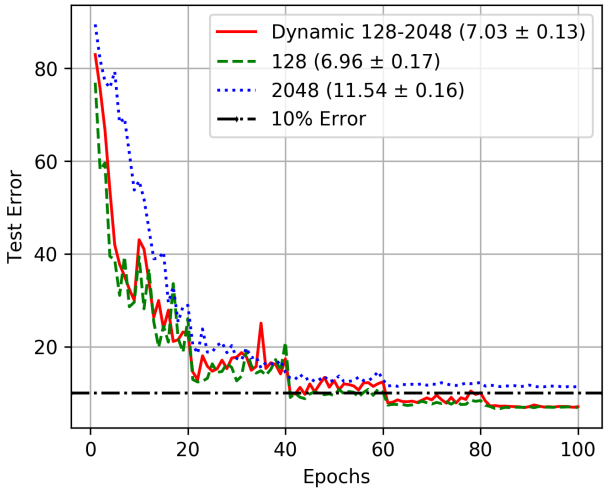
For example, let us briefly illustrate this point on a fully connected layer with a weight matrix $W \in \mathbb{R}^{m \times n}$, input $X = [\mathbf{x}_1, ..., \mathbf{x}_r] \in \mathbb{R}^{n \times r}$ and error gradient $V = [\mathbf{v}_1, ..., \mathbf{v}_r] \in \mathbb{R}^{m \times r}$. Notice that for a batch of size r the most computationally expensive operations during training are matrix-matrix multiplications

$$Y = WX \qquad \text{and} \tag{7.6}$$

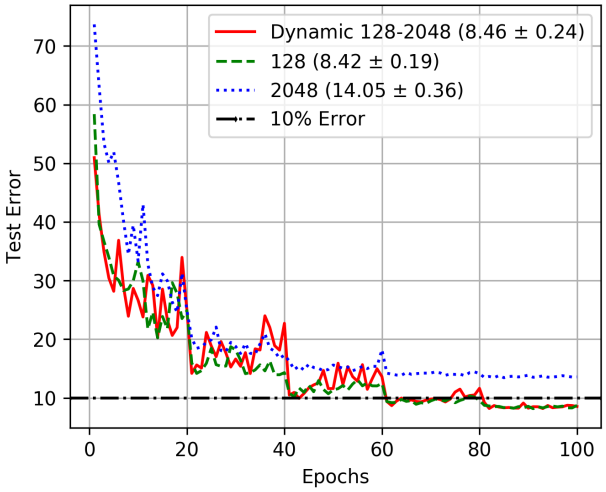
$$U = W^T V \tag{7.7}$$

in forward and backward propagation, respectively. These operations require $O(mnr)$ flops per iteration and $O(mnrq)$ flops per epoch. Notice that the amount of computation depends linearly on the batch size r. If we select a new, larger batch size of βr, then the flops per iteration increase to $O(mn\beta r)$. However, increasing the batch size by a factor of β also reduces the number of iterations required by a factor of β. As a result the flops per epoch remains fixed at $O(mnrq)$ despite requiring more flops per iteration. Since larger batch sizes do not change the flops per epoch, they are likely to result in performance *improvements* due to better hardware efficiency.

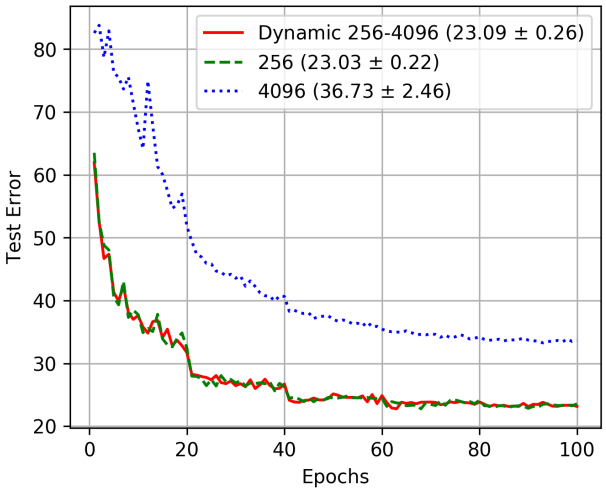
Our experimental results in Section 7.3 confirm these conclusions for CNNs.



(a) VGG19 with Batch Norm.



(b) ResNet-20



(c) AlexNet

Figure 7.3: Comparison of CIFAR-10 test errors for adaptive versus fixed small and large batch sizes. The plots show the lowest test error and report mean ± standard deviation over 5 trials.

7.3 Experimental Results

In this section, we illustrate the accuracy tradeoffs and performance improvements of our adaptive batch size technique. We test this technique using the VGG [109], ResNet [64], and AlexNet [76] deep learning networks on CIFAR-10 [74], CIFAR-100 [75], and ImageNet [42] datasets. We implement our algorithms using PyTorch version 0.1.12 with GPU acceleration. Our experimental platform consists of 4 NVIDIA Tesla P100 GPUs interconnected via NVIDIA NVLink.

Fixed vs. Dynamic Batch Sizes

As we have illustrated in Section 7.1, learning rate decay is a widely used technique to avoid stagnation during training. While learning rate schedules may help improve test error, they rarely lead to faster training times. We begin by performing experiments to validate our claim that adaptive batch sizes can be used without significantly affecting test accuracy. For these experiments, we use SGD with momentum of 0.9, weight decay of 5×10^{-4}, and perform 100 epochs of training. We use a base learning rate of $\alpha = 0.01$ and decay it every 20 epochs. For the adaptive method we decay the learning rate by 0.75 and simultaneously double the batch size at the same 20-epoch intervals. The learning rate decay of 0.75 and batch size doubling combine for an *effective* learning rate decay of 0.375; therefore, we use a learning rate decay of 0.375 for the fixed batch size experiments for the most direct comparison. All experiments in this section are performed on a single Tesla P100.

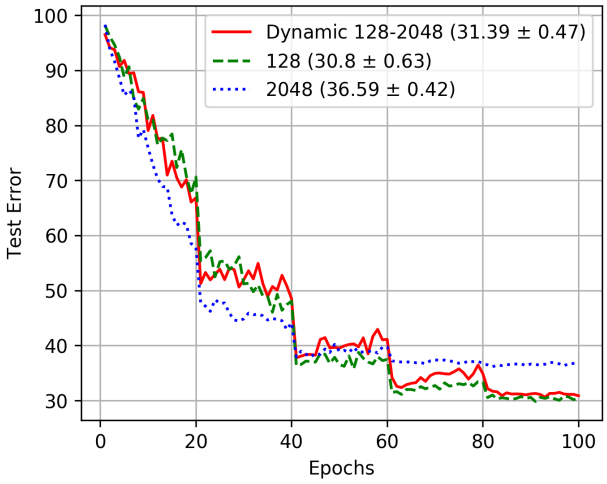
Figure 7.3 shows the test error on the CIFAR-10 dataset for (7.3a) VGG19 with batch normalization, (7.3b) ResNet-20, and (7.3c) AlexNet. For AlexNet the fixed batch sizes are 256 and 4096. For VGG19 and ResNet-20 the fixed batch sizes are 256 and 4096, which are smaller due to the constraint of fitting within the memory of a single GPU. Figure 7.3 plots the best test error for each batch size setting, but reports the mean and standard deviation over five trials in the legends. The noticeable drops in test error every 20 epochs are due to the learning rate decay. We observed that the adaptive batch size technique attained mean test errors within 1% of the smallest fixed batch size. Compared to the largest fixed batch sizes, the adaptive technique attained significantly lower test errors. Note that the fixed batch size experiments for VGG19 and ResNet-20 attain test errors comparable to those reported in prior work [64].

Figure 7.4 shows similar results on the CIFAR-100 dataset for the same networks and batch size settings. Once again, we see that the adaptive batch size technique attains test errors within 1% of the smallest fixed batch size. Our results indicate that learning rate schedules and batch size schedules are related and complementary. Both can be used to achieve similar effects on test accuracy. However, adapting the batch size provides the additional advantage of better efficiency and scalability without the need to sacrifice test error.

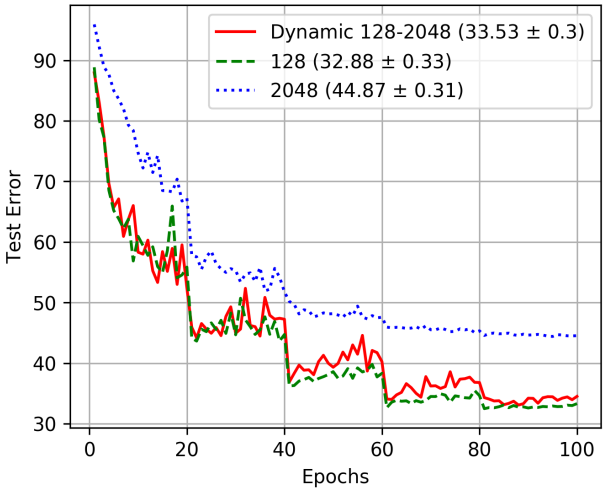
Network	Batch Size	Forward Time (speedup)	Backward Time (speedup)
VGG19_BN	128	933.79 sec. (1×)	1571.35 sec. (1×)
	128-2048	707.13 sec. (1.32×)	1322.59 sec. (1.19×)
ResNet-20	128	256.59 sec. (1×)	661.35 sec. (1×)
	128-2048	218.97 sec. (1.17×)	578.63 sec. (1.14×)
AlexNet	256	66.24 sec. (1×)	129.39 sec. (1×)
	256-4096	44.34 sec. (1.49×)	89.69 sec. (1.44×)

Table 7.1: Comparison of CIFAR-100 forward and backward propagation running time over 100 epochs for adaptive versus fixed batch sizes. The table shows mean over 5 trials.

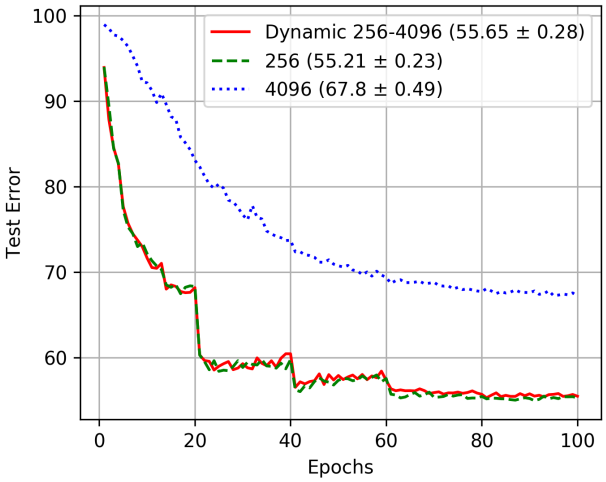
Table 7.1 quantifies the efficiency improvements that come from adaptive batch sizes. We



(a) VGG19 with Batch Norm.



(b) ResNet-20



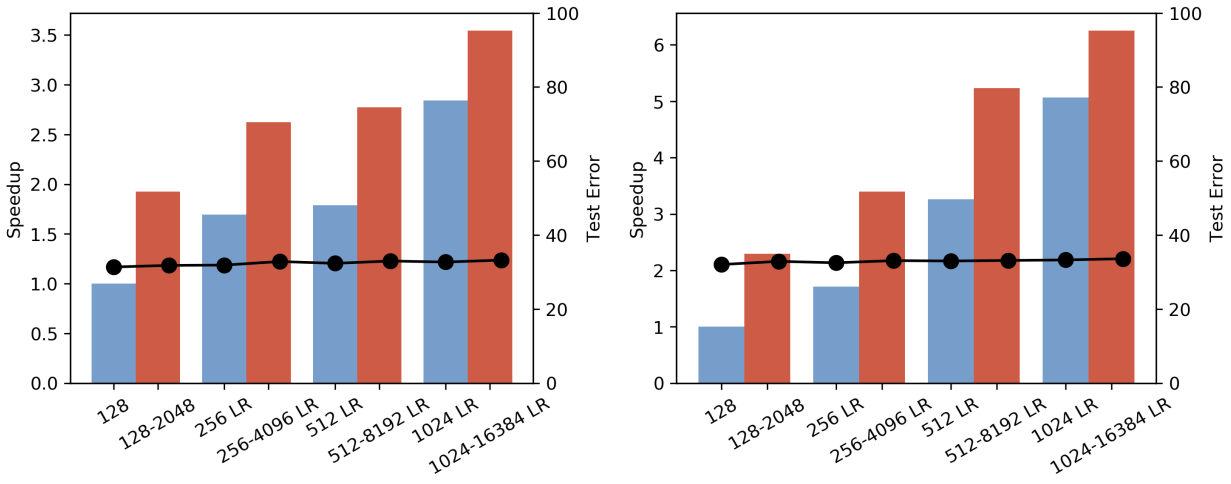
(c) AlexNet

Figure 7.4: Comparison of CIFAR-100 test errors for adaptive versus fixed small and large batch sizes. The plots show the lowest test error and report mean ± standard deviation over 5 trials.

report the running times on the CIFAR-100 dataset over 100 epochs of training. We omit the largest batch sizes from the table since they do not achieve comparable test errors. We also omit CIFAR-10 performance results since it is the same dataset. For all networks tested, we observed that the mean forward and backward propagation running times of adaptive batch sizes were better.

Multi-GPU Performance

While the speedups on a single GPU are modest, the ability to use batch sizes up to 4096 allows for better scalability in multi-GPU settings. As we have illustrated, adaptively increasing the batch



(a) VGG19 with Batch Norm.

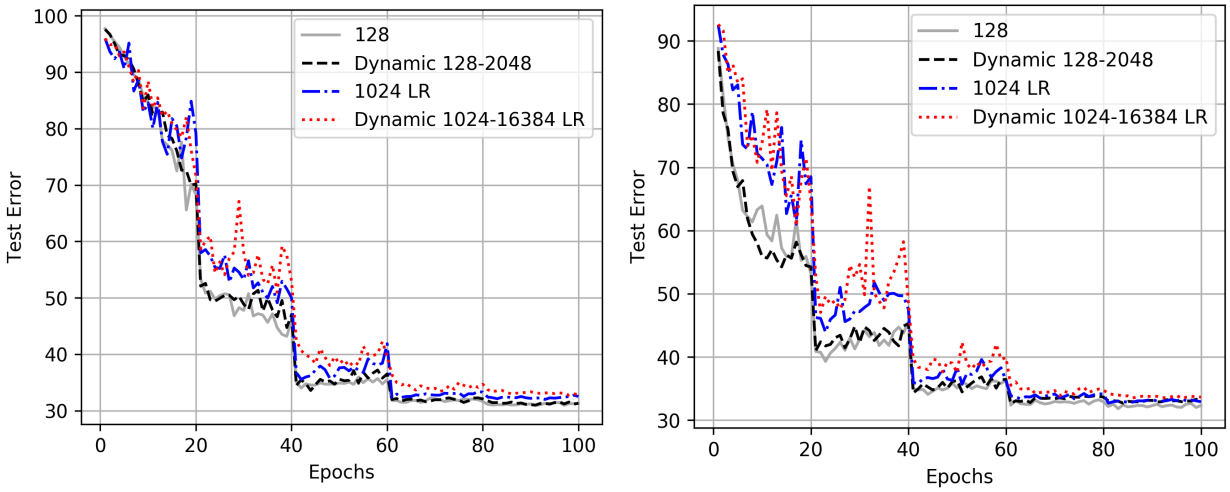
(b) ResNet-20

Figure 7.5: Comparison of CIFAR-100 speedup (left vertical axis) and test errors (right vertical axis) for adaptive (in red) vs. fixed batch sizes (in blue), where "LR" uses gradual learning rate scaling for the first 5 epochs. We also report test error (black dots) to illustrate that it does not change significantly for different combinations of batch sizes and techniques used.

size can act like a learning rate decay. In particular, we have experimentally shown that doubling the batch size behaves similarly to halving the learning rate. This suggests that our adaptive batch size technique can be applied to existing large batch size training techniques [60, 123]. In this section, we will combine the former approach with our adaptive batch size technique and explore the test error and performance tradeoffs. Note that we do not use the latter approach since we would like to ensure our distributed batch sizes fit in each GPU's memory. We use PyTorch's `torch.nn.DataParallel` facility to parallelize across the 4 Tesla P100 GPUs in our test system.

We perform our experiments on CIFAR-100 using VGG19 with Batch Normalization and ResNet-20. We use SGD with momentum of 0.9 and weight decay of 5×10^{-4}. The baseline settings for both networks are fixed batch sizes of 128, base learning rate of 0.1, and learning rate decay by a factor of 0.25 every 20 epochs. The adaptive batch size experiments start with large initial batch sizes, perform gradual learning rate scaling over 5 epochs and double the batch every 20 epochs and decay learning rate by 0.5. We perform 100 epochs of training for all settings.

Figure 7.5 shows the speedups (left vertical axis) and test errors (right vertical axis) on (7.5a) VGG19 and (7.5b) ResNet-20. All speedups are normalized against the baseline fixed batch size of 128. The additional "LR" labels on the horizontal axis indicate settings which require a gradual learning rate scaling in the first 5 epochs. Compared to the baseline fixed batch size setting, we see that adaptive 1024–16384 batch size attains average speedups (over 5 trials) of $3.54\times$ (VGG19) and $6.25\times$ (ResNet-20) with less than 2% difference in test error. Note that the speedups are attained due to the well-known observation that large batch sizes train faster [60, 70, 35]. It is also useful to note that our approach can scale to more GPUs due to the use of progressively larger



(a) VGG19 with Batch Norm.

(b) ResNet-20

Figure 7.6: Comparison of CIFAR-100 test errors curves for adaptive versus fixed batch sizes.

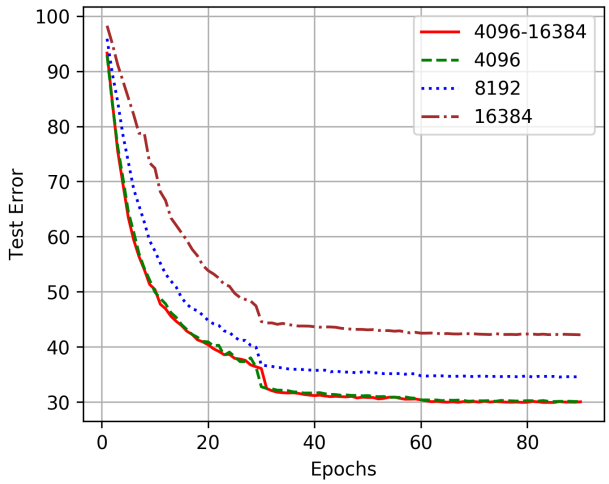
batch sizes.

Figure 7.6 shows the test error curves for 4 batch size settings: fixed 128, adaptive 128–2048, fixed 1024 with learning rate warmup, and adaptive 1024–16384 with learning rate warmup. We report results for VGG19 with batch norm and ResNet-20 on the CIFAR-100 dataset. These experiments illustrate that adaptive batch sizes converge to test errors that are similar ($< 1\%$ difference) to their fixed batch size counterparts. Furthermore, the results indicate that adaptive batch sizes can be coupled with the gradual learning rate warmup technique [60] to yield progressively larger batch sizes during training. We conjecture that our adaptive batch size technique can also be coupled with layer-wise learning rates [123] for even larger batch size training.

ImageNet Training with AdaBatch

In this section, we illustrate the accuracy and convergence of AdaBatch on ImageNet training with the ResNet-50 network. Once again we use PyTorch as the deep learning framework and its `DataParallel` API to parallelize over 4 NVIDIA Tesla P100 GPUs. Due to the large number of parameters, we are only able to fit a batch of 512 in multi-GPU memory. When training batch sizes > 512 we choose to accumulate gradients. For example, when training with a batch size of 1024 we perform two forward and backward passes with batch size 512 and accumulate the gradients before updating the weights. The effective batch size for training is thus 1024, as desired, however the computations are split into two iterations with batch size 512. Due to this gradient accumulation[3] we do not report performance results.

[3]PyTorch supports gradient accumulation which means that we can select a batch size that fits in single- or multi-GPU memory and keep a running sum of gradients for several iterations before adding to the weight matrix and zeroing out the gradient sum buffer. This feature allows us to train with large batch sizes implicitly.



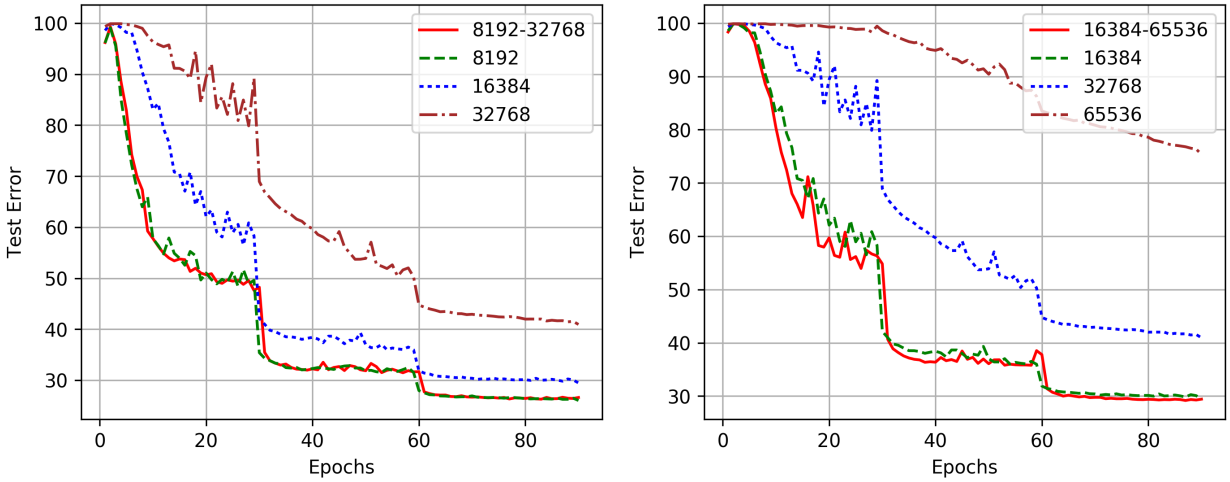
(a) ResNet-50

Figure 7.7: Comparison of ImageNet test errors curves for adaptive versus fixed batch sizes.

Figure 7.7 shows the evolution of the test error over 90 Epochs of training. We train ResNet-50 with a starting learning rate of 0.1 which is decayed by 0.1 every 30 epochs. The network is trained using SGD with momentum of 0.9 and weight decay of 1×10^{-4}. For the AdaBatch experiment we double the batch size and decay the learning rate by 0.2 every 30 epochs. Note that the effective learning rate for AdaBatch and fixed batch sizes is the same for fair comparison. As the results indicate, AdaBatch convergence closely matches the fixed 4096 batch size. Fixed batch sizes of 8192 and 16,384 do not converge to the test errors of fixed batch size 4096 and AdaBatch. These results indicate that AdaBatch attains the same test error as fixed, small batches, but with the advantage of eventually training with a large batch size of 16,384. Due to the progressively larger batch sizes, it is likely that AdaBatch will achieve higher performance and train faster.

Figure 7.8 illustrates the test errors of large batch size ImageNet training using learning rate warmup over the first 5 epochs [60]. For these experiments we use a baseline batch size of 256 when linearly scaling the learning rate. All other training parameters remain the same as in Figure 7.7. Figure 7.8a compares the test error curves of adaptive batch size 8192-32768 against fixed batch sizes $8192, 16384$, and 32768. Figure 7.8b performs the same experiment but with starting batch sizes of 16384. In both experiments, we observe that adaptive batch sizes have similar convergence behavior to the small, fixed batch size settings (8192 in Fig. 7.8a and 16384 in Fig. 7.8b). Furthermore, the adaptive batch size technique attains lower test errors than large, fixed batch sizes.

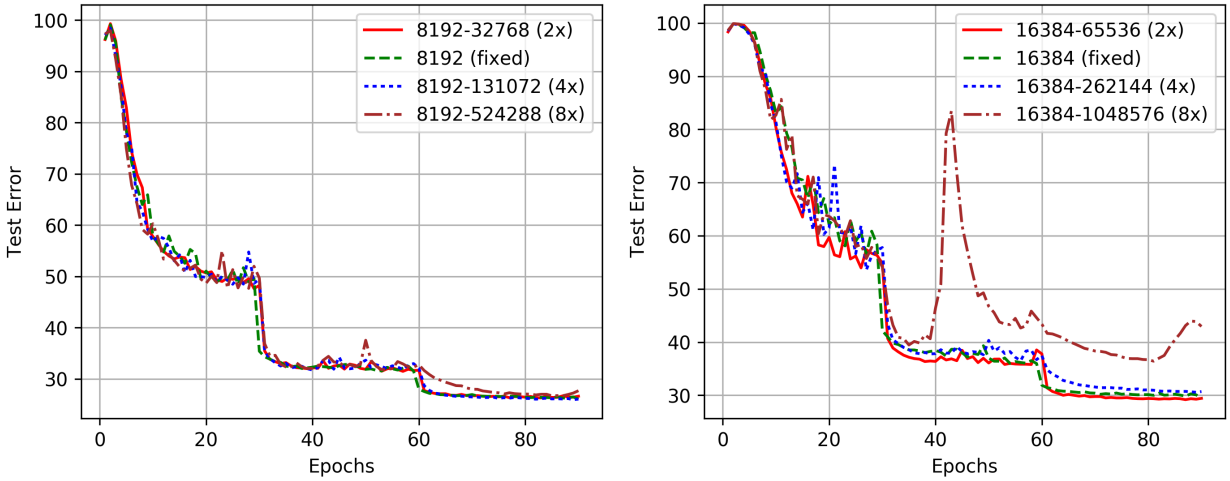
Thus far, the adaptive batch size experiments have doubled the batch size at fixed intervals. In Figure 7.9 we explore the convergence behavior of adaptive batch sizes with increase factors of $2\times$, $4\times$ and $8\times$. We use learning rate warmup with a baseline batch size of 256 for learning rate scaling and with all other training parameters the same as in previous ImageNet experiments. Figure 7.9a compares the test errors of fixed batch size 8192 and adaptive batch sizes starting at



(a) ResNet-50, starting batch size 8192.

(b) ResNet-50, starting batch size 16384.

Figure 7.8: Comparison of ImageNet test errors curves for adaptive versus fixed batch sizes with LR warmup.



(a) ResNet-50, starting batch size 8192.

(b) ResNet-50, starting batch size 16384.

Figure 7.9: Comparison of ImageNet test errors curves for adaptive batch sizes with LR warmup and batch size increases of 2x, 4x, and 8x.

8192 with batch size increase factors of $2\times$, $4\times$, and $8\times$. Our results indicate that the test error curves of all adaptive batch size settings closely match the fixed batch size curve. The convergence of adaptive batch size with $8\times$ increase slows after Epoch 60, however the final test error is similar to other curves. This experiment suggests that adaptive batch sizes enable ImageNet training with batch sizes of up to 524288 without significantly altering test error. Figure 7.9b shows test error

curves with batch sizes starting at 16384. Unlike Figure 7.9a, increasing the batch size by $8\times$ results in poor convergence. This is a result of increasing the batch size too much and too early in the training process. Therefore, it is important to tune the batch size increase factor proportional to the starting batch size. With starting batch size of 16384, we are able to increase the batch size by a factor of $4\times$ without significantly altering final test error and attain a final batch size of 262144.

7.4 Conclusions and Future Work

In this paper we have developed an adaptive scheme that dynamically varies the batch size during training. We have shown that using our scheme for AlexNet, ResNet and VGG neural network architectures with CIFAR-10, CIFAR-100 and ImageNet datasets we can maintain the better test accuracy of small batches, while obtaining higher performance often associated with large batches. In particular, our results demonstrate that adaptive batch sizes can attain speedups of up to $6.25\times$ on 4 NVIDIA P100 GPUs with less than 1% accuracy difference when compared to fixed batch size baselines. Also, we have briefly analysed the effects of choosing larger batch size with respect to learning rate, test accuracy and performance as well as work per epoch. Our ImageNet experiments illustrate that batch sizes of up to 524288 can be attained without altering test error performance. In the future, we would like to explore the effects of different schedules for adaptively resizing the batch size, including possibly shrinking it to improve convergence properties of the algorithm.

Chapter 8

Future Work

In this chapter we will present directions for future work. We have used block coordinate descent to solve least-squares, support vector machines, and kernel problems. Block coordinate descent is one among many methods one can use to solve optimization problems. Therefore, applying the communication-avoiding technique presented in this thesis to other methods like Stochastic Gradient Descent (SGD) [13, 71, 103], and quasi-Newton methods like the popular Broyden-Fletcher-Goldfarb-Shanno (BFGS) method (and its limited-memory variant) [17, 48, 49, 57, 108] is an interesting direction for future work. Exploring the performance tradeoffs of the various methods for each problem is also important and developing criteria for choosing the best method for a specific input matrix, machine model, and programming model remains future work.

In Chapter 6, we showed predicted performance results for our CA-methods solving kernel problems. Implementing the derived methods and exploring the actual performance tradeoff between CA and non-CA methods remains future work.

While we have shown in Chapter 7 that the adaptive batch size technique can yield large speedups when training neural networks on GPUs, developing a generalizable criterion to decide when to increase batch size would be fruitful. Furthermore, we showed that ImageNet can be training using ResNet-50 with a batch size of $524,288$. Note that we used PyTorch's gradient accumulation feature to simulate training with such a large batch size. Obtaining actual performance results and exploring the tradeoff space at such a large batch size remains future work.

For much of the thesis (Chapter 3 - 6), we have considered a few convex optimization problems. However, there are numerous convex and non-convex optimization problems that we have not considered. Extending our technique to other convex optimization problems is future work. For non-convex problems, like neural networks, this becomes more challenging due to the large number of neural network architectures which need to be considered and re-arranged to avoid communication. It remains to be seen whether our technique can be automated for neural networks such that a communication-avoiding version can be obtained without derivation by hand.

The choices of s in all chapters were tuned by hand to illustrate the potential speedups. However, it is impractical to hand-tune s given the large number possible machine architecture, programming model, and dataset combinations. Therefore, designing an auto-tuner with low overhead that finds reasonably good values of s is an important step in making our algorithms practical. We

made the implicit assumption in this thesis that computation and communication cannot be overlapped. However, if we assume that each processor has enough memory, then the rows or columns of the input matrix required for the next iteration can be extracted while the previous iteration's partial Gram matrices are sum-reduced.

This could yield additional speedups over what we have observed from our experiments. Implementation of all algorithms in other programming models, like Spark, would be impactful. Preliminary work has already shown speedups, but further work is required in this area. Furthermore, if Spark is used in a cloud environment, where latency is more dominant than in MPI in a supercomputing environment, then our algorithms should yield even larger speedups. Quantifying those speedups is important.

We have assumed that double-precision floating-point numbers are used in the computations. In most machine learning applications, much less precision is required. For single and reduced precision, computation cost and bandwidth cost decrease but latency remains the same. As a result, the CA-methods can obtain even larger speedups then observed in this thesis.

Finally, an error analysis of the CA-methods is necessary to understand how the forward and backward error depend on s, block size, condition number of A, and λ, the regularization parameter. Note that the algorithm analysis uses the α-β model. More refined models are available and further analysis in those models would yield more accurate algorithm running time predictions.

Bibliography

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. “LogGP: Incorporating long messages into the LogP model for parallel computation”. In: *Journal of parallel and distributed computing* 44.1 (1997), pp. 71–79.

[2] C. Allauzen, C. Cortes, and M. Mohri. “A Dual Coordinate Descent Algorithm for SVMs Combined with Rational Kernels”. In: *Int. J. Found. Comput. Sci.* 22 (2011), pp. 1761–1779.

[3] G. Ballard. “Avoiding Communication in Dense Linear Algebra”. PhD thesis. EECS Department, University of California, Berkeley, 2013.

[4] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. “Communication lower bounds and optimal algorithms for numerical linear algebra”. In: *Acta Numerica* 23 (2014), pp. 1–155.

[5] G. Ballard, A. Buluc, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo. “Communication Optimal Parallel Multiplication of Sparse Random Matrices”. In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA ’13. New York, NY, USA: ACM, 2013, pp. 222–231.

[6] L. Balles, J. Romero, and P. Hennig. “Coupling Adaptive Batch Sizes with Learning Rates”. In: *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence (UAI)*. 2017, pp. 410–419.

[7] A. Beck and M. Teboulle. “A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems”. In: *SIAM Journal on Imaging Sciences* 2.1 (2009), pp. 183–202.

[8] Y. Bengio and Y. LeCun. “Scaling Learning Algorithms Towards AI”. In: *Large Scale Kernel Machines*. MIT Press, 2007.

[9] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[10] Ã. Björck. *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics, 1996.

[11] L. Bottou. “Large-scale machine learning with stochastic gradient descent”. In: *Proceedings of Computation Statistics*. Springer, 2010, pp. 177–186.

[12] L. Bottou, F. E. Curtis, and J. Nocedal. “Optimization Methods for Large Scale Machine Learning”. In: *arXiv preprint arXiv:1606.04838* (2016).

[13] L. Bottou, F. E. Curtis, and J. Nocedal. *Optimization Methods for Large-Scale Machine Learning*. Tech. rep. 2016.

[14] *Box Plots*. `https://www.mathworks.com/help/stats/box-plots.html`.

[15] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[16] L. Breiman. "Better Subset Regression Using the Nonnegative Garrote". In: *Technometrics* 37.4 (1995), pp. 373–384.

[17] C. G. Broyden. "The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations". In: *IMA Journal of Applied Mathematics* 6.1 (1970), pp. 76–90.

[18] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. "Efficient algorithms for all-to-all communications in multiport message-passing systems". In: *IEEE Transactions on Parallel and Distributed Systems* 8.11 (1997), pp. 1143–1156.

[19] R. H. Byrd, G. M. Chin, J. Nocedal, and Y. Wu. "Sample size selection in optimization methods for machine learning". In: *Mathematical programming* 134.1 (2012), pp. 127–155.

[20] E. Carson. "Communication-Avoiding Krylov Subspace Methods in Theory and Practice". PhD thesis. EECS Department, University of California, Berkeley, 2015.

[21] E. Carson and J. Demmel. "A residual replacement strategy for improving the maximum attainable accuracy of s-step Krylov subspace methods". In: *SIAM Journal on Matrix Analysis and Applications* 35.1 (2014), pp. 22–43.

[22] E. Carson and J. Demmel. "Accuracy of the s-step Lanczos method for the symmetric eigenproblem in finite precision". In: *SIAM Journal on Matrix Analysis and Applications* 36.2 (2015), pp. 793–819.

[23] E. Carson, N. Knight, and J. Demmel. "An efficient deflation technique for the communication-avoiding conjugate gradient method". In: *Electronic Transactions on Numerical Analysis* 43 (2014), pp. 125–141.

[24] E. Carson, N. Knight, and J. Demmel. "Avoiding Communication in Nonsymmetric Lanczos-Based Krylov Subspace Methods". In: *SIAM Journal on Scientific Computing* 35.5 (2013), S42–S61.

[25] C. Chan, D. Unat, M. Lijewski, W. Zhang, J. Bell, and J. Shalf. "Software Design Space Exploration for Exascale Combustion Co-design". In: *Supercomputing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 196–212. ISBN: 978-3-642-38750-0.

[26] C.-C. Chang and C.-J. Lin. "LIBSVM: A library for support vector machines". In: *ACM Transactions on Intelligent Systems and Technology* 2 (3 2011), pp. 1–27.

[27] K.-W. Chang, C.-J. Hsieh, and C.-J. Lin. "Coordinate Descent Method for Large-scale L2-loss Linear Support Vector Machines". In: *J. Mach. Learn. Res.* 9 (June 2008), pp. 1369–1398.

[28] A. T. Chronopoulos. “A class of parallel iterative methods implemented on multiprocessors”. PhD thesis. University of Illinois Urbana-Champaign, Department of Computer Science, 1987.

[29] A. T. Chronopoulos and C. W. Gear. “On the efficient implementation of preconditioned s-step conjugate gradient methods on multiprocessors with memory hierarchy”. In: *Parallel Computing* 11.1 (1989), pp. 37 –53.

[30] A. T. Chronopoulos and C. W. Gear. “s-step iterative methods for symmetric linear systems”. In: *Journal of Computational and Applied Mathematics* 25.2 (1989), pp. 153 –168.

[31] A. T. Chronopoulos and C. D. Swanson. “Parallel iterative S-step methods for unsymmetric linear systems”. In: *Parallel Computing* 22.5 (1996), pp. 623–641.

[32] C. Cortes and V. Vapnik. “Support-vector networks”. In: *Machine Learning* 20.3 (1995), pp. 273–297.

[33] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, and T. Subramonian R.and Von Eicken. *LogP: Towards a realistic model of parallel computation*. Vol. 28. 7. ACM, 1993.

[34] H. Daneshmand, A. Lucchi, and T. Hofmann. “Starting Small - Learning with Adaptive Sample Sizes”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 2016, pp. 1463–1471.

[35] D. Das, S. Avancha, D. Mudigere, K. Vaidynathan, S. Sridharan, D. Kalamkar, B. Kaul, and P. Dubey. “Distributed deep learning using synchronous stochastic gradient descent”. In: *arXiv preprint arXiv:1602.06709* (2016).

[36] I. Daubechies, M. Defrise, and C. De Mol. “An iterative thresholding algorithm for linear inverse problems with a sparsity constraint”. In: *Communications on Pure and Applied Mathematics* 57.11 (2004), pp. 1413–1457.

[37] Timothy A. Davis, Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. “A survey of direct methods for sparse linear systems”. In: *Acta Numerica* 25 (2016), pp. 383–566.

[38] S. De, A. Yadav, D. Jacobs, and T. Goldstein. “Big Batch SGD: Automated Inference using Adaptive Batch Sizes”. In: *arXiv preprint arXiv:1610.05792* (Oct. 2016).

[39] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113.

[40] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. *Avoiding Communication in Computing Krylov Subspaces*. Tech. rep. UCB/EECS-2007-123. EECS Department, University of California, Berkeley, 2007.

[41] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. “Communication-avoiding parallel and sequential QR and LU factorizations”. In: *SIAM Journal of Scientific Computing* (2008).

[42] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". In: *Conference on Computer Vision and Pattern Recognition*. 2009.

[43] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[44] A. Devarakonda, M. Naumov, and M. Garland. "AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks". In: *arXiv preprint arXiv:1712.02029* (2017).

[45] A. Devarakonda, K. Fountoulakis, J. Demmel, and M. W. Mahoney. "Avoiding communication in primal and dual block coordinate descent methods". In: *arXiv preprint arXiv:1612.04003* (2016).

[46] A. Devarakonda, K. Fountoulakis, J. Demmel, and M. W. Mahoney. "Avoiding Synchronization in First-Order Methods for Sparse Convex Optimization". In: *arXiv preprint arXiv:1712.06047* (2017).

[47] O. Fercoq and P. Richtárik. "Accelerated, Parallel, and Proximal Coordinate Descent". In: *SIAM Journal on Optimization* 25.4 (2015), pp. 1997–2023.

[48] R. Fletcher. "A new approach to variable metric algorithms". In: *The Computer Journal* 13.3 (1970), pp. 317–322.

[49] R. Fletcher. *Practical methods of optimization*. John Wiley & Sons, 2013.

[50] A. Fog. *Instruction tables*. http://www.agner.org/optimize/instruction_tables.pdf. 2018.

[51] K. Fountoulakis and J. Gondzio. "Performance of First- and Second-Order Methods for L1-Regularized Least Squares Problems". In: *arXiv preprint arXiv:1503.03520* (Mar. 2015).

[52] K. Fountoulakis and R. Tappenden. "Robust Block Coordinate Descent". In: *arXiv preprint arXiv:1407.7573* (July 2014).

[53] M. P. Friedlander and M. Schmidt. "Hybrid Deterministic-Stochastic Methods for Data Fitting". In: *SIAM Journal on Scientific Computing* 34.3 (2012), A1380–A1405.

[54] J. Friedman, T. Hastie, and R. Tibshirani. "A note on the group lasso and a sparse group lasso". In: *arXiv preprint arXiv:1001.0736* (Jan. 2010).

[55] S. H. Fuller and L. I. Millett. "Computing performance: Game over or next level?" In: *Computer* 1 (2011), pp. 31–38.

[56] A. Gittens, A. Devarakonda, E. Racah, M. Ringenburg, L. Gerhardt, J. Kottalam, J. Liu, K. Maschhoff, S. Canon, J. Chhugani, P. Sharma, J. Yang, J. Demmel, J. Harrell, V. Krishnamurthy, M. W. Mahoney, and Prabhat. "Matrix factorizations at scale: A comparison of scientific data analytics in spark and C+MPI using three case studies". In: *2016 IEEE International Conference on Big Data*. 2016, pp. 204–213.

[57] D. Goldfarb. "A Family of Variable-Metric Methods Derived by Variational Means". In: *Mathematics of Computation* 24.109 (1970), pp. 23–26.

[58] G. H. Gonnet. “Expected length of the longest probe sequence in hash code searching”. In: *Journal of the ACM (JACM)* 28.2 (1981), pp. 289–304.

[59] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[60] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”. In: *arXiv preprint arXiv:1706.02677* (2017).

[61] S. L. Graham, M. Snir, and C. A. Patterson. *Getting up to speed : the future of supercomputing*. Washington, DC: National Academies Press, 2005. ISBN: 0-309-09502-6.

[62] W. D. Gropp, E. Lusk, N. Doss, and A. Skjellum. “A high-performance, portable implementation of the MPI message passing interface standard”. In: *Parallel computing* 22.6 (1996), pp. 789–828.

[63] R. Harikandeh, M. O. Ahmed, A. Virani, M. Schmidt, J. Konečný, and S. Sallinen. “Stop Wasting My Gradients: Practical SVRG”. In: *Advances in Neural Information Processing Systems 28*. Curran Associates, Inc., 2015, pp. 2251–2259.

[64] K. He, X. Zhang, S. Ren, and J. Sun. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[65] C. Heinze, B. McWilliams, and N. Meinshausen. “Dual-loco: Distributing statistical estimation using random projections”. In: *Proceedings of AISTATS*. 2016.

[66] G. E. Hinton, S. Osindero, and Y. W. Teh. “A Fast Learning Algorithm for Deep Belief Nets”. In: *Neural Computation* 18 (2006), pp. 1527–1554.

[67] M. Hoemmen. “Communication-avoiding Krylov subspace methods”. PhD thesis. University of California, Berkeley, 2010.

[68] 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*. ICML ’08. Helsinki, Finland: ACM, 2008, pp. 408–415. ISBN: 978-1-60558-205-4.

[69] S. Ioffe and C. Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. Proceedings of Machine Learning Research. Lille, France, 2015, pp. 448–456.

[70] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima”. In: *arXiv preprint arXiv:1609.04836* (2016).

[71] J. Kiefer and J. Wolfowitz. “Stochastic Estimation of the Maximum of a Regression Function”. In: *Ann. Math. Statist.* 23.3 (Sept. 1952), pp. 462–466.

[72] S. K. Kim and A. T. Chronopoulos. "An efficient nonsymmetric Lanczos method on parallel vector computers". In: *Journal of Computational and Applied Mathematics* 42.3 (1992), pp. 357 –374.

[73] A. Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[74] A. Krizhevsky, V. Nair, and G. Hinton. *CIFAR-10 (Canadian Institute for Advanced Research)*. 2009.

[75] A. Krizhevsky, V. Nair, and G. Hinton. *CIFAR-100 (Canadian Institute for Advanced Research)*. 2009.

[76] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, pp. 1097–1105.

[77] K. Lang. "NewsWeeder: Learning to Filter Netnews". In: *Proceedings of the 12th International Machine Learning Conference*. 1995.

[78] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage". In: *ACM Trans. Math. Softw.* 5.3 (Sept. 1979), pp. 308–323.

[79] J. D. Lee, Y. Sun, and M. A. Saunders. "Proximal Newton-Type Methods for Minimizing Composite Functions". In: *SIAM Journal on Optimization* 24.3 (2014), pp. 1420–1443.

[80] M. Lichman. *UCI Machine Learning Repository*. 2013.

[81] O. L. Mangasarian and D. R. Musicant. "Successive Overrelaxation for Support Vector Machines". In: *Trans. Neur. Netw.* 10.5 (Sept. 1999), pp. 1032–1037.

[82] J. Mareček, P. Richtárik, and M. Takáč. "Distributed block coordinate descent for minimizing partially separable functions". In: *Numerical Analysis and Optimization*. Springer, 2015, pp. 261–288.

[83] A. McCallum. *SRAA: Simulated/Real/Aviation/Auto UseNet data*. `https://people.cs.umass.edu/~mccallum/data.html`.

[84] M. D. Mitzenmacher. "The Power of Two Choices in Randomized Load Balancing". PhD thesis. EECS Department, University of California, Berkeley, 1996.

[85] M. Mohiyuddin. "Tuning Hardware and Software for Multiprocessors". PhD thesis. EECS Department, University of California, Berkeley, 2012.

[86] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. "Minimizing Communication in Sparse Matrix Solvers". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon: ACM, 2009, 36:1–36:12. ISBN: 978-1-60558-744-8.

[87] M. Naumov. "Feedforward and Recurrent Neural Networks Backward Propagation and Hessian in Matrix Form". In: *arXiv preprint arXiv:1709.06080* (2017).

[88] *NERSC Cori Configuration*. `http://www.nersc.gov/users/computational-systems/cori/configuration/`.

[89] *NERSC Edison Configuration*. http://www.nersc.gov/users/computational-systems/edison/configuration/.

[90] Y. Nesterov. "Efficiency of Coordinate Descent Methods on Huge-Scale Optimization Problems". In: *SIAM Journal on Optimization* 22 (2012), pp. 341–362.

[91] A. Nitanda. "Stochastic Proximal Gradient Descent with Acceleration Techniques". In: *Advances in Neural Information Processing Systems*. 2014, pp. 1574–1582.

[92] NVIDIA. *NVIDIA NVLink High-Speed Interconnect: Application Performance*. 2014. URL: http://www.nvidia.com/object/nvlink.html.

[93] NVIDIA. *NVIDIA Tesla P100 GPU Architecture*. 2016. URL: http://www.nvidia.com/object/pascal-architecture-whitepaper.html.

[94] NVIDIA. *NVIDIA Tesla V100 GPU Architecture*. 2017. URL: http://www.nvidia.com/object/volta-architecture-whitepaper.html.

[95] X. Pan. "Parallel Machine Learning Using Concurrency Control". PhD thesis. EECS Department, University of California, Berkeley, 2017.

[96] N. Parikh and S. Boyd. "Proximal Algorithms". In: *Foundations and Trends in Optimization* 1.3 (2014), pp. 127–239.

[97] J. Platt. *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines*. Tech. rep. 1998.

[98] *PyTorch*. pytorch.org. Accessed: 2017-10-12.

[99] M. Raab and A. Steger. ""Balls into Bins" − A Simple and Tight Analysis". In: *Randomization and Approximation Techniques in Computer Science*. Springer, 1998, pp. 159–170.

[100] B. Recht, C. Ré, S. Wright, and F. Niu. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *Advances in Neural Information Processing Systems*. 2011, pp. 693–701.

[101] P. Richtárik and M. Takáč. "Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function". In: *Mathematical Programming* 144.1 (2014), pp. 1–38.

[102] Peter Richtárik and Martin Takáč. "Distributed Coordinate Descent Method for Learning with Big Data". In: *J. Mach. Learn. Res.* 17.1 (Jan. 2016), pp. 2657–2681.

[103] H. Robbins and S. Monro. "A Stochastic Approximation Method". In: *Ann. Math. Statist.* 22.3 (Sept. 1951), pp. 400–407.

[104] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (Oct. 1986), 533 EP –.

[105] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[106] S. Shalev-Shwartz and T. Zhang. "Stochastic dual coordinate ascent methods for regularized loss". In: *The Journal of Machine Learning Research* 14.1 (2013), pp. 567–599.

[107] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter. "Pegasos: primal estimated sub-gradient solver for SVM". In: *Mathematical Programming* 127.1 (2011), pp. 3–30.

[108] D. F. Shanno. "Conditioning of Quasi-Newton Methods for Function Minimization". In: *Mathematics of Computation* 24.111 (1970), pp. 647–656.

[109] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[110] S. L. Smith, P.-J. Kindermans, and Q. V. Le. "Don't Decay the Learning Rate, Increase the Batch Size". In: *arXiv preprint arXiv:1711.00489* (2017).

[111] V. Smith. "System-Aware Optimization for Machine Learning at Scale". PhD thesis. EECS Department, University of California, Berkeley, 2017.

[112] V. Smith, S. Forte, M. I. Jordan, and M. Jaggi. "L1-Regularized Distributed Optimization: A Communication-Efficient Primal-Dual Framework". In: *arXiv preprint arXiv:1512.04011* (2015).

[113] E. Solomonik. "Provably efficient algorithms for numerical tensor algebra". PhD thesis. EECS Department, University of California, Berkeley, 2014.

[114] J. Stamper, A. Niculescu-Mizil, S. Ritter, G.J. Gordon, and K.R. Koedinger. "Algebra 2008-2009 from Challenge data set". In: *KDD Cup 2010 Educational Data Mining Challenge*. 2010.

[115] M. Takáč, P. Richtárik, and N. Srebro. "Distributed Mini-Batch SDCA". In: *arXiv preprint arXiv:1507.08322* (2015).

[116] R. Thakur and W. D. Gropp. "Improving the performance of collective operations in MPICH". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Verlag, 2003, pp. 257–267.

[117] R. Thakur and W. D. Gropp. "Improving the Performance of MPI Collective Communication on Switched Networks". In: *Technical report ANL/MCS-P1007-1102, Mathematics and Computer Science Division, Argonne National Laboratory* (2002).

[118] R. Tibshirani. "Regression Shrinkage and Selection via the Lasso". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 58.1 (1996), pp. 267–288.

[119] J. Van Rosendale. "Minimizing inner product data dependencies in conjugate gradient iteration". In: IEEE Computer Society Press, Silver Spring, MD, 1983.

[120] H. F. Walker. "Implementation of the GMRES Method Using Householder Transformations". In: *SIAM Journal on Scientific and Statistical Computing* 9.1 (1988), pp. 152–163.

[121] S. Williams, M. Lijewski, A. Almgren, B. Van Straalen, E. Carson, N. Knight, and J. Demmel. "s-step Krylov subspace methods as bottom solvers for geometric multigrid". In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE. 2014, pp. 1149–1158.

[122] S. J. Wright. “Coordinate Descent Algorithms”. In: *Math. Program.* 151.1 (June 2015), pp. 3–34.

[123] Y. You, I. Gitman, and B. Ginsburg. “Scaling SGD Batch Size to 32K for ImageNet Training”. In: *arXiv preprint arXiv:1708.03888* (2017).

[124] 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*. 2015, pp. 847–859.

[125] H.-F. Yu, H.-Y. Lo, H.-P. Hsieh, J.-K. Lou, T. G. Mckenzie, J.-W. Chou, P.-H. Chung, C.-H. Ho, C.-F. Chang, J.-Y. Weng, E.-S. Yan, C.-W. Chang, T.-T. Kuo, P. T. Chang, C. Po, C.-Y. Wang, Y.-H Huang, Y.-X. Ruan, Y.-S. Lin, S.-D. Lin, H.-T. Lin, and C.-J. Lin. “Feature engineering and classifier ensemble for KDD Cup 2010”. In: *JMLR Workshop and Conference Proceedings*. 2011.

[126] M. Yuan and Y. Lin. “Model selection and estimation in regression with grouped variables”. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 68.1 (2006), pp. 49–67.

[127] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. “Spark: Cluster Computing with Working Sets”. In: HotCloud’10. Boston, MA: USENIX Association, 2010.

[128] Y. Zhang, J. C. Duchi, and M. J. Wainwright. “Communication-efficient Algorithms for Statistical Optimization”. In: *J. Mach. Learn. Res.* 14.1 (Jan. 2013), pp. 3321–3363.

[129] Y. Zhang and X. Lin. “DiSCO: Distributed Optimization for Self-Concordant Empirical Loss”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. 2015, pp. 362–370.

[130] Y. Zhang, M. J. Wainwright, and J. C. Duchi. “Communication-efficient algorithms for statistical optimization”. In: *Advances in Neural Information Processing Systems*. 2012, pp. 1502–1510.

[131] Z. A. Zhu, W. Chen, G. Wang, C. Zhu, and Z. Chen. “P-packSVM: Parallel Primal grAdient desCent Kernel SVM”. In: *IEEE International Conference on Data Mining*. 2009, pp. 677–686. ISBN: 978-0-7695-3895-2.

[132] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. “Parallelized Stochastic Gradient Descent”. In: *Advances in Neural Information Processing Systems*. 2010, pp. 2595–2603.