Framework Design for Improving Computational Efficiency and Programming Productivity for Distributed Machine Learning

Jin Kyu Kim

CMU-CS-18-127
December 2018

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Garth A. Gibson (Co-Chair)
Eric P. Xing (Co-Chair)
Phillip Gibbons
Joseph E. Gonzalez (University of California Berkeley)

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

This research was sponsored by the National Science Foundation under grant number CCF-1629559, the Defense Advanced Research Projects Agency under grant number FA8702-15-D-0002, Samsung, and Intel ISTC-CC. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Distributed Systems, Large-Scale Machine Learning, Programming Framework, Computer Science

Dedicated to my family and friends.

Abstract

Machine learning (ML) methods are used to analyze data in a wide range of areas, such as finance, e-commerce, medicine, science, and engineering, and the size of machine learning problems has grown very rapidly in terms of data size and model size in the era of big data. This trend drives industry and academic communities toward distributed machine learning that scales out ML training in a distributed system for completion in a reasonable amount of time. There are two challenges in implementing distributed machine learning: computational efficiency and programming productivity. The traditional data-parallel approach often leads to suboptimal training performance in distributed ML due to data dependencies among model parameter updates and nonuniform convergence rates of model parameters. From the perspective of an ML programmer, distributed ML programming requires substantial development overhead even with high-level frameworks because they require an ML programmer to switch to a different mental model for programming from a familiar sequential programming model.

The goal of my thesis is to improve the computational efficiency and programming productivity of distributed machine learning. In an efficiency study, I explore model update scheduling schemes that consider data dependencies and nonuniform convergence speeds of model parameters to maximize convergence per iteration and present a runtime system STRADS that efficiently execute model update scheduled ML applications in a distributed system. In a productivity study, I present familiar sequential-like programming API that simplifies conversion of a sequential ML program into a distributed program without requiring an ML programmer to switch to a different mental model for programming and implement a new runtime system STRADS-Automatic Parallelization(AP) that efficiently executes ML applications written in our API in a distributed system.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisors, Garth Gibson and Eric Xing, for their guidance and support throughout my entire PhD study at CMU. I want to especially thank them for showing unwaivering trust in me and letting me pursue diverse interests in my research. Garth Gibson and Eric Xing have been model researchers and leaders as well as great academic mentors. They have motivated me to excel in every aspect of my research journey. I also want to thank my thesis committee members Phillip Gibbons and Joseph Gonzalez for their insightful feedback and comments. I would also like to express deep gratitude to BigLearning meeting members: Garth Gibson, Eric Xing, Phillip Gibbons, Greg Ganger, Abutalib Aghayev, James Cipar, Henggang Cui, Wei Dai, Aaron Harlap, Qirong Ho,Kevin Hsieh, Seunghak Lee, Aurick Qiao, Jinliang Wei, Hao Zhang, and Xun Zheng. Since 2013 we have met every week and discussed various topics regarding system for machine research. These meetings were extremely valuable for conducting my PhD research, and many ideas in my thesis research were inspired by these people. I also want to thank the members of Parallel Data Lab and the companies of Parallel Data Lab Consortium (including Alibaba Group, Amazon, Datrium, Dell/EMC, Facebook, Google, Hewlett-Packard, Hitachi., IBM Research, Intel, Micron, Microsoft Research, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma, Veritas, Western Digital) for their interest, insights, feedback, and support. I would also like to thank my friends and colleagues: Logan Brooks, Avinava Dubey, Bin Fan, Partik Fegade, Kiryong Ha, Zhiting Hu, U Kang, Kunhee Kim, Soonho Kong, Alankar Kotwal, Hyeontaek Lim, Shane Moon, Willie Neiswanger, Swapnil Patil, Wolfgang Richter, Kijeong Shin, Julian Shun, Jiri Simsa, Yuanhao Wei, and Pengtao Xie. It was a great pleasure to discuss various research topics and random subjects with these great friends. Finally, I would like to thank my parents as well as my wife and daughter, Hanna and Rachel, for their endless love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation aims to improve **the efficiency of distributed machine learning computation** and **the productivity of programming distributed machine learning programs**. More specifically, efficiency denotes *the ratio of the training speed gain to the number of processors* where the training speed gain refers to the ratio of the training time of a sequential machine learning program to the training time [1] of a distributed machine learning program. The (distributed) training time can be viewed as the product of two factors: the count of iterations that a (distributed) program needs until it achieves a desired quality of solution; and the average time per iteration of the (distributed) program. Productivity denotes *the development time* for a programmer to convert a sequential program[2] into a distributed program that runs in a cluster.

In an efficiency study, first of all, we present that the loss of statistical accuracy in distributed machine learning using a traditional data-parallel approach has large negative impacts on the computational efficiency in many applications. In other words, the loss of statistical accuracy in the distributed program increases the number of iteration needed until the distributed program achieves the same convergence or prediction accuracy of the corresponding sequential program, which negatively affects performance gains and computational efficiency of the distributed program. To address this problem, we present machine learning task scheduling, SchMP[3] and, to show the feasibility of SchMP in a cluster, implement a new runtime system, **STRADS**[4] that efficiently executes SchMP ma-

[1] In this dissertation, training time is used interchangeably with time to solution(latency) in [24].

[2] We assume that distributed machine learning code starts from sequential code — which is considered as a normal practice in traditional machine learning development.

[3] SchMP stands for Scheduled Model-Parallel

[4] STRADS stands for STRucture-Aware Dynamic/Static scheduler

chine learning applications in a cluster. SchMP avoids data conflicts in parallel execution and allows a distributed machine learning program to achieve high statistical accuracy close to that of a sequential program. However, SchMP might incur system overheads, such as scheduling latency, frequent synchronization or increased communication, that might lead to suboptimal iteration throughput in a cluster. To execute SchMP machine learning applications efficiently in a cluster, the STRADS runtime system implements a few critical system optimizations that can be applied to a wide range of machine learning applications: (1) pipelined scheduling that makes trade-offs between statistical accuracy and system throughput; (2) static scheduling that removes runtime scheduling latency for applications whose scheduling plan is known prior to runtime; and (3) parameter prioritization that gives more execution chances to the less converged model parameters, which greatly reduces the waste of CPU cycles that would be consumed to update fully (or nearly) converged model parameters without making any contribution to convergence otherwise. The results of performance benchmarking a few popular applications, such as latent dirichlet allocation (LDA)[8], stochastic gradient descent matrix factorization (SGDMF)[44], sparse regression (lasso)[82], and logistic regression[7], show that the application of SchMP achieves ideal progress per iteration (close to that of a sequential program), and the STRADS runtime system achieves comparable iteration throughput to data-parallel alternatives. As a result, the combination of SchMP and the STRADS runtime system improves the training speed by up to an order of magnitude compared to alternative distributed data-parallel approach.

While STRADS improves efficiency significantly, it does not improve productivity satisfactorily because it leaves the burden of writing machine learning task scheduling code and managing the mechanics of distributed programming (i.e. data partitioning, fault-tolerance) in the hands of machine learning programmers. The mismatch of machine learning skill and distributed programming skill strongly motivates us to conduct a development productivity study.

In the development productivity study that follows, we aim to simplify distributed machine learning programming to allow a machine learning-savvy researcher without extensive distributed programming knowledge to convert a sequential machine learning program into a distributed program almost mechanically, without significant programming burden. Toward this goal, first, we investigate the cost of using a high-level framework for distributed machine learning. Our investigation reveals that some high-level frameworks require a different mental model for programming, and switching to a different mental model for programming from a simple sequential programming model lowers development productivity significantly. To address this challenge, we present a new framework STRADS-AP with a familiar sequential API that does not require a machine learning programmer to switch to different mental model for programming for distributed machine learning programming. The new API consists of two components: (1) a set of distributed data structures (DDSs) that provides a similar interface of C++ STL containers while hiding details of partitioning data in a cluster, which allows a programmer to reuse data structures of a sequential program;and (2) two parallel loop operators, SyncFor and ASyncFor that automatically parallelize machine learning computations while hiding details of concurrency control and parameter aggregation, which allows a programmer to reuse the parameter update code, which is a core part of ML program, of a sequential program. To evaluate the potential benefit of our new API, we design and implement a new distributed runtime system, **STRADS-Automatic Parallelization** (**STRADS-AP**) that is responsible for data partitioning of DDSs, parallelization of computations, generation of machine learning task schedule plan (for SchMP applications), parameter aggregation (for data-parallel applications), fault-tolerance, and DDS prefetching/caching for performance improvement. A productivity evaluation of STRADS-AP demonstrates that STRADS-AP allows a programmer to reuse most of data structures and core computation routines of a sequential program when developing a distributed program and reduces development time significantly. A performance evaluation shows that STRADS-AP applications, including stochastic gradient descent matrix factorization (SGDMF), multi-class logistic regression (MLR), word embedding (Word2vec), and knowledge graph embedding (TransE), achieve comparable performance to hand-tuned distributed implementations programmed manually or on an machine learning specialized high-level framework such as TensorFlow.

1.1 Scope of work

This section defines the scope of work in this dissertation – clarifying the specific tasks in modern data analytics pipelines, the specific stages of machine learning application development processes, and the specific groups of people our work targets.

1.1.1 Training in machine learning

In this dissertation, we review a canonical pipeline of modern data analytics with a focus on the model training stage that this dissertation targets.

Over the last two decades, Internet and storage media technologies have improved extremely fast, leading to massive collections of data. Data analytic techniques in the fields of machine learning have also improved at a breaknek speed. These improvements

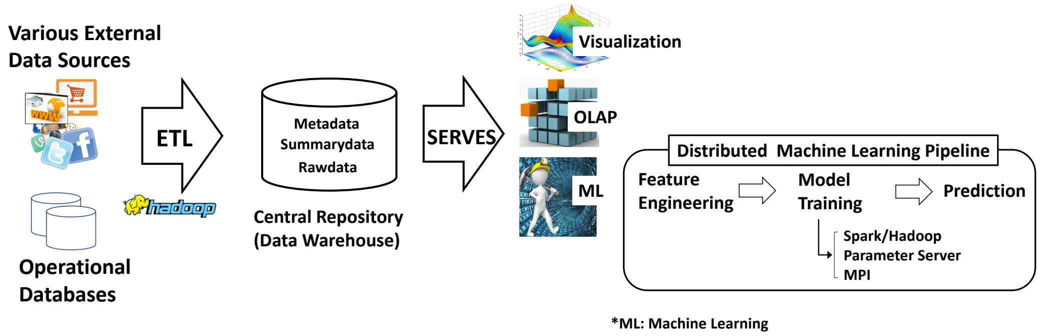


Figure 1.1: A contemporary data processing flow: Machine learning programs have emerged as a primary method for decision support.

have enabled quantum leaps in the quality of a wide range of useful technologies, from speech recognition to autonomous driving. Because of these advances, machine learning has emerged as a primary tool to analyze and explore big datasets.

A common data processing flow in a data warehouse begins with an ETL (Extract, Transform, Load) stage, which extracts data from various sources, then transform the data into the proper format or structure for query and data analysis. Then, it loads the transformed data into a central repository (a data warehouse). The data in a data warehouse is used by traditional data warehouse applications, such as OLAP(Online Analytical Processing) and visualization tools, and more recently machine learning applications. Traditionally, database tools have been used to perform the ETL process, while relational databases have been used to maintain the data in a data warehouse. More recently, distributed data processing frameworks, such as Hadoop and Spark, were developed to perform ETL on big datasets faster. NoSQL databases, such as BigTable [14], HBase and DynamoDB, were developed to maintain data in a data warehouse.

The machine learning pipeline consists of three stages: feature engineering, model training, and prediction. Feature engineering improves training data by extracting attributes of raw input data relevant to machine learning algorithms. Model training is an iterative process to search for the parameter values of a specific model class that best represent the training data. Training usually starts with randomly initialized parameter values and stops when a stopping condition is satisfied (i.e. when sufficiently good parameter values are obtained). Finally, in the prediction stage, the obtained parameter values are used to identify the labels of newly-arriving data instances that are not present in the training data or to explain hidden distribution(s) of the input data.

Of these three stages in the machine learning pipeline, the first part of this dissertation focuses on **model training particularly in a distributed environment**. Due to the large volume of training data, model parameters and associated computation, the training stage usually serves as a bottleneck for the entire pipeline. The exponential growth of training datasets and model parameters prevents the timely completion of training on a single machine that has limited computation power. Such a training constraint drives the machine learning research community and industry toward distributed parallel training. However, distributed training has an efficiency issue — distributed computing resources are typically less well utilized for training a machine learning model. Naive ways of parallelizing the training tasks tend to increase iteration count until convergence (= make less progress per iteration), which leads to poor training performance in a distributed system. Our efficiency study is dedicated to improving the efficiency of distributed parallel training.

1.1.2 Development of distributed machine learning programs

In this section, we review the machine learning program development process and the people who are involved in the process with a focus on the distributed machine learning development stage that this dissertation mainly targets.

Before discussing the development process, we present a vertical and logical dissection of a typical machine learning program to help a general audience understand the machine learning program development process. At the highest level, in Figure 1.2, a problem of interest is defined as fitting data to a statistical model (i.e. classification problem fitted to a linear model, perhaps by a logistic regression or support vector machine). This typically reduces to an optimization problem with a set of unknown model parameters (i.e. coefficients in a linear model) and objective (or likelihood) function (i.e. logistic loss function for logistic regression or hinge loss function for support vector machine). We can solve these optimization problems using various algorithms that iteratively search for the best values for the unknown model parameters to maximize or minimize the value of an objective function. Note that there are almost always multiple algorithms available for solving a machine learning optimization problem[5]. An algorithm for solving a machine learning optimization problem specifies a set of steps in a search for better parameter values based on current parameter values and repeats the search steps until the algorithm obtains a desirable quality of parameter values. The conversion of search steps to a computer

[5] For solving classification problem in the form of a logistic regression function, the machine learning community has developed many algorithms, such as BBR (cyclic coordinate descent)[31], CDN (coordinate descent with one-dimensional newton step)[37], SCD (stochastic coordinate descent)[76], CGD-GD (coordinate gradient descent)[84], TRON (Trust Region Newton Method)[50], IPM (Interior Point Method)[43], BMRM (Bundle Method for regularized Risk Minimization)[79], OWL-QN (Orthant-Wise Limited-memory Quasi-Newton)[3], and Lassplore[52], to name but a few.

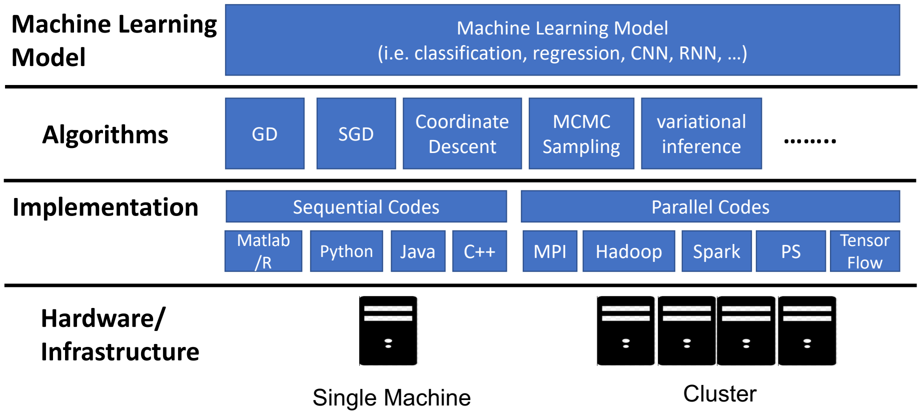


Figure 1.2: Block diagram of machine learning programs.

program is distinct task in model training when expressed as a single threaded sequential program. This conversion process is usually a straightforward task thanks to the various well-established programming environments available for statistical programming, such as MATLAB, R, Python, Java, C/C++, Julia, and Scala.

While there has been recent interest in parallelizing ML algorithms, most of the published model training algorithms are described in the form of a sequential algorithm, and most experiments implement and test sequential programs. This tendency to use sequential codes is not surprising. A sequential code is easy to reason about, requires less cognitive effort, and machine learning researchers are usually more concerned with the correctness and statistical convergence guarantee of an algorithm than its runtime performance. When the input data and its training computation are too big for a sequential program to complete a model training in a timely manner, it is then important to reimplement the sequential program as a distributed parallel program[6] to utilize the computation resources of a cluster. Then, the distributed program is deployed in a cluster and becomes available for data engineers and domain experts who solves domain-specific problems. Figure 1.3 summarizes the development process of machine learning applications from modeling to deployment in a cluster.

Machine learning is a relatively new and collaborative field that requires people with different knowledge and skill sets to be successful. We categorize people involved in the machine learning program development process into four groups:

[6] Distributed machine learning often involves both distributed programming across machines and parallel(and concurrent) programming within a shared-memory machine.

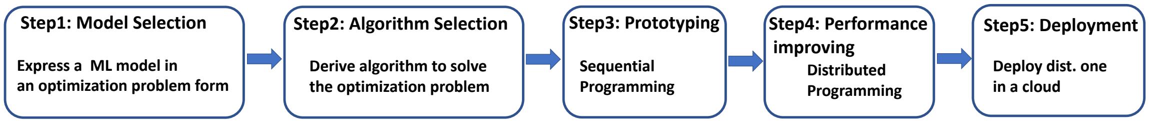

Step1: Model Selection
Express a ML model in an optimization problem form
Step2: Algorithm Selection
Derive algorithm to solve the optimization problem
Step3: Prototyping
Sequential Programming
Step4: Performance improving
Distributed Programming
Step5: Deployment
Deploy dist. one in a cloud

Figure 1.3: Development process of machine learning programs. The tasks in step 1 and 2 are typically conducted by machine learning researchers. Step 1 and 2 require high-level mathematical and statistical skills as well as domain knowledge. For quick development and verification, in step 3, they typically implement an algorithm of step 2 in sequential program and conduct verification with decent sized input data on a single machine. In step 4, the sequential program is reimplemented to be a distributed program for handling large data problems. Step 4 usually involves system engineers with distributed programming knowledge. Finally, in step 5, machine learning solutions are used by domain experts in various application domains. This thesis aims to improve the programming productivity of distributed machine learning programming in step 4 and improve training efficiency of distributed machine learning programs.

- **Machine learning researchers** are machine learning-savvy people who have expertise in statistics and mathematics that is essential for theoretical development tasks, such as inventing a new model, deriving a new algorithm, modifying an existing model to satisfy problem specific needs, and modifying an existing algorithm to improve prediction accuracy or satisfy application specific needs.

- **Domain experts** are people who have knowledge and skills in a particular field (i.e natural language processing, vision, health care, finance, warehouse management, supply chain management, manufacturing management, and so forth). These people are users of machine learning solutions and are often consulted in the modeling step. Traditionally, many application communities have developed their own methodologies over a long time, but recently machine learning technologies have been widely accepted and replaced traditional domain-specific methodologies across many application domains.

- **System engineers for machine learning** are people who have expertise in distributed parallel programming and knowledge about distributed machine learning. The distributed machine learning development often requires more skills than traditional distributed program development does. Unlike traditional computer programs, machine learning programs have unique properties, such as error-tolerance and uneven convergence, and a careful design that can exploit these properties is often critical to deliver efficient distributed programs.

- **Data engineers** are people who collect data, integrate data from multiple sources, clean up data, and make data available for machine learning applications.

As problem sizes grow rapidly, and application needs change quickly, fast development and deployment of distributed machine learning applications becomes important. However, parallel/concurrent programming is hard to reason about and introduces a large class of bugs — race conditions, atomicity violations, and deadlocks — that do not exist in a sequential code. And efficient distributed programming introduces even more complexities such as handling data partitioning, load-balancing, fault-tolerance, and network communication. Furthermore, ML-savvy people are interested more in sequential code than in distributed code. Therefore, the development of distributed machine learning program usually requires collaboration of machine learning researchers and system engineers. This collaboration often does not proceed smoothly because communication among people with different backgrounds is not easy and often takes longer than expected, which slows down distributed machine learning development.

Our development productivity study is dedicated to proposing a new high-level framework that allows machine learning researchers to convert a sequential machine learning program almost mechanically into a distributed program without much help from system engineers and reduces time to delivery of distributed program — better development productivity for distributed machine learning.

1.2 Thesis Statement

My thesis statement is:

Using the ability to make trade-offs between statistical accuracy and system throughput in theoretical machine learning scheduling and its practice in a distributed environment allows the training speed of model parallel applications to be improved by an order of magnitude; moreover, a familiar sequential-like programming API with a new runtime can simplify distributed programming for a wide range of machine learning applications while delivering performance comparable to hand-tuned distributed programs.

To support this thesis statement, we conducted five studies in this dissertation. Regarding efficiency, we investigated challenges in parallel ML with two popular ML applications and proposed the SchMP (Schedule Model Parallel) approach for making faster convergence per update compared to data-parallel. To show the validity of SchMP in a distributed system, we implemented a runtime system STRADS to execute SchMP applications efficiently in a distributed system. The STRADS runtime consists of two different execution engines:(1) a STRADS-Static engine that executes a static scheduling plan; and (2) a STRADS-Dynamic engine that schedules model parameters dynamically. Our evaluation with four popular applications shows (1) that our SchMP applications requires many fewer iterations than data-parallel for achieving the same quality of outputs and (2) system optimizations in the STRADS runtime system achieve high iteration throughput comparable to data-parallel implementations.

Regarding development productivity, we investigate the cost of using high-level frameworks for distributed machine learning and present STRADS-AP programming API that consists of distributed data structures (DDS) and loop operators that allow a machine learning programmer to convert a sequential program into a distributed program by adding few mechanical changes To show the effectiveness of our new API, we implement a new runtime system, STRADS-AP, that executes STRADS-AP applications efficiently in a distributed system. Our productivity evaluation results show that the STRADS-AP API allows a machine learning programmer to convert a sequential machine learning program into a distributed machine learning program almost mechanically, and our performance evaluation results demonstrate that the runtime system, with a few critical system optimization techniques, achieves performance comparable to hand-tuned distributed programs and machine learning specialized framework programs.

The rest of the dissertation is organized as follows.

- Chapter 2 reviews how distributed machine learning has been evolving for last two decades and presents our contributions. The advances of distributed machine learning have been led by progress from two different fields: (1) parallel machine learning that has proposed various ways for gradient/parameter aggregation schemes friendly to distributed execution while also providing a theoretical proof of safety for the new schemes; (2) systems for machine learning that use new programming models that hide the details of distributed programming from a machine learning programmer while also implementing system optimizations that improve training performance by exploiting unique properties of machine learning computations. We present a story intertwining these two fields and summarize our contributions.

- In Chapter 3, we investigate efficiency challenges in parallel machine learning with two popular ML applications: Lasso[81] and LDA topic modeling[8]. To address the efficiency challenges, we define and explore model parameter update scheduling approach (SchMP) and present an overview of STRADS runtime system that executes SchMP machine learning applications efficiently in a cluster. Our investigation reveals that common data-parallel implementations of these applications suffer from serious performance issues, such as low performance gain and poor scalability, because they do not address two challenges: model-dependency and uneven convergence. To address these challenges, we propose **scheduled model-parallel (SchMP)**, which schedules model parameter updates in a way that bounds the degree of dependency among concurrent model parameter updates to a threshold and prioritizes model parameter updates according to contribution to convergence. In this section, we explores various scheduling strategies and present two practical scheduling schemes, according to the dependency structures of targeted machine learning algorithms, by combining a subset of these strategies. To deploy SchMP machine learning applications in a cluster, we implement the STRADS runtime system that currently supports aforementioned two practical scheduling schemes. This chapter covers core parts of the STRADS runtime shared by these two schemes. The scheduling schemes and their implementations (Staticengine and Dynamicengine) are presented in the next two chapters.

- In Chapter 4, we present **Static-Scheduling** for machine learning algorithms, in which a serializable parameter update schedule[7], can be found prior to run time. Here, we address two questions: (1) What conditions of ML programs should be satisfied in an update schedule? and (2) What system optimizations should be supported to efficiently execute static-scheduled ML applications? On the machine learning side, we identified conditions for static-scheduling and presented a generic static-scheduling algorithm. On the system side, we presented a high-throughput STRADS-Static engine that implements two system optimizations: a ring overlay network and a strategy to address the load balancing problems.

- In Chapter 5, we present **Dynamic-Scheduling** for machine learning algorithms that have an error-tolerant dependency structure[8] and an uneven convergence rate[9]. This work is divided into two parts. On the machine learning side, we present a two-phase scheduling algorithm that improves statistical progress by respecting the dependency structure among parameter updates and considers the uneven convergence rate of model parameters. To maximize training speed, the dynamic scheduler inten-

[7] Serializable parameter update schedule ensures serializability for parallel execution and achieves statistical progress per iteration comparable to that of sequential execution, which we consider ideal.

[8] An error-tolerant dependency structure is a unique property of ML computation; an ML algorithm can absorb a certain degree of numerically bounded errors from concurrently executing dependent operations (i.e. operations having read-write or write-write dependencies on the same memory) and finally achieve convergence. However, concurrently executing dependent operations tends to increase the amount of computation required to reach convergence.

[9] An uneven convergence rate is a property of ML computation; different model parameters may take different number of updates to converge.

tionally ignores minor dependencies among parameter updates, which reduces statistical progress per update, but increases update throughput. The logic behind this strategy is that the performance gains from increased update throughput are larger than the loss from reduced progress per update. On the system side, we present a STRADS-Dynamic engine that implements a distributed scheduler and pipelines update operations to improve update throughput. The pipelining allows the $(t+1)$-th iteration[10] to start before the current iteration (t) is completed so that network latency of the $(t+1)$-th iteration can be overlapped with computation of the (t)-th iteration. The pipelining technique improves update throughput and is expected to reduce training time as a result. However, there may be pairs of very tightly coupled, high-priority parameters[11] across (t) and $(t{+}1)$ iterations, and pipelining these iterations might lower statistical progress per update so much that the performance gains from increased update throughput are less than the performance loss from reduced statistical progress per update. STRADS-Dynamic engine addresses this problem by changing the order of parameter updates to ensure that the update results of high-priority parameters are always available to the other high-priority parameters in the following iterations in order that the negative effects of pipelining are minimized.

- In Chapter 6, we present development productivity case study that investigates the development cost of using existing high-level frameworks for developing distributed machine learning. Our case study implements SGDMF on Spark[94] — one of most popular high-level frameworks for data analytics — and reveals that a high-level framework requires a machine learning programmer to switch to a different programming model from the sequential programming model, and that this switch to different programming model slows down the process of converting a sequential program into a distributed program. Under the different programming model that the framework of choice imposes, a machine learning programmer is required to redesign data structures and computation routine of a sequential machine learning program to fit into the programming model of the framework, and this redesign process costs a machine learning programmer significant efforts and time. Furthermore, the peculiarities of high-level frameworks often lead to suboptimal performance.

- In Chapter 7, we present a new framework, STRADS-AP, that consists of a familiar, sequential-like API and a new runtime system. STRADS-AP API consists of DDSs

[10] An iteration is a group of independent update operations.

[11] A high-priority parameter is a parameter that will have substantial change in its value and make significant contributions to convergence if it is updated. Pipelining two tightly coupled, high-priority parameters P_i and P_j over (t) and $(t+1)$ iterations may introduce substantial errors to updating P_j because the new value of P_i is not visible to P_j, and these errors lower statistical progress.

and two parallel operators. A DDS is a fine-grained mutable, distributed, in-memory container that enables reuse of data structures from a sequential program. Currently, STRADS-AP supports dvector, dmap, and dmultimap. STRADS-AP parallel loop operators (Sync/ASync_For) automatically parallelize machine learning computation in synchronous mode (data-parallel) or asynchronous mode (serializable asynchronous) while hiding details of aggregating shared parameters in Sync_For and details of managing concurrency control in Async_For.

We implemented STRADS-AP runtime system in about 15,000 lines of C++. To achieve automatic parallelization of loop operators, we design reconnaissance execution (RE) that executes a single read-only iteration to record read/write accesses to data and parameters stored in DDSs and thereby detect data dependencies among parameter update operations. STRADS-AP runtime uses this R/W access information to generate a parallel execution plan and improve performance of prefetching/caching DDS data elements from remote nodes. The full list of technically interesting parts in STRADS-AP runtime system design includes reconnaissance execution that detect data dependency, DDS library that implements prefetching, separate execution engines for Synch_For and Asynch_For operators, and STRADS-AP preprocessor that address the lack of reflection capability of C++. For performance benchmark, we implement four applications: recommendatation system (SGDMF)[44], word embedding (Word2vec)[61, 34], Multiclass Logistic Regression (MLR)[7], and knowledge graph embedding (TransE)[11], on STRADS-AP and other baseline frameworks. The evaluation results show that STRADS-AP achieve performance comparable to other baseline frameworks and hand-tuned implementations while simplifying distributed machine learning programming significantly.

- Finally, in Chapter 8, we present conclusion and future work.

Chapter 2

Background

In this chapter, we review distributed machine learning with a focus on consistency models and system supports for distributed machine learning. Most machine learning algorithms and their theoretical analysis assume sequential execution, which means that model update at step (t) is always available for the following update at step $(t+1)$. This sequential execution assumption has imposed serious challenges on parallelizing machine learning workloads in a cluster and limited distributed machine learning to BSP (Bulk Synchronous Parallel) model for a long time[1]. Distributed machine learning with BSP[2] ensures reproduction of sequential execution output, which makes the safety proof of parallel execution trivial, but BSP execution incurs high overheads because of two factors: (1) network communication latency is $10^2 \sim 10^3$ times longer than main memory access latency, which makes synchronization expensive; (2) synchronization once per iteration introduces staleness in parameter values when asynchronous-friendly algorithms[3] are parallelized in BSP model. The strict synchronization in BSP makes it hard to achieve high scalability in a commodity cluster. To address these problems, the system and machine learning communities have presented relaxed consistency models for distributed machine learning – lock-free asynchronous, fully asynchronous, cyclic delay, bounded staleness – and have provided safety proofs for some of them. Their basic idea is to allow trade off between synchronization overheads and staleness of parameter values. Because the choice of consistency model affects system design and training performance, we will review details of

[1] Here, we limit machine learning algorithms to synchronous algorithm. Asynchronous algorithm with BSP will be discussed later.

[2] Distributed machine learning with BSP is a data-parallel machine learning with strict consistency.

[3] Asynchronous-friendly algorithms are algorithms designed for asynchronous mode. These algorithms tend to make much better convergence per iteration in asynchronous computation mode than in synchronous computation mode. See Table 2.1 for definitions of asynchronous and synchronous computation modes

Term	Description
worker	a processing unit (a thread or process)
iteration	one pass over all input data while updating corresponding model parameters. Superscript represents iteration count
minibatch	a partition of input data
clock	a logical clock that measures progress of a worker in terms of iteration or minibatch, locally or globally
subepoch	one pass over a minibatch while updating corresponding model parameters. Superscript represents subepoch count
synchronous execution mode	executes model parameter updates based on the parameter values from the previous iteration. At $iteration^{(t)}$, synchronous computation calculates gradients Δ_{all}^{t} for all model parameters based on model parameter values $P^{(t-1)}$ from $iteration^{(t-1)}$, and then adds Δ_{all}^{t} to $P^{(t-1)}$ to obtain $P^{(t)}$. Note that there is no change to the state of model parameters while calculating Δ_{all}^{t}.
asynchronous execution mode	An iteration consists of many fine-grained updates on model parameters. In an iteration, each update operation makes changes to the model parameter state, and the latest state of the model parameters is immediately available for executing the following update operation.

Table 2.1: Definitions of terminologies. Note that synchronous execution and asynchronous execution in this chapter carry different meanings from what they mean in system literature.

three consistency models commonly used for distributed machine learning in Section 2.1.

The task of developing a distributed machine learning program is divided into two parts: (1) application programming to write application-specific optimization routines; (2) distributed system programming to write system routines that handles essential requirements of distributed computing, such as consistency management, data partitioning, computation parallelization, load-balancing, fault-tolerance, and network communication. The distributed programming part is often too much burden on machine learning programmers[4], and is difficult to do correctly and efficiently. Furthermore, implementing the system routines repeatedly for each application is a big waste of programming efforts. To address this distributed system programming overhead, the system community has developed various distributed programming frameworks – MPI[28, 65, 62], MapReduce[26], Pregel[55], GraphLab[54, 53], PowerGraph[32], Parameter Server[68, 19, 20, 48, 21, 88], FlumeJava[13], Spark[94], TensorFlow[1], Caffe[40], MXNet[16], and PyTorch[69], to name but a few. These frameworks provide different levels of programming abstractions, and many of them are specialized for distributed machine learning; that is, a framework is harnessed with distributed machine learning specific consistency models for achieving

[4] We assume that most machine learning programmers do not have distributed programming experience.

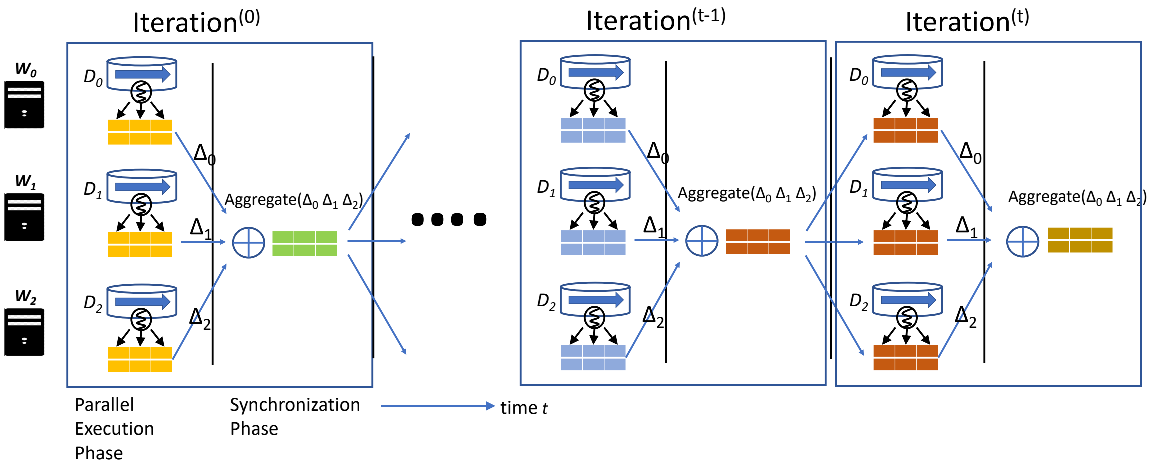


Figure 2.1: Diagram of BSP machine learning

high training performance and/or provides a programming API that simplifies distributed machine learning programming.

In the rest of this chapter, we present how distributed machine learning consistency models and system framework design have evolved in last two decades. For the general audience, Table 2.1 provides definitions of a few terminologies commonly used in this chapter.

2.1 Consistency models in distributed machine learning

In this section, we review three popular consistency models (BSP, fully asynchronous, bounded staleness) in distributed machine learning, which have huge effects on training performance and system design. The training time of a distributed machine learning program is the product of the iteration count until convergence and the average time per iteration, and the consistency model critically affects both factors. The key differences between different consistency models are whether to allow use of stale parameter values for running updates and how to manage staleness of model parameters if staleness is allowed.

2.1.1 BSP Model

Developed originally for parallel simulations, BSP models physical processes that happen concurrently based on global input and then combine changes for the next global time step.

The BSP model guarantees that all updates made between $iteration^{(0)}$ and $iteration^{(t-1)}$ are available for running $iteration^{(t)}$ in all workers. Figure 2.1 depicts iterations from $iteration^{(0)}$ to $iteration^{(t)}$ in BSP machine learning. All workers start $iteration^{(t)}$ with a globally consistent parameter state from $iteration^{(t-1)}$; that is, all workers see all updates made before $iteration^{(t)}$. One iteration of BSP machine learning is divided into two phases, computation and synchronization. During the computation phase, a worker sweeps through an input data partition and calculates updates (i.e. gradients) independently without making changes to the parameter state – updates in current iteration are not available in current iteration, but become available for the next iteration after synchronization. When all workers complete a computation phase, they aggregate local updates and make a globally consistent model parameter state for $iteration^{(t+1)}$. For gradient descent algorithm that runs synchronous computation, distributed machine learning in BSP model gives output equivalent to that of a sequential execution, but it cannot for machine learning algorithms that run asynchronous computation.

Because of its simplicity and deterministic execution, BSP model has been widely used for distributed machine learning for a long time and is well supported by many data processing frameworks, such as MPI [28], MapReduce [26], and Spark [92, 77, 93]. We will discuss these frameworks in Section 2.2. However, it suffers from a few critical challenges in a large cluster [17]. The network communication in synchronization phase is not overlapped with computation phase, and the straggler problem in a large cluster increases the synchronization overhead because workers can start synchronization phase only when all participating workers complete a computation phase and become ready for synchronization. Furthermore, in asynchronous algorithms, synchronization per iteration introduces substantial staleness in parameter values which lowers progress per iteration.

2.1.2 Fully asynchronous model

In a fully asynchronous consistency model, all workers runs independently without coordination, and shared parameter synchronization is performed asynchronously[5] – no global lock step. Updates on a worker are propagated to other workers with delay, and there is no threshold on the delay. In a fully asynchronous model, update computation is overlapped with communication for synchronization, and workers proceed without being blocked by stragglers. This leads to high iteration throughput. However, the unbounded delay on update propagation (i.e. unbounded staleness) causes several problems: (1) unbounded staleness often increases iteration count until convergence substantially, which results in poor training performance; (2) it makes theoretical analysis hard; (3) innocuous changes

[5] Parameter synchronization is performed as a background task without blocking update computation.

in system configurations (i.e. the number of workers, network speed, CPU clocks) and varying system workloads (i.e. network congestion or fluctuation of workloads on shared cluster) affect delays on propagation updates and make it hard to predict training performance – unstable training performance. Fully asynchronous model is implemented in YahooLDA[2] and Hogwild[71].

Efficient implementation of this consistency model could be too much burden on machine learning programmers because it requires understanding of distributed systems and non-trivial engineering efforts. To address this problem, system community has introduced the parameter server architecture that hides details of parameter consistency model from machine learning programmers. Parameter server will be discussed in Section 2.2.4.

2.1.3 Bounded staleness model

In bounded staleness model [80, 36] the staleness of shared model parameters is bounded. It guarantees that all updates made until $subepoch^{(t-s)}$ are available at $subepoch^{(t)}$ where s is a user-configurable staleness threshold. All workers report their progress in terms of clocks[6], and the clock distance between the fastest worker and the slowest worker should be lower than a threshold – the staleness threshold s; that is, $Clock_{fastest} - Clock_{slowest} < s$ is guaranteed. The staleness threshold plays the role of a knob that can be used to trade off subepoch throughput for subepoch count until convergence; that is, small s reduces staleness (i.e higher progress per subepoch) but causes more parameter synchronization overheads (i.e. less subepoch throughput) while large s reduces parameter synchronization overheads (i.e. higher subepoch throughput) but increases staleness (i.e. less progress per subepoch). Tuning staleness threshold that makes good trade-offs leads to high training performance. Furthermore, bounded staleness makes training performance more stable and predictable. Theoretical analysis with bounded staleness machine learning is algorithm-specific. Ho [36][7] and Li [49] present theoretical analysis for stochastic gradient descent and proximal gradient descent with bounded staleness respectively.

Many parameter server works[48, 19, 20] support bounded staleness model by default. We will discuss them in Section 2.2.4.

[6] A worker increments clock count every subepoch.

[7] I participated in design and evaluation of the bounded staleness parameter server project [36] at the early stage of my doctoral research. This experience led to many ideas in this dissertation.

2.2 System supports for distributed machine learning

For simplicity and efficiency of distributed programming, the system community has worked on distributed programming frameworks for a long time. Early works target general programs and tend to provide low-level abstractions while more recent works target more domain-specific applications, such as data processing applications and machine learning applications, with high-level abstractions. In this section, we overview how these programming frameworks have been evolved and how they support the aforementioned distributed machine learning consistency models, with several monumental works (MPI, MapReduce, Spark, parameter servers, GraphLab, TensorFlow).

2.2.1 MPI

MPI (Message Passing Interface)[28] is a standard of communication functions that simplifies implementing various communication/computation patterns commonly found in high performance computing. MPICH [62] and OpenMPI [65] are available as open-sourced implementations. MPI standard is one of the most popular distributed programming standards in the supercomputing community and is also commonly used for distributed machine learning. BSP machine learning applications can be implemented easily using MPI because BSP was invented for supercomputing simulation, which is also iterative update-based (but often not convergent). Specifically, a user can implement BSP machine learning easily using broadcast, reduce, allreduce, and barrier functions in MPI. However, MPI is limited to being a communication abstraction. It lacks distributed shared memory abstraction for input data and model parameters and requires programmers to deal with details of distributed programming, such as data partitioning, parallelization of workloads, fault-tolerance, and load-balancing. Furthermore, it requires programmers to be aware of distributed processing entities and does not provide an illusion of sequential programming.

2.2.2 MapReduce/Spark

MapReduce: Dean [26] presented MapReduce that simplifies distributed programming mainly for data processing applications. In MapReduce, a user writes a program by specifying two functions, Map and Reduce. The map function takes a user-defined function (MapUDF) and a sequence of key/value pairs as input, applies MapUDF to all elements of the sequence, and generates a set of intermediate key/value pairs. Given a user-defined binary function and the set of intermediate key/value pairs, Reduce function [8] merges all intermediate key/value pairs associated with the same intermediate key. Once a program is written using map and reduce functions, the MapReduce runtime system automatically parallelizes the program in a cluster. It is responsible for data partitioning, automatic parallelization, load-balancing, and fault-tolerance. MapReduce provides distributed shared data abstraction by input and output files in a distributed file system.

MapReduce programming model fits well into BSP model; that is, computation phase and synchronization phase in BSP can be implemented easily by Map and Reduce respectively. Compared to MPI, MapReduce simplifies distributed machine learning programming significantly. It gives an illusion of sequential programming. The aforementioned details of distributed programming are hidden from a programmer, and deterministic execution simplifies debugging. Chu [78] and Mahout project [5] use Hadoop [4], an open-sourced MapReduce implementation, for implementing various machine learning algorithms. However, one well-known disadvantage of using MapReduce for running iterative machine learning workloads is that it incurs disk I/O overhead for reading a whole training data set in every iteration.

Spark: inherits the functional style programming model from MapReduce but improves runtime performance significantly for iterative workloads. Spark provides RDDs (Resilient Distributed Datasets) – an immutable distributed memory abstraction that supports in-memory computation – to store a large data set in a cluster and a rich set of functions that operate on RDDs in a cluster. Spark RDDs can persist in main memory across multiple operations so that Spark saves disk I/O and improves performance significantly for iterative workloads. In Spark, a user writes an application as a driver program, and the Spark runtime system automatically parallelizes the driver program. The Spark runtime system is responsible for distributed programming details mentioned above. Particularly, for RDD fault-tolerance, the Spark runtime system keeps track of each RDD's lineage (or provenance) information, which records input RDD(s) and operators to generate the target RDD chunk. On a machine failure, RDD lineage information allows for recomputing only the lost chunks – a fine-grained fault recovery.

Driver programming, integration with the Scala interpreter and deterministic execution simplify distributed machine learning programming, and caching an RDD in main memory over multiple iterations allows Spark to achieve $10 \sim 10^2$ times faster training than Hadoop. The Spark community has developed MLlib [57] – a collection of Spark machine learning applications – and optimization method library that users can compose to solve their optimization problems. Again, the BSP consistency model can be easily implemented using Spark.

[8] User defined function for Reduce should be commutative and associative.

These functional style programming frameworks simplify distributed machine learning programming significantly. However, in MapReduce/Spark, it is hard to support flexible distributed machine learning consistency models mentioned in section 2.1.2, 2.1.3 because of their constrained distributed data abstractions (i.e. immutable input file in MapReduce, immutable RDD in Spark) and deterministic execution.

2.2.3 GraphLab

Gonzalez et al. [32, 53, 54] presented GraphLab, in which data is encoded as a input graph and computation routines as operations on the input graph. A computation routine is associated with a vertex, and data (input data and model parameters) is associated with edges and vertexes. Computational dependencies are also encoded in the input graph, in which an edge $e_{i,j}$ represents computational dependency between two vertexes V_i, V_j. A user writes an application program as a vertex program which is executed for each vertex, and the GraphLab runtime system automatically parallelizes these executions in a cluster – concurrently executes the vertex program on multiple vertexes in a cluster. The runtime system supports both synchronous computation and asynchronous computation. In asynchronous execution, locking protocols can ensure serializability and provides three different consistency models, full consistency (the strongest consistency), edge consistency, and vertex consistency (the most relaxed consistency), to meet different consistency requirements of different applications and achieve best parallel performance.

The GraphLab abstraction is powerful and simplifies distributed programming for many graph mining and graphical model-based machine learning applications, but converting a sequential algorithm (program) into a vertex program often requires significant cognitive effort. And use of a predefined consistency model based on graph structure and a serializability guarantee make it hard to make flexible trade-offs between parallelism and consistency, which leads to poor performance when a machine learning problem has a dense dependency structure.

2.2.4 Parameter servers

The parameter server architecture was proposed to scale out data-parallel machine learning by exploiting trade-offs between staleness and network communication overhead. For parameter sharing, it provides an ML-specialized distributed shared memory abstraction that supports aforementioned staleness-based consistency models.

In 2010, Russel [68] proposed the parameter server architecture in Figure 2.2 and pro-

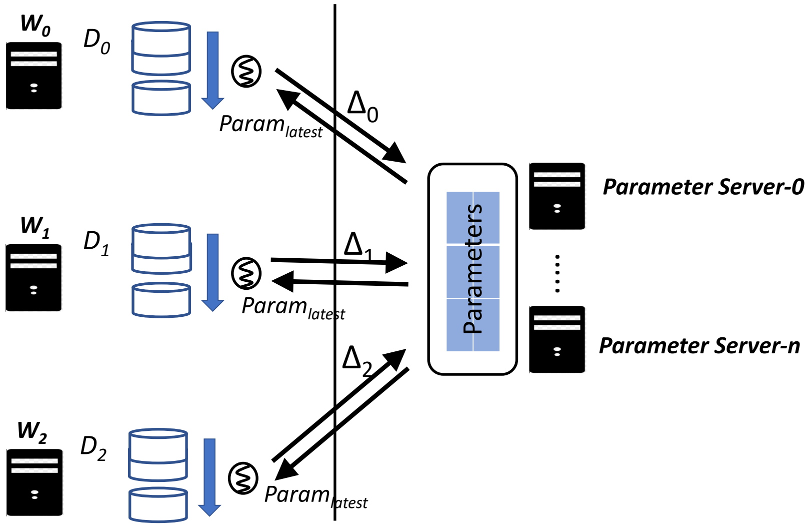


Figure 2.2: Diagram of parameter server architecture

gramming model, but it did not include machine learning optimizations such as relaxed consistency management. In 2012, Amr [2] presented fully asynchronous parameter server for implementing YahooLDA – a large scale LDA solution. In YahooLDA, each worker computes LDA model updates with its own data partition independently. Parameter synchronization is performed by a background client thread in each worker. The client thread synchronizes local parameter updates with global parameter state in parameter servers in background without blocking the progress of the workers. This achieves high iteration throughput, but staleness (inconsistency) of parameter values in each worker is arbitrary and unbounded. This unbounded staleness causes several problems as mentioned in Section 2.1.2.

To address these problems, Ho et al. [36, 19] proposed SSP (Stale Synchronous Parallel) parameter server that implements bounded staleness consistency in 2013. Li [48] presented Parameter Server, a specific framework named the same as the general technique, that supports all three consistency models and DHT (Distributed Hash Table)-based fault-tolerance in 2014.

Cui [20] presented the IterStore parameter server that prefetches necessary parameter values into workers from remote parameter servers based on parameter access profiling. Because the same memory access pattern is repeated over many iteration in machine learning, IterStore runs profiling once before it starts ML computations and reuses the profiling information for the following iterations; that is, profiling overhead is amortized over many iterations.

Wei [88] presented a managed communication parameter server that prioritizes more important parameter updates when selecting parameter updates to communicate and maximizes update communication without overusing network bandwidth. It reduces staleness (inconsistency) of shared parameters and improves convergence per iteration without lowering update throughput, which improves training performance.

The parameter server programming model simplifies data-parallel machine learning significantly by relieving a programmer of writing codes to manage consistency model. However, it does not give an illusion of sequential programming and leaves many details of distributed programming, such as data partitioning, task parallelization, load balancing, in the programmer's hands.

2.2.5 Dataflow framework

TensorFlow: In TensorFlow[1], a programmer puts all data and model parameters in tensors and specifies machine learning model using numerical operators and tensors, which builds a dataflow graph where a vertex represents a computation operation, and an edge represents flow of tensors. The programmer specifies computing devices for computation statically. Once a program is written in this dataflow programming model, the TensorFlow runtime system automatically differentiates the specified model – auto-differentiation[9] – and updates model parameter tensors – auto-gradient updates. At runtime, TensorFlow schedules computation in a way to maximize parallelism based on the dataflow graph. Each operator is implemented to utilize vector instructions provided by modern computing devices (CPU or GPU) or tensor (matrix) instructions supported by Volta tensor core [56] to accelerate computation. Compared to previous frameworks, TensorFlow simplifies machine learning programming more because it allows a machine learning programmer to skip deriving an update algorithm from the model and writing the code for the derived update algorithm for the cases where the automatic method is appropriate. For neural network models that are composed of well-established operators and use gradient descent-based optimization algorithms, TensorFlow gives enormous benefits; data flow programming model improves programming productivity significantly; and well-tuned standard operators using modern CPU/GPU features achieves high training performance.

However, TensorFlow has few critical limitations when we use it for general machine learning programs. First, TensorFlow's productivity benefit from auto-differentiation and auto-gradient updates is not available for machine learning researchers who want to modify machine learning optimization algorithms. Searching for a more efficient update algorithm

[9] For appropriate problems, auto-differentiation is a huge benefit for programmers

(that is, deriving more efficient update algorithm) for a given machine learning model is one of the major research directions in the machine learning research community. With TensorFlow, these people would not benefit from TensorFlow's auto-differentiation/gradient updates, but would need to develop new kernel operators to implement and test their new algorithms. Unfortunately, programming TensorFlow kernel operators requires substantial engineering efforts, so much so that most system engineers are reluctant to attempt it. Second, TensorFlow's high-performance benefits might not be available for many sparse or asynchronous machine learning problems. TensorFlow utilizes vectorized instructions with assumption that computation is dense and runs in synchronous computation mode. However, many ML problems, such as recommendation and sparse algorithms, have sparse computation, and make higher convergence per iteration in asynchronous computation mode than in synchronous computation mode. In such problems, TensorFlow system throughput suffers.

Tensor Comprehensions: Tensor Comprehensions (TC) [86] is a domain specific language that aims to (1) facilitate programming a custom operator or a new layer and (2) achieve further performance optimizations across operators for deep learning models (i.e. convolutional and recurrent networks) on top of existing deep learning frameworks such as Caffe2, PyTorch and TensorFlow. TC allows an ML programmer to express a custom operator or a new layer (computation on multi-dimensional arrays) using an elegant notation that uses the Einstein notation (a.k.a Einstein summation convention). TC's notation allows element-wise accesses, and the user-defined custom operator written in TC can be invoked like the operators that the underlying framework provides by default. At runtime, a polyhedral JIT (Just-In-Time) compiler lowers TC code into Halide-IR (Intermediate Representation)[70], then to polyhedral-IR. Then, polyhedral transformation and mapping to GPU are performed by the polyhedral JIT compiler, and CUDA kernel code is finally generated, which demonstrates the suitability of polyhedral framework for the deep neural network domain. For reducing the polyhedral JIT compilation overhead and achieving higher performance, TC provides a compilation cache and an autotuner. For a given TC code, the compilation cache stores generated CUDA kernel code with information of input data (i.e. size and shape), environment (i.e. information about GPU architecture), and optimization options. The autotuner starts with candidate configurations that are from similar TC code snippets and tunes configurations such as tile, block, grid sizes, admissible schedules, shared or private memory usage using a genetic algorithm.

TC simplifies programming a custom operator and a new layer that do not fit high performance library calls, and the automatically generated CUDA kernel code achieves comparable performance to hand-crafted code that uses high-performance libraries calls. TC saves significant engineering cost for appropriate deep learning problems. However, it still has performance limitations like TensorFlow as we mentioned above when targeted machine learning applications require sparse asynchronous computation.

Chapter 3

Computational Efficiency of Parallel ML

In this chapter, we investigate two efficiency challenges of parallel machine learning with three popular applications and show their impacts on computational efficiency. To address these challenges and achieve high computational efficiency in distributed machine learning, we present the SchMP (Scheduled Model Parallel) programming approach. SchMP has multiple model update scheduling strategies. We also present the STRADS runtime system that executes SchMP machine learning applications efficiently in a cluster. In summary, this chapter

- Explores two efficiency challenges in parallelizing machine learning – model dependency and uneven convergence – and investigates their impacts on computation efficiency in three well-established ML applications, $l1$-regularized sparse regression (Lasso), topic modeling (LDA), stochastic gradient descent matrix factorization (SGDMF).
- Presents SchMP with multiple possible scheduling strategies.
- Presents the core of the STRADS runtime system that is shared across multiple SchMP scheduling implementations.

Mathematical notation: For elegant and concise description, we use mathematical notation throughout this chapter together with verbal description. As a baseline, we describe a generic iterative-convergent machine learning algorithm using mathematical notation in equation (3.1).

$$A^{(t)} = F(A^{(t-1)}, \Delta(D, A^{(t-1)})), \tag{3.1}$$

where index t refers to the current iteration, A refers to the model parameters, $\Delta()$ is the model update function, and F is a summation function on $A^{(t-1)}$ and $\Delta()$.

We will use this baseline expression in explaining challenges of parallel machine learning and extend it incrementally to describe various SchMP scheduling strategies.

3.1 Efficiency challenges of parallel machine learning

In this section, first, we review two parallel machine learning approaches – data-parallel and model-parallel – and explain their challenges and discuss potential to improve computational efficiency of distributed machine learning.

3.1.1 Data-Parallel

Data-parallel is a common approach that allows for parallelizing machine learning computation over input data. Data-parallel machine learning partitions input training data D into P partitions D_1 ... D_P and assigns data partitions to P workers W_1 ... W_P in a cluster. At runtime, a worker W_p applies the update function Δ to a data partition D_p, makes partial updates, then synchronizes partial updates to aggregate them. Data-parallel machine learning can be expressed as

$$A^{(t)} = F(A^{(t-1)}, \textstyle\sum_{p=1}^{P} \Delta(A^{(t-1)}, D_p)) \tag{3.2}$$

In the data-parallel approach, we assume that outputs of update function Δ are aggregated via simple summation (i.e. aggregation operation is associative and commutative), which is common in stochastic gradient based algorithms. This additive property allows outputs of multiple updates associated with the same parameter to be aggregated in each worker before synchronization over network – like MapReduce allows combiners to run local reductions between Map and Reduce functions to reduce data transfer. This additive property of updates is a foundation that allows relaxed consistency models and parameter server architectures to improve data-parallel machine learning performance.

To improve training performance by trade-offs between communication cost and staleness, we add relaxed consistency models (i.e. fully asynchronous, bounded staleness etc.), which allow for staleness on shared parameters, to data-parallel machine learning. This can be expressed as

$$A^{(t)} = F(A^{(t-j)}, \textstyle\sum_{p=1}^{P} \Delta(A^{(t-j)}, D_p)) \tag{3.3}$$

where $j \leq s$ and s is staleness threshold.

The theoretical assumption regarding correctness of data-parallel approach with staleness is that the data samples – random variables in statistics parlance – are independent and identically distributed (i.i.d.) and enable a simple aggregation of sub-updates. Although each machine suffers from staleness of model-parameters, which leads to inaccurate update results (i.e. inaccurate gradient calculation), machine learning algorithms can converge as long as staleness is not arbitrarily large, thanks to the error-tolerance of the machine learning algorithm. Theoretical correctness guarantees are found in [71, 45, 97]. The logic behind the correctness proof is that the potential errors from stale model parameters by sub-updates offset each other, or are small enough to be tolerated when aggregated thanks to the nature of optimization algorithm making small updates in each iteration.

In data-parallel machine learning with the relaxed consistency models, training performance is determined by trade-offs between communication cost for synchronizations and staleness. Though relaxed consistency models and parameter servers improve data-parallel training performance by making a balance between staleness and communication cost, data-parallel machine learning still has residual staleness on shared parameters, which lowers progress per iteration. As a result, data-parallel machine learning tends to take longer iterations to achieve convergence than ideal serial implementations.

3.1.2 Model-Parallel

In contrast to the data-parallel approach, the model-parallel approach partitions model parameters over workers and lets each worker update a set of model parameters based on the whole data or a partition of data, as shown in Figure 3.1. If necessary, model repartitioning is conducted at runtime. [1] Model-parallel machine learning extends equation (3.1) to the following form:

$$A^{(t)} = F(A^{(t-1)} + \textstyle\sum_{p=1}^{P} \Delta_p(D, A^{(t-1)}, S_p(D, A^{(t-1)}))), \tag{3.4}$$

[1] Some communication for model repartitioning is necessary, but it can be much less frequent than that in data parallel model.

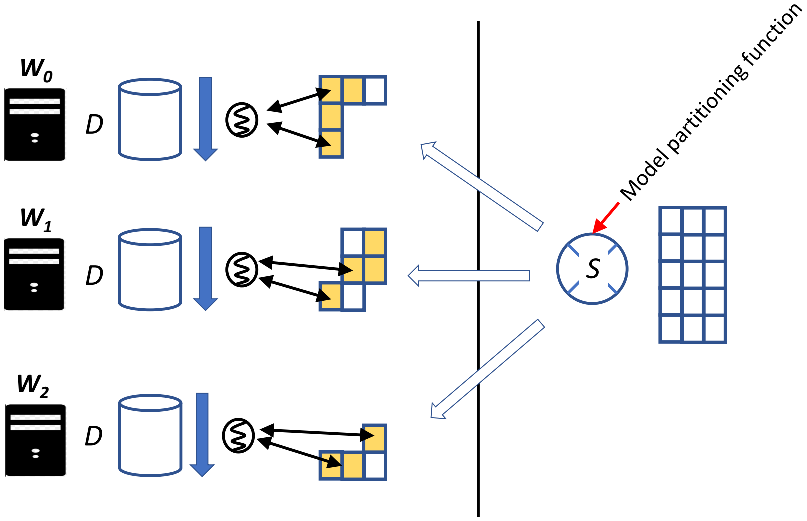


Figure 3.1: Model-Parallel ML: Model parameters are partitioned over machines in a cluster by a model partitioning function S — which could be implemented using various strategies to improve training performance, and a machine gets an exclusive access to its assigned model parameter partition.

where $\Delta_p()$ is the model update function executed at parallel worker W_p. The "schedule" $S_p()$ outputs a subset of parameters in A (and a subset of update operations that are associated with the subset of parameters), which tells the p-th parallel worker which parameter updates it should execute *sequentially* (i.e. workers may not further parallelize within $S_p()$). In naive model-parallelism, the scheduling output of S is determined by data partitioning or random model-partitioning. Since the data D is immutable, we leave it out from the equation for clarity:

$$A^{(t)} = F(A^{(t-1)}, \textstyle\sum_{p=1}^{P} \Delta_p(A^{(t-1)}, S_p(A^{(t-1)}))). \tag{3.5}$$

Unlike the data-parallel approach, in which the data-partitioning scheme determines model parameter update partitioning implicitly, model-parallel has potential to improve training performance further by making intelligent model parameter update partitioning decisions in function S which eliminates or further reduce staleness. To achieve this, the update partitioning function should consider computational dependencies among model parameter updates and be able to make schedule plans that do not run conflicting update operations concurrently. Since the dependencies among model parameter updates are determined by update operation's read/write accesses to model parameters only[2], we refer to

[2] Input data does not affect dependencies among updates because input data is read only in most machine

this challenge as **model-dependency challenge**.

A flexible update partitioning in the model-parallel approach allows us to exploit another important property of machine learning – uneven convergence of model parameters. In many machine learning algorithms, some model parameters require far more iterations than others for convergence. Thus, these few parameters bottleneck the entire machine learning training, which we refer to as an **uneven convergence challenge**. If model parameter update partitioning function S identifies these slowly converging model parameters at runtime and grants more updates to them, it could improve the progress per iteration further.

In the rest of this chapter, we will see the impacts of these two challenges on training performance in three distributed machine learning applications (Lasso, LDA, SGDMF), in which model parameter update partitioning is done at random or determined implicitly by a data partitioning scheme. Then, we will extend the model-parallel approach to support intelligent scheduling, SchMP, which handles these challenges appropriately to achieve progress per iteration close to or better than that of ideal sequential execution.

3.1.3 Example I: Lasso

Sequential Lasso: Lasso is a ℓ_1-regularized least-squares regression that is used to identify a small set of important features from high-dimensional data. Lasso is an optimization problem:

$$\underset{\beta}{\text{minimize}} \qquad \frac{1}{2}\sum_{i=1}^{n}\left(y_i - h_\beta(\mathbf{x}_i)\right)^2 + \lambda\|\beta\|_1 \tag{3.6}$$

where ℓ_1-regularizer $\|\beta\|_1 = \sum_{j=1}^{d}|\beta_j|$ and λ is a preset user parameter that controls the number of zero entries in β. X is an n-by-m design matrix where rows represent samples and columns represent characteristics, and y is an n-by-1 observation vector where a row represents a label for a sample of X. β is an m-by-1 coefficient vector.

During the training phase, we learn the β vector for a given X, Y by solving equation 3.6. The update rule in Lasso is:

$$\beta_j^{(t)} \leftarrow \mathcal{S}(\mathbf{x}_j^T\mathbf{y} - \sum_{k\neq j}\mathbf{x}_j^T\mathbf{x}_k\beta_k^{(t-1)}, \lambda), \tag{3.7}$$

where $\mathcal{S}(\cdot, \lambda)$ is a soft-thresholding operator [29].

learning algorithms.

Algorithm 1 Parallel Lasso

```
while Until converge do
    Chooses random subset of $\beta$ weights in $\{1, 2, 3, .., n\}$
    In parallel on processors $p = 1..P$
        Take assigned coefficient j
        $\beta_j = \Delta_p(j, \beta, X, Y, \lambda)$
end while
Function $\Delta_p(j, \beta, X, Y, \lambda)$
    $\beta_j$ = $\leftarrow \mathcal{S}(\mathbf{x}_j^T \mathbf{y} - \sum_{k \neq j} \mathbf{x}_j^T \mathbf{x}_k \beta_k, \lambda)$
```

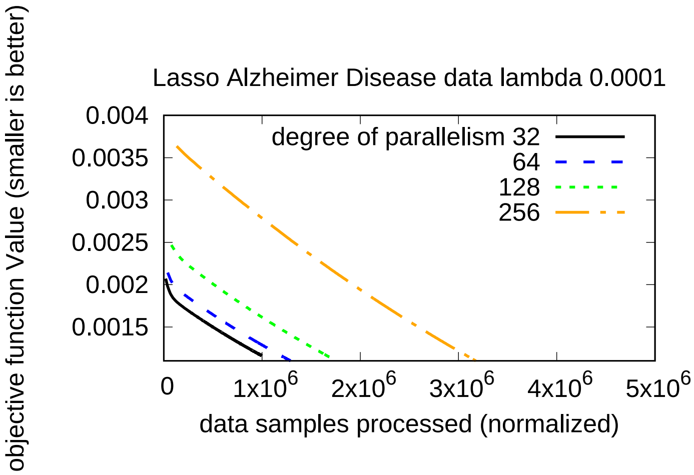


Figure 3.2: Random Model-Parallel Lasso: Objective value (the lower the better) versus processed data samples, with 32 to 256 workers performing concurrent updates. Under naive (random) model-parallel, higher degree of parallelism results in worse progress.

Lasso is an inherently serial algorithm, since the update operation of updating β_k takes all β_i where $i = 1, .., n$ and $i \neq k$ in equation(3.7).

Parallel Lasso: Lasso can be parallelized by updating multiple coefficients in parallel [12], which might cause inaccuracy in parameter updates but will still converge when dependencies among selected coefficients are not arbitrarily large due to the error tolerance property of machine learning. We implement a distributed parallel Lasso illustrated in Algorithm 1 that updates a set of randomly selected model parameters in each iteration.

Model dependency challenge in parallel Lasso: In practice, the random selection of model parameters[3] does not always fill the set S (a set of model parameter to update in

[3] In Lasso, a parameter update operation updates exactly one model parameter so that the selection of

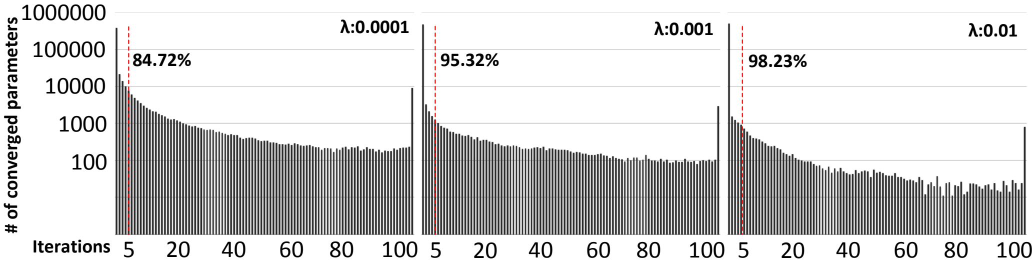


Figure 3.3: Uneven Parameter Convergence: The number of converged parameters at each iteration, with different regularization parameters λ. Red bar shows the percentage of converged parameters at iteration 5.

parallel per iteration) with independent model parameters, particularly when the design matrix X is dense[4]. Updating dependent model parameters introduces errors (inaccuracy) to the update results, which lowers statistical progress per iteration and takes more iterations to converge. In some cases, it can cause the algorithm to diverge.

Figure 3.2 shows progress per data processed (= progress per update) of parallel lasso with an Alzheimer's Disease (AD) data set [95]. The Y axis represents optimization error (smaller equals better progress) and the X axis represents the amount of data points processed (approximately equal to the amount of computation cycles consumed). Figure 3.2 shows that increasing the degree of parallelism diminishes the gain (progress) per computation. Note that, in order to get the objective value of 0.001, the experiment with 256 degrees of parallelism requires about 3 times the computation of the experiment with 32 degrees of parallelism.

Uneven convergence rate challenge in Lasso: In Lasso and other regression applications, a column corresponds to a model parameter. In the optimization algorithm perspective, such popular columns are model parameters with dependency on a large number of other model parameters. These require more iterations to achieve convergence because they are frequently influenced by other dependent model parameters. On the other hand, unpopular model parameters that take the major portion of model parameters require fewer iterations to converge. Figure 3.3 shows histograms of the number of iterations to convergence for approximately a half million parameters in Lasso with the AD data set using three differ-

update parameters here means the selection of update operations corresponding to those model parameters.

[4] In more details, the dependency between two model parameters β_i, β_j in Lasso can be defined as Pearson correlation of two columns $X[:, i], X[:, j]$ of input X matrix corresponding to β_i, β_j. Therefore, the random parameter selction would fill the set S with more dependent model parameters when design matrix is dense.

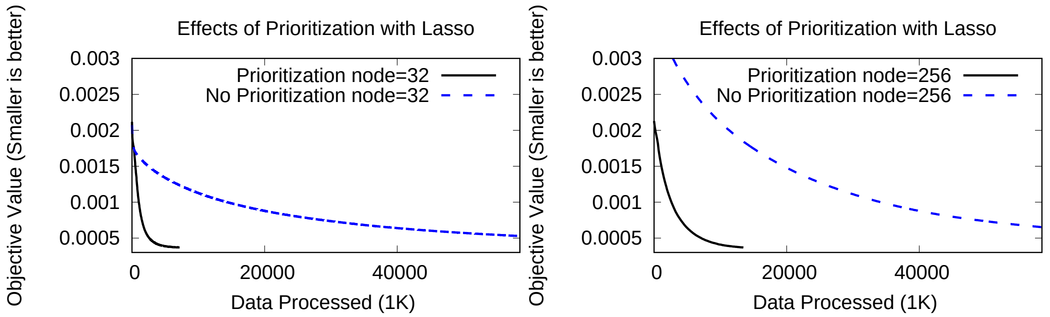


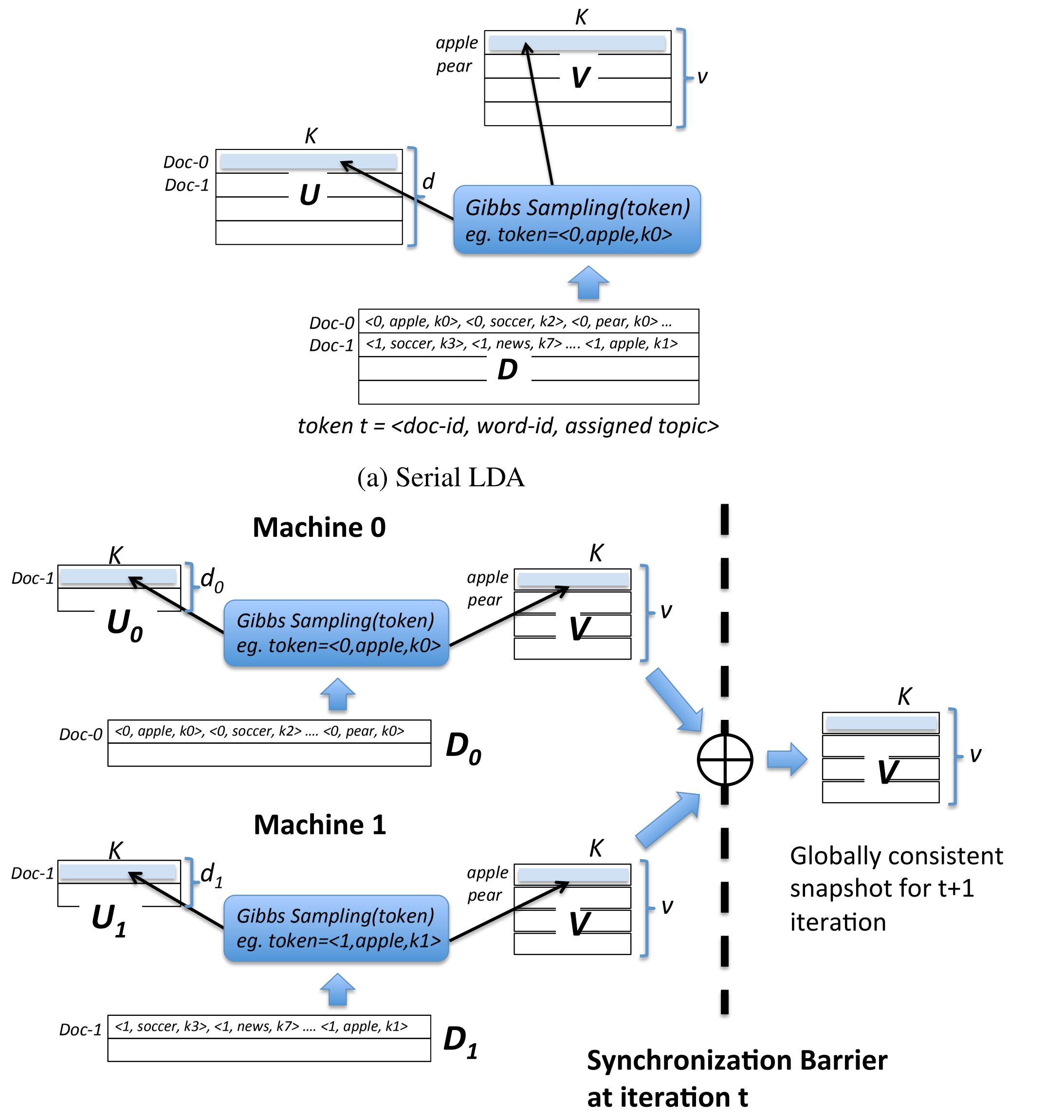
Figure 3.4: Effects of Prioritization with Lasso: Objective value (lower the better) versus processed data samples, with prioritization enabled/disabled on 32 and 256 nodes. In both experiments, prioritization based on convergence status of model parameters improve performance significantly.

ent regularization parameter (λ) values. It shows that about 85 percent, 95 percent, and 98 percent of parameters require less than five iterations to converge respectively with λ values of 0.0001, 0.001, 0.01. Because updating parameters that are already converged or very close to the converged value does not contribute much to the overall progress, adapting computational power to the convergence rate of parameters is essential to improving progress per update.

Figure 3.4 shows expected improvement on progress per iteration when the uneven convergence is appropriately considered duing model parameter selection — which gives more update chances to unconverged parameters and skips updating fully converged parameters.

3.1.4 Example II: LDA topic modeling

Sequential LDA: LDA is a hierarchical Bayesian model that considers each document as a mixture of K topics, where each topic is defined as a multinomial distribution over the vocabulary. The main goal of LDA is to infer the underlying topics from a given set of documents d. Statistically, this is equivalent to inferring the posterior distribution, which can be efficiently approximated by the Gibbs sampling algorithm. Figure 3.5a shows a sequential implementation of Gibbs sampling LDA with three data structures: document-topic table denoted as U, word-topic table denoted as V, and document collection denoted as D. U is an N-by-K array where N is the number


K
apple
pear
V
v
K
Doc-0
Doc-1
U
d
Gibbs Sampling(token)
eg. token=<0,apple,k0>
Doc-0 <0, apple, k0>, <0, soccer, k2>, <0, pear, k0> …
Doc-1 <1, soccer, k3>, <1, news, k7> …. <1, apple, k1>
D
token t = <doc-id, word-id, assigned topic>
Machine 0
K
Doc-1
d_0
U_0
Gibbs Sampling(token)
eg. token=<0,apple,k0>
K
apple
pear
V
v
Doc-0 <0, apple, k0>, <0, soccer, k2> …. <0, pear, k0>
D_0
Machine 1
K
Doc-1
d_1
U_1
Gibbs Sampling(token)
eg. token=<1,apple,k1>
K
apple
pear
V
v
Doc-1 <1, soccer, k3>, <1, news, k7> …. <1, apple, k1>
D_1
K
V
v
Globally consistent snapshot for t+1 iteration
Synchronization Barrier at iteration t

(a) Serial LDA

(b) Parallel LDA

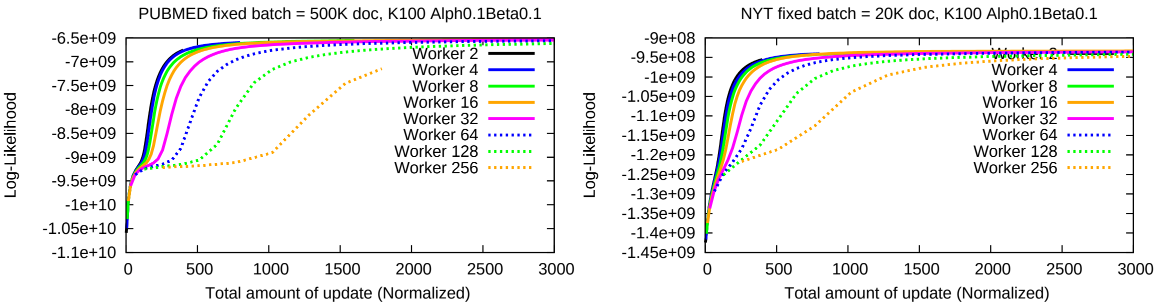
Figure 3.5: Input data and model parameter access pattern of Gibbs sampling algorithm in LDA

of documents, and K is the number of topics. V is a v-by-K array where v is the size of the vocabulary (=dictionary size). D is a collection of documents, in which each document contains tokens and tokens' topic assignments. For simplicity, we omit the 1-by-K topic summary vector. The unit of atomic update operation is a token. For each token topic indicator z_{ij} (document i, j-th word position) with observed word d_{ij} (this is word id, an integer representing some word, like "soccer" or "apple"), the update operation makes a temporary probability vector with length K, based on information in the row $V[d_{ij}][]$ and the column $U[i][]$. From this probability vector, a new topic assignment "newtopic" is sampled, and the old topic assignment is recorded: oldtopic=z_{ij}. We then update $z_{ij} = newtopic$, and manipulate the doc-topic and word-topic tables to reflect this change: decrement both $V[d_{ij}]$[oldtopic] and $U[i]$[oldtopic] by one and increment $U[i]$[newtopic] and $V[d_{ij}]$[newtopic] by one.

Gibbs sampling for LDA is inherently a serial algorithm because of the dependency on other tokens — when we sample a new value for z_{ij}, we change U, V, which in turn affects other z_{ij}.

Parallel LDA: We can parallelize LDA over the document collection, while ignoring the dependency among tokens. Figure 3.5b shows parallel LDA with two machines in BSP (Bulk Synchronous Parallel) style. A machine p is assigned a partition D_p of the document collection, and keeps a partial document topic table U_p corresponding to D_p (since U_p is only ever accessed by machine p). That is, these two data structures D_p, U_p will be dedicated to each worker. On the other hand, the word topic table V need to be shared across workers since all workers access the table to make the probability vector for sampling. In our parallel LDA illustrated in the figure 3.5b, each machine keeps a local copy of the whole word topic table V. While processing a batch (iteration), each worker repeats sampling with its own local word topic table V and updates its own local copy without communication with other workers. At the end of an iteration, all workers synchronize on the word topic table, meaning that the deltas of each worker's word topic table are aggregated to make one globally consistent view of the word topic table. The new word topic table V is copied to all workers before the next iteration starts.

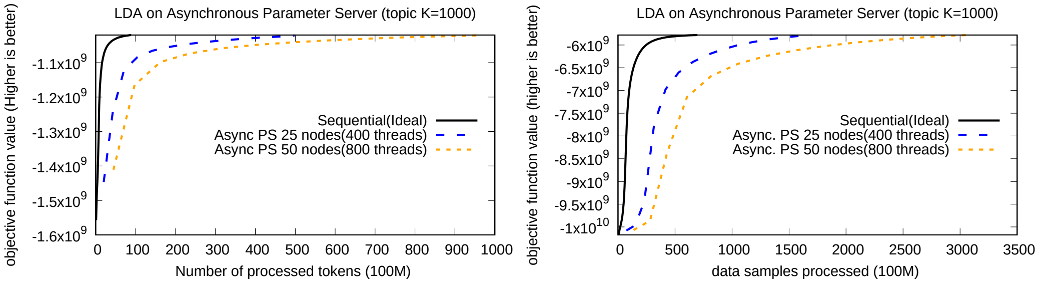
Model dependency challenge in LDA: The parallel LDA in Figure 3.5b inevitably introduces parallel errors since workers sample using the inaccurate (stale) word topic table V, which does not reflect other workers' changes during a batch. If batch size is fixed, the degree of inaccuracy increases as the number of workers increases. The use of a smaller batch size helps reduce inaccuracy of the word topic table V. However, it is not possible to shrink the batch size to an arbitrarily small number, due to the synchronization and communication cost in a distributed environment. The larger degree of parallelism increases the amount of work done per second but increases the amount of inaccuracy on the word



(a) Pubmed data with Topic K = 100

(b) NYT data with Topic K = 100

Figure 3.6: LDA topic modeling on BSP simulation: reports convergence per update. Note that larger degree of parallelism leads to less progress per update due to model dependency.



(a) NYT Data with Topic K=1000

(b) PubMed Data with Topic K=1000

Figure 3.7: LDA topic modeling on asynchronous parameter server: reports convergence per processed data (= progress per update). Note that larger degree of parallelism leads to less progress per update due to model dependency.

topic table V, which diminishes progress (gains) per iteration. Figure 3.6 shows progress per normalized amount of work completed. The experiments are conducted with two different data sets, NYTimes and PubMed. In both experiments, synchronization happens when all workers complete updates for about 10 percent of all tokens. The results show that progress per computation diminishes as the degree of parallelism increases, aggravating the staleness on the word topic table V in worker machines.

In addition to data-parallel LDA with fixed interval synchronization, we conduct a fully asynchronous parameter server LDA experiments using open sourced implementation by Yahoo and present results in Figure 3.7. Compared to sequential executions (ideal), asynchronous parameter server LDA runs 9 times more iterations on 800 workers with NYT data set and 4 time more iterations on 800 workers with PubMed data set until reaching the same convergence points of sequential executions.

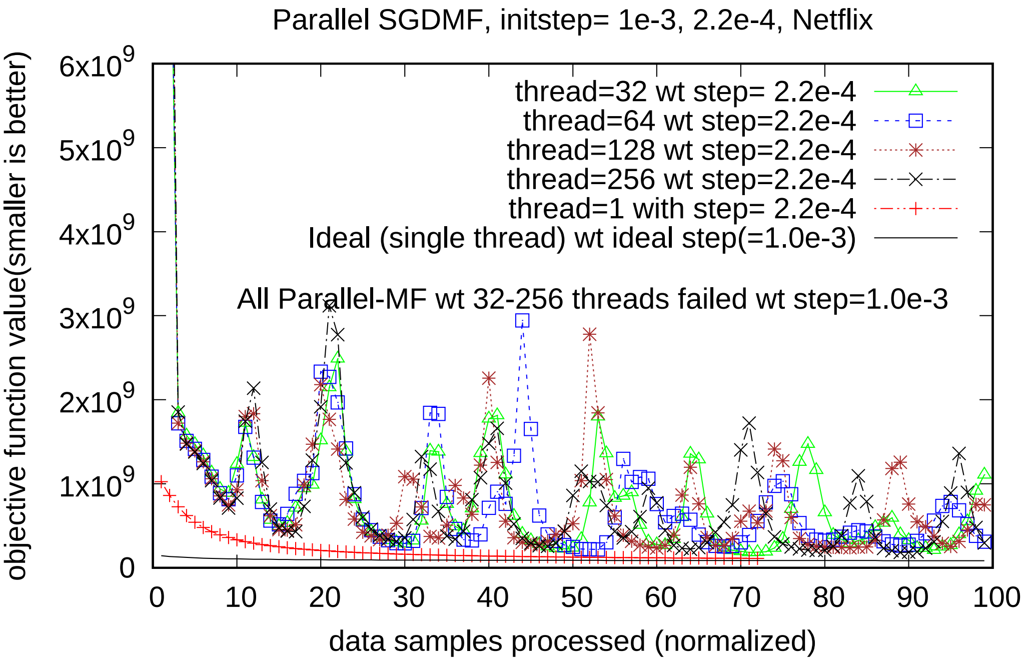


Figure 3.8: Parallel SGDMF in BSP with Netflix data, rank size=1000 : Objective value (lower the better) versus iteration. Parallel SGDMF makes it hard to use optimal initial step size=1.0e-3 that sequential execution allows. Progress per iteration of parallel SGDMF with initial step size(2.2e-4) is lower that that of sequential execution, and parallel SGDMF's convergence curves are unstable.

3.1.5 Example III: SGDMF

Sequential SGDMF: Matrix factorization learns user's preferences over all products from an incomplete rating dataset represented as a sparse matrix $A \in \mathbb{R}^{M \times N}$ where M and N are the number of users and products, respectively. It factorizes the incomplete matrix A into two low-rank $W \in \mathbb{R}^{M \times K}$ and $H \in \mathbb{R}^{N \times K}$ matrices such that $W \cdot H^T$ approximates A. Stochastic gradient descent algorithm for MF (SGDMF) in Algorithm 7 iterates over the ratings in the matrix A. For each rating $r_{i,j}$, it calculates gradients $\Delta W[i]$, $\Delta H[j]$ and adds the gradients to $W[i]$, $H[j]$, respectively.

SGDMF is a sequential algorithm since result of processing rating r_t are immediately available for processing the following rating rating r_{t+1}.

Parallel SGDMF: We implement parallel SGDMF in a similar way of parallel LDA in Figure 3.5. Parallel SGDMF partitions input rating data A into A_p and user matrix W into W_p across P machines, as we did on document collection D and document topic table U in parallel LDA in Figure 3.5. And H parameter table is copied on each worker.

While processing a batch (iteration), a machine M_p processes ratings of a partition of A_p independently while reading/writing to W_p and its own local copy of H. At the end of batch, all workers synchronize local copies of H to make a globally consistent state of H for the next batch processing.

Model dependency challenge in SGDMF: Unlike previous examples, the training performance of SGDMF is sensitive to initial step size(=initial learning rate); that is it's important to find proper initial step size. Empirically, larger step size leads to better progress per iteration in SGDMF, but use of too large initial step size causes divergence or makes poor performance. In our experiments, we first find the best initial step size in sequential execution — 1.0e-3 — and runs parallel SGDMF with the same step size. However, all parallel SGDMF experiments diverge. Then, we shrink the initial step size to 2.2e-4, which is the best initial step size for parallel SGDMF. Compared to the experiment with best initial step size=1.0e-3, the experiment with initial step size=2.2e-4 achieves lower progress per iteration, which means that model dependency makes it hard to use optimal step size that sequential execution allows. With initial step size=2.2e-4, parallel executions on 32 to 256 threads takes longer iterations than sequential execution while showing unstable convergence curve as shown in Figure 3.8.

3.2 Scheduled Model Parallel

In this section, we present a programming approach, SchMP (Scheduled Model-Parallel) that address model dependency and uneven convergence challenges by controlling the way of partitioning model parameter updates and adapting computation frequency per model parameter to its convergence progress. We explore various form of model-parallel and present four scheduling strategies for SchMP.

Usually, convergence per update of sequential execution is ideal because sequential execution does not cause numeric errors. Therefore, we often use the serial algorithms' convergence rate per update as an ideal baseline. We have two goals with SchMP. First, we aim to improve convergence per update of parallel ML to be close to that of sequential ML by scheduling model-dependency. Second, we try to further improve the convergence rate by prioritizing model parameters based on the convergence state of individual model parameters. Relating back to the general model-parallel equation (3.5), SchMP harnesses scheduling function S with model dependency checking and model parameter prioritization. Sometimes, it is neither practical nor possible to find a "perfect" parallel execution scheme for a machine learning algorithm, which means that some dependencies will be violated, leading to incorrect update operations. But, unlike classical computer science algorithms where incorrect operations always lead to failure, iterative-convergent ML programs (also known as "fixed-point iteration" algorithms) can be thought of as having a buffer to absorb inaccurate updates or other errors: they will not fail as long as the buffer is not overrun. Even so, there is a strong incentive to minimize errors; the more dependencies the system finds and avoids, the more progress the ML algorithm will make each iteration. Unfortunately, finding those dependencies may incur non-trivial computational costs, leading to reduced iteration throughput. Because an ML program's convergence speed is essentially progress per iteration multiplied by iteration throughput, it is important to balance these two considerations.

Below, we explore this idea by explicitly discussing some variations within model-parallel in order to expose possible ways by which model-parallel can be made more efficient.

3.2.1 Variations of model-parallel

We restrict our attention to model-parallel programs that partition M model parameter updates across P workers in an approximately load-balanced manner; highly unbalanced partitions are inefficient and undesirable. Here, we introduce variations on model-parallel, which differ on their partitioning quality. Concretely, partitioning involves constructing a size-M^2 dependency graph, with weighted edges e_{ij} that measure the dependency between two parameter updates $Update_{(i)}$ and $Update_{(j)}$. This measure of dependency differs from algorithm to algorithm. For example, in Lasso regression, e_{ij} is the correlation between the i-th and j-th data dimensions. The total violation of a partitioning equals the sum of the edges' weights that cross between the P partitions, which we wish to minimize.

Ideal Model-Parallel: Theoretically, there exists an "ideal" load-balanced parallelization over P workers that gives the highest possible progress per iteration. This is indicated by an ideal (but not necessarily computable) schedule $S_p^{ideal}()$ that replaces the generic $S_p()$ in equation (3.5).

There are two points to note: (1) even this "ideal" model parallelization can still violate model dependencies and incur errors when compared to sequential execution because of residual cross-worker coupling, and (2) computing $S_p^{ideal}()$ is generally expensive because graph-partitioning is NP-hard. Ideal model parallel achieves the highest progress per iteration amongst load-balanced model-parallel programs, but may incur a large one-time, or even every-iteration, partitioning cost, which can greatly reduce iteration throughput.

Random Model-Parallel: At the other extreme is random model parallelization, in which a schedule $S_p^{rand}()$ simply chooses one parameter update at random for each worker p [12].

As the number of workers P increases, the expected number of violated dependencies will also increase, leading to poor progress per iteration (or even algorithm failures). However, there is practically no scheduling cost to iteration throughput.

Approximate Model-Parallel: As a middle ground between ideal and random model parallelization, we may *approximate* $S_p^{ideal}()$ via a cheap-to-compute schedule $S_p^{approx}()$. A number of strategies exist: one may partition small subsets of parameter updates at a time (instead of the M^2-size full dependency graph), apply approximate partitioning algorithms [75] such as METIS [41] (to avoid NP-hard partitioning costs), or use strategies that are unique to a particular machine learning program's structure.

3.2.2 Practical strategies for model partitioning and scheduling

Among three aforementioned variations of model parallel, we focus on the approximate model-parallel and present four practical scheduling strategies, static partitioning, dynamic partitioning, stale model-parallel, and prioritization.

Static Partitioning: A fixed, static schedule $S_p^{fix}()$ hard-codes the partitioning for every iteration beforehand. Progress per iteration varies depending on how well $S_p^{fix}()$ matches the machine learning program's dependencies. Like random model-parallel, this has little cost to iteration throughput.

Dynamic Partitioning: Dynamic partitioning $S_p^{dyn}()$ tries to select independent parameter updates by performing pair-wise dependency tests between a small number L of parameter updates (which can be chosen differently at different iterations, based on some priority policy as discussed later). The idea is to only do L^2 computational work per iteration, which is far less than M^2 (where M is the total number of parameter updates), based on a priority policy that selects the L parameters that matter most to the program's convergence. Dynamic partitioning can achieve high progress per iteration, similar to ideal model-parallel, but may suffer from poor iteration throughput on a distributed system: because only a small number of parameters are updated each iteration, the time spent computing $\Delta_p()$ at P workers may not be able to amortize network latencies and the cost of computing $S_p^{dyn}()$.

Stale Input to Updating and Scheduling: This is not a different type of model-parallel, but a complementary technique that can be applied to any model-parallel strategy. Stale input allows the next iteration(s) to start before the current one finishes, ensuring that computation is always fully utilized. However, this introduces *staleness* into the model-parallel execution:

$$A^{(t)} = F(A^{(t-1)}, \textstyle\sum_{p=1}^{P} \Delta_p(A^{(t-s)}, S_p(A^{(t-s)}))). \tag{3.8}$$

Note how the model parameters $A^{(t-s)}$ being used for $\Delta_p(), S_p()$ come from the iteration $(t-s)$, where s is the degree of staleness. Because machine learning algorithms are error-tolerant, they can still converge under stale model images (up to a practical limit) [36, 23]. Therefore, stale input sacrifices some progress per iteration to increase iteration throughput, making it a good way to raise the throughput of dynamic partitioning.

Prioritization: Like stale input, prioritization is complementary to model-parallel strategies. The idea is to modify $S_p()$ to prefer parameters that, when updated, will yield the most convergence progress [53], while avoiding parameters that are already converged [48]. This is effective because many ML algorithms exhibit uneven parameter convergence rate. Since computing a parameter's potential progress can be expensive, we may employ cheap-but-effective approximations or heuristics to estimate the potential progress (as shown later in Chapter 5). Prioritization can thus greatly improve progress per iteration, at a small cost to iteration throughput.

In Chapter 4, 5, we will present two practical SchMP scheduling implementation based on combinations of these strategies.

3.2.3 Programming scheduled model parallel

Model-parallelism accommodates a wide range of partitioning and prioritization strategies (i.e., the schedule $S_p()$), from simple random selection to complex, dependency-calculating functions that can be more expensive than the updates $\Delta_p()$. In existing ML program implementations, the schedule is often written as part of the update logic, ranging from simple for-loops that sweep over all parameters one at a time, to sophisticated systems such as GraphLab [54, 53], which "activates" a parameter whenever one of its neighboring parameters changes. We contrast this with **scheduled model parallel(SchMP)**, in which the schedule $S_p()$ computation is explicitly separated from the update $\Delta_p()$ computation. The rationale behind SchMP is that the schedule can be a distinct object for systematic investigation, separate from the updates, and that a model-parallel ML program can be improved by simply changing $S_p()$ without altering $\Delta_p()$.

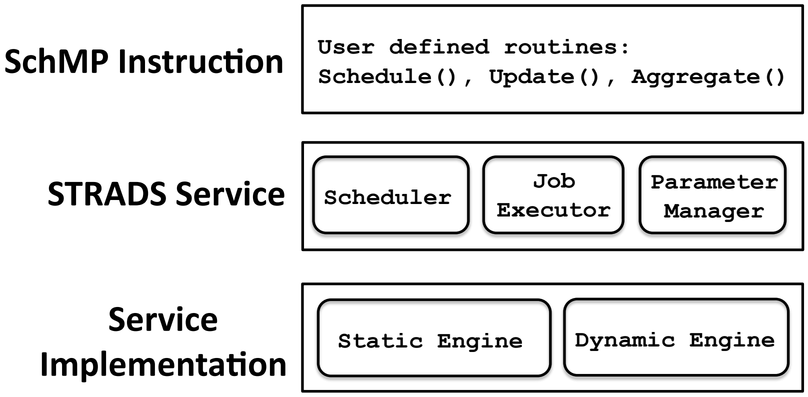


Figure 3.9: STRADS Architecture: To create a SchMP program, the user codes the SchMP Instructions, similar to MapReduce. The Services are system components that execute SchMP Instructions over a cluster. We provide two Implementations of the Services: a Static Engine and a Dynamic Engine, specialized for high performance on static-schedule and dynamic-schedule SchMP programs respectively. The user chooses which engine (s)he would like to use.

3.3 STRADS runtime design

In order to run SchMP machine learning applications efficiently in a cluster, we implement a new framework, STRADS [42] that conceptually consists of SchMP Instruction layer, common service layer and service implementation layer (SchMP engine layer). The service implementation layer currently supports two runtime execution engines, STRADS-Static and STRADS-Dynamic that exploit Stale Input to Updating and Scheduling and Prioritization. In this section, we cover common layer of STRADS runtime system, and the details of Static and Dynamic engines will be covered in Chapter 4, 5.

We present details of each layer:

- **SchMP Instruction Layer** is a set of programming primitives that are used to implement SchMP ML programs, including `Schedule()`, `Update()` and `Aggregate()`. To create a SchMP program, the user programs ML application in the from of these primitives.
- **Service Layer** is a set of system entities in a distributed system including Scheduler, Job Executor, ParamterManager (distributed key value store) and communication substrate for transferring control data and bulk user data. This layer executes SchMP instructions over cluster.
- **Service Implementation Layer(Engine Layer)** is to implement different SchMP

SchMP Function	Purpose	Available Inputs	Output
`schedule()`	Select parameters A to update	model A, data D	P parameter jobs $\{\mathbf{S}_p\}$
`update()`	Model parallel update equation	one parameter job $\mathbf{S}_p$, local data D_p, parameters A	one intermediate result R_p
`aggregate()`	Collect R_p and update model A	P intermediate results $\{R_p\}$, model A	new model state $A_p^{(t+1)}$

Table 3.1: SchMP Instructions: To create a SchMP program, the user implements these Instructions. The available inputs are optional — e.g. `schedule()` does not necessarily have to read A, D (such as in static partitioning).

program (scheduling) patterns using SchMP services, optimizing data flow topology, providing special optimizations for high performance. Current STRADS implementation provides two common service implementations, static and dynamic engines. The user choose which engine (s)he would like to use.

3.3.1 SchMP instruction layer

Table 3.1 shows the three SchMP Instructions, which are abstract functions that a user implements in order to create a SchMP program. All SchMP programs are iterative, where each iteration begins with `schedule()`, followed by parallel instances of `update()`, and ending with `aggregate()`; Algorithm 2 shows the general form of a SchMP program.

3.3.2 STRADS service layer

STRADS executes SchMP Instructions across a cluster via three Services: the **Scheduler**, **Job Executors**, and the **Parameter Manager**. The Scheduler is responsible for computing `schedule()` and passing the output jobs $\{\mathbf{S}_p\}$ on; most SchMP programs only require one machine to run the Scheduler, others may benefit from parallelization and pipelining over multiple machines. The Scheduler can keep local program state between iterations (e.g. counter variables or cached computations).

The P jobs $\{\mathbf{S}_p\}$ are distributed to P Job Executors, which start worker processes to run `update()`. On non-distributed file systems, the Job Executors must place worker processes exactly on machines with the data. Global access to model variables A is provided by the Parameter Manager, so the Job Executors do not need to consider model placement. Like the Scheduler, the Job Executors may keep local program state between

Algorithm 2 Generic SchMP ML program template

```
$A$: model parameters
$D_p$: local data stored at worker $p$
$P$: number of workers
Function schedule($A, D$):
    Generate $P$ parameter subsets $[\mathbf{S}_1, \ldots, \mathbf{S}_P]$
    Return $[\mathbf{S}_1, \ldots, \mathbf{S}_P]$
Function update($p, \mathbf{S}_p, D_p, A$):      // In parallel over $p = 1..P$
    For each parameter $a$ in $\mathbf{S}_p$:
        $R_p[a]$ = updateParam($a, D_p$)
    Return $R_p$
Function aggregate($[R_1, \ldots, R_P], A$):
    Combine intermediate results $[R_1, \ldots, R_P]$
    Apply intermediate results to $A$
```

iterations.

Once the worker processes finish `update()` and generate their intermediate results R_p, the aggregator process on scheduler (1) performs `aggregate()` on $\{R_p\}$, and (2) commit model updates and thus reach the next state $A_p^{(t+1)}$. Control is then passed back to the Scheduler for the next iteration $(t + 1)$. Finally, the Parameter Manager supports the Scheduler and Job Executors by providing global access to model parameters A.

3.3.3 STRADS service implementation layer (SchMP Engine)

In Chapter 4, 5, we propose two scheduling schemes, static-scheduling and dynamic-scheduling – based on aforementioned Static Partitioning and Dynamic Partitioning, Stale Input to Updating Scheduling, and Prioritization strategies – and implement two execution engines, Static and Dynamic, using STRADS service corresponding to these two scheduling schemes.

Chapter 4

Static-SchMP & STRADS-Static Engine

In this chapter, we explore a static scheduling scheme (Static-SchMP) for machine learning applications, in which a perfect schedule plan[1] can be made prior to runtime. First, we present our static scheduling algorithm with machine learning applications' property that allows our static scheduling scheme. Then, we summarize system challenges in executing Static-SchMP programs in a cluster and present STRADS-Static engine that addresses the challenges. In summary, this chapter:

- Explores a machine learning program property that allows our static scheduling scheme.
- Presents our static scheduling algorithm that makes a perfect scheduling plan for a machine learning algorithm that satisfies the property above.
- Identifies four system design challenges and present STRADS-Static engine that addresses the challenges.
- Quantifies the benefits of Static-SchMP and STRADS-Static engine using three different metrics. Our evaluation shows that Static-SchMP LDA topic modeling on the static engine increases training speed by six times compared to YahooLDA, which is a carefully designed distributed LDA on a fully asynchronous parameter server.

Because the scheduling plan is fixed prior to runtime, Static-SchMP schedule functions tend to be computationally light. However, our Static-SchMP requires machine learning

[1] A perfect schedule plan is a distributed execution plan that ensures serializability and achieves progress per iteration comparable to that of sequential execution — which is ideal in terms of progress per iteration.

algorithms to satisfy a property in shared parameter access pattern. Here, we present this property and depict a general static scheduling algorithm that can be applied to the machine learning algorithms that have this property — Static-SchMP compatible algorithm.

If a targeting machine learning algorithm does not have this property, the dynamic scheduling in Chapter 5 can serve them. Or, one might use a dynamic schedule on a Static-SchMP compatible algorithm and outperform the equivalent of static scheduling. The costs of this tactic are computational overhead of dynamic scheduling and the programming efforts required in order to implement a relatively more complicated scheduling algorithm than static scheduling. Because these dynamic and static scheduling schemes have different system needs, we separately provide two distinct but complete implementations of their scheduling engines (STRADS-Static and STRADS-Dynamic Engines). In this chapter, we focus on the Static-SchMP and STRADS-Static engine that achieves high system throughput of Static-SchMP machine learning applications in a cluster. The Dynamic-SchMP and its engine implementation will be covered in Chapter 5.

4.1 Static-SchMP

In this section, we present a property that our static scheduling scheme requires a targeted application to satisfy, and present our static scheduling algorithm with two examples.

4.1.1 Program property for static-scheduling

Static scheduling follows Static Partitioning strategy in Section 3.2.2, but not all of machine learning algorithms can be efficiently scheduled through static scheduling. Therefore, in our static scheduling, we limit our static scheduling to a group of machine learning applications that satisfies a property — the parameter update function[2] accesses k parameter entries from k different parameter sets.

To help machine learning programmers check whether their targeting machine learning applications are eligible for our static scheduling scheme, we provide a guidance as follows. First, we categorize machine learning algorithms into two types according to the signature of their update function: (1) **variadic algorithms**, where the update function reads/writes a variadic number of shared parameters and (2) **non-variadic algorithms**, where the update function reads/writes a fixed number k of shared parameters. We further categorize non-variadic machine learning algorithms into two cases according to the

[2] This is a scheduling unit.

Algorithm 3 static scheduling where k = 1

```
$M$: the number of worker nodes
$S[]$: an array of shared parameter shards where length = $M$
while Until Convergence do
    for ($i = 0; i < M; i$++){
        worker[0] = access permission to S[$(i + 0)\%M$]
        worker[1] = access permission to S[$(i + 1)\%M$]
        ...
        worker[$M - 2$] = access permission to S[$(i + M - 2)\%M$]
        worker[$M - 1$] = access permission to S[$(i + M - 1)\%M$]
    }
end while
```

Algorithm 4 static scheduling for $k > 1$

```
$M$: the number of worker nodes
$S_0[], S_1[], .., S_{k-1}[]$: shard arrays of k different sets of shared parameters
while Until Convergence do
    for($i_0 = 0; i_0 < M; i_0$++){
      for($i_1 = 0; i_1 < M; i_1$++){
        for($i_{k-1} = 0; i_{k-1} < M; i_{k-1}$++){
          worker[0] = access to $S_0[(i_0 + 0)\%M]$,$S_1[(i_1 + 0)\%M]$,..,$S_{k-1}[(i_{k-1} + 0)\%M]$
          worker[1] = access to $S_0[(i_0 + 1)\%M]$,$S_1[(i_1 + 1)\%M]$,..,$S_{k-1}[(i_{k-1} + 1)\%M]$
          ...
          worker[$M - 1$] = access to $S_0[(i_0 + M - 1)\%M]$,$S_1[(i_1 + M - 1)\%M]$,..,$S_{k-1}[(i_{k-1} + M - 1)\%M]$
        }
      }
    }
end while
```

dependency graph structure: (1) k-partite algorithm in which a single update operation accesses k different parameters from k different sets of shared parameters; and (2) non-k partite algorithm, in which there is no such constraint as k-partite algorithm has. Static-SchMP targets k-partite algorithms while variadic algorithms and non-variadic/non-k partite algorithms can be supported by dynamic scheduling.

4.1.2 STRADS static-scheduling

For simplicity of explanation, we present STRADS static scheduling algorithm in two forms depending on k value. Algorithm 3 is for k = 1 case, and Algorithm 4 is for $k > 1$.

Static Scheduling for machine learning Algorithms with k = 1: Algorithm 3 is the static scheduling algorithm for the case of k = 1, where each update accesses a single shared parameter and may access other data objects that are stored locally on a worker node. The shared parameters are partitioned into M disjoint shards, where M is the number of

Algorithm 5 Static-SchMP for LDA Topic Modeling

```
$U, V$: doc-topic table, word-topic table (model params)
$N, M$: number of docs, vocabulary size
$\{z\}_p, \{w\}_p$: topic indicators and token words stored at worker $p$
$c$: persistent counter in schedule() Function schedule():
  For $p = 1..P$:        // “word-rotation” schedule
    $x = (p - 1 + c)$ mod $P$
    $\mathbf{S}_p = (xM/P, (x+1)M/P)$        // $p$’s word range
  $c = c + 1$
  Return $[\mathbf{S}_1, \ldots, \mathbf{S}_P]$ Function update($p, \mathbf{S}_p, \{U\}_p, V, \{w\}_p, \{z\}_p$):
  [lower,upper] = $\mathbf{S}_p$        // Only touch $w_{ij}$ in range
  For each token $z_{ij}$ in $\{z\}_p$:
    If $w_{ij} \in$ range(lower,upper):
      old = $z_{ij}$
      new = SparseLDAsample($U_i, V, w_{ij}, z_{ij}$)
      Record old, new values of $z_{ij}$ in $R_p$
  Return $R_p$ Function aggregate($[R_1, \ldots, R_P], U, V$):
  Update $U, V$ with changes in $[R_1, \ldots, R_P]$
```

worker nodes. One iteration is divided into M subiterations. At each subiteration i where $i = 0, 1, .., M-1$, a worker node W_m obtains execlusive access to the $(i+m)\%M$-th shard. It then executes a subset of update operations that require access to the $(i+m)\%M$-th partition. After M subiterations, worker nodes complete all update operations once, which is equal to the workload of one iteration.

Static Scheduling for machine learning Algorithm with $k > 1$: In k-partite algorithms, shared parameters consist of k different independent sets $S_0, S_1, ..S_{k-1}$. An update operation accesses k shared parameters, each of which comes from k different sets. From the machine learning perspective, the sets of different parameters represent different types of objects. In Algorithm 4, each parameter set is partitioned into M disjoint shards, and one iteration is divided into M^k subiterations. Algorithm 4 runs k nested loops to perform scheduling. Every M subiteration, the loop statement at level $k-1$ (= the innermost loop) completes one full pass and increments the loop index at level $k-2$. After M^k subiterations, the algorithm complete one full pass of the loop at level-0 (=the outermost loop), which is equal to the amount of work of one iteration.

4.1.3 Static-scheduling example with LDA topic modeling

We apply Static-SchMP to LDA topic modeling algorithm in Section 3.1.4. For simplicity, we will not show the details of `update()` and `aggregate()`. Instead, we focus on how `schedule()` controls which tokens’ topic assignments z_{ij} are being updated by which workers. For LDA algorithm, see Section 3.1.4 and Figure 3.5(a). Algorithm 5 shows SchMP schedule for LDA. The algorithm assumes that input data (a document set) is randomly partitioned over workers, and document-topic table U is partitioned according to the document partitioning. A row id of document-topic table U corresponds to a document id in the document set so that partitioning the document-topic table to be aligned with the partitioning of the document set can be done trivially by using the same hash function of document set partitioning. Once U is partitioned to be aligned with the document set partitioning, shards of U are locally accessed during running updates, and update function accesses a single row of V table to process a token, which allows Static-SchMP with k = 1. Word-topic table V is partitioned into P shards, $V_0, V_1, .., V_{p-1}$ over P workers initially, and the scheduling algorithm rotates the word-topic table partitions along the workers so that only a single worker can access a shard V_p at a time. On arrival of a shard V_p, a worker updates topic assignment $z_{i,j}$ for word tokens $d_{i,j}$ that are associated with V_p. For running an iteration, the scheduling algorithm rotates V partitions P times so that every word $d_{i,j}$ in each worker is processed exactly once.

One might ask why `schedule()` is useful, because a common strategy is to have workers sweep over all their z_{ij} every iteration [2]. However, as we exhibited in Section 3.1.4, this common strategy causes concurrent access to the same rows in V and staleness problem — running updates with stale parameter values causes computational inaccuracy and makes less progress per iteration than Statich-SchMP application and sequential application.

4.2 STRADS-Static Engine

In this section, we summarize four system design challenges in running Static-SchMP applications in a cluster and present our solutions to address them.

4.2.1 System design challenges

In this section, we depict a common communication pattern found in Static-SchMP, which we can exploit to improve system throughput, and present three other system challenges.

Ring communication pattern: Static-SchMP in Algorithms 3 and 4 rotates shared parameter shards along a ring of worker nodes. For example, Algorithm 3 starts with M parameter shards, each of which is assigned to a worker. In each subiteration, it shifts a mapping of a worker and a parameter partition in one click in the clock-wise direction of the ring.

Synchronization challenge: Because static scheduling scheme shifts the mapping of worker nodes and parameter partitions by one click every subiteration, frequent synchronizations bottleneck iteration throughput. For instance, in the case of K = 1, one iteration goes through M synchronizations where M is the number of workers. In the general k-partite algorithm, the scheme goes through $M, M^2, ..., M^K$ synchronizations per one iteration. Such frequent synchronizations make the straggler problem and load balancing problem even worse. Therefore, reducing the number of synchronizations per iteration is critical to improving iteration throughput of Static-SchMP applications.

A worker node design challenge: Because synchronization overhead tends to be proportional to the number of worker nodes, it is desirable to make a worker node per a physical node that has multiple cores (usually 4 $\sim$ 64 cores), instead of making a worker node per a core. However, a worker node design with many cores is more likely to suffer from high lock contention and high cache miss ratio — which results in poor update throughput per core. We will address this problem by grouping update operations according to a shared model parameter and sequentially executing a group of update operations (= a job) on a dedicated thread.

Load balancing challenge: To be executed efficiently within a worker node, a group of update operations that touch a shared model parameter should be dedicated to a thread and executed sequentially. However, that grouping technology might cause a load balancing problem among threads within a worker node. In machine learning problems, workload distribution per model parameter is often uneven. For example, the word token distribution in LDA is highly skewed, meaning that few update groups associated with few frequent words takes much longer than others associated. Therefore, certain threads with particularly heavy jobs will cause load-balancing problem. We will address this problem using a heavy-job prioritization technique, which prioritizes heavy update groups when selecting an update group to execute or pass to a neighbor in the ring.

4.2.2 STRADS static-engine implementation

Figure 4.1 shows the overall architecture of STRADS-Static engine. The input training data is partitioned over worker nodes, and shared parameters are stored in Parameter Manager (a distributed KV store) and circulated along a ring overlay network.

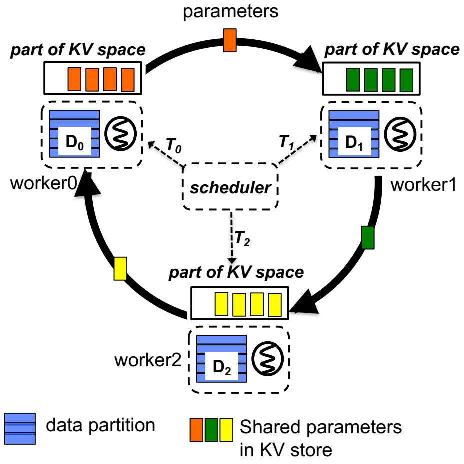


Figure 4.1: Overall Architecutre of Static Engine: training data is partitioned over worker nodes, and shared parameters are stored in Parameter Manager (a distributed key value store). The key range of store is partitioned over worker nodes. Shared parameters are circulated along the ring of worker nodes. The scheduler sends messages that trigger iteration and put synchronization barrier.

Ring overlay-network for addressing synchronization overhead

To improve iteration throughput, we exploit the ring-based data movement pattern of Static-SchMP. We implement a ring overlay network on worker nodes and let worker nodes rotate parameters along the ring network at fine granularity. When running the innermost loop, the static engine lets workers send completed parameters to the next machine immediately, instead of waiting for an entire partition to be completed. Therefore, the static engine performs one global synchronization when all the parameters at level $k-1$ complete one rotation (= the innermost loop completes one full pass). Therefore, the count of synchronizations at level $k-1$ is reduced from M to 1, where M is the number of worker nodes. For the example of Static-SchMP application with k = 1, the synchronization count per iteration is reduced from M to 1 where M is the number of worker nodes. In Static-SchMP applications with k=2, the synchronization count is reduced from M^2 to M.

Job Pool for multi-threading worker node

A worker node runs three types of threads as shown in Figure 4.2: a parameter manager thread that circulates shared parameter shards continuously along the ring of worker nodes; a job pool manager that coalesces associated updates into a job and manage job pool; and

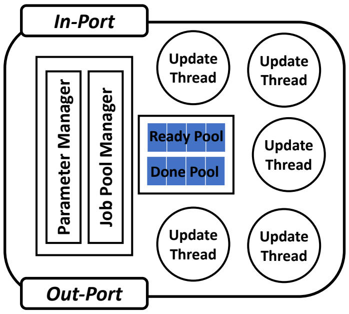


Figure 4.2: Worker node architecture of Static engine: runs the parameter manager thread, job pool manager that creates and dispatches jobs to update threads.

update thread that executes updates of a job sequentially. The job pool manager maintains two pools, a ready pool and a done pool. Upon receiving a shared parameter p_i from a ring neighbor, the job pool manager creates a job J_i with update operations that are associated with the parameter p_i, and puts J_i in the ready pool. At runtime, update threads poll and pull available jobs from the ready pool. On obtaining J_i, an update thread sequentially executes all update operation of J_i. Then, the completed job J_i is put into the done pool and finally p'_i is passed onto the next ring neighbor by the parameter manager.

Prioritization in job dispatching

Due to the skewed workload distribution of machine learning, a job for a popular object (i.e. tokens associated with a popular word in LDA or ratings associated with a popular product in SGDMF) could be far heavier than others, and thus take much longer time to process. If jobs in the ready and done pools are served in FIFO (First In, First Out), these heavy jobs will aggravate load-balancing problem, meaning that few threads with these jobs could bottleneck iterations. To mitigate this problem, the Job pool manager/Parameter Manager prioritizes heavy jobs when placing a new job on the ready pool and pulling done jobs from the done pool. For Read/Done pools with prioritization, we implement priority queues.

4.3 Evaluation

We compare SchMP programs implemented on STRADS against existing parallel execution schemes — either a well-known publicly-available implementation, or if unavailable,

ML app	Data set	Workload	Feature	Raw size
LDA	NYTimes	99.5M tokens	300K documents, 100K words 1K topics	0.5 GB
LDA	PubMed	737M tokens	8.2M documents, 141K words, 1K topics	4.5GB
LDA	ClueWeb	10B tokens	50M webpages, 2M words, 1K topics	80 GB
MF	Netflix	100M ratings	480K users, 17K movies (rank=40)	2.2 GB
MF	x256 Netflix	25B ratings	7.6M users, 272K movies (rank=40)	563 GB

Table 4.1: Data sets used in the evaluation.

our own implementation — as well as sequential execution. We intend to show that SchMP implementations executed by STRADS have significantly improved progress per iteration over other parallel execution schemes; in some cases, they come fairly close to "ideal" sequential execution. At the same time, the STRADS system can sustain high iteration throughput (i.e. model parameters and data points processed per second) that is competitive with existing systems. Together, the high progress per iteration and high iteration throughput lead to faster machine learning program completion times (i.e. fewer seconds to convergence).

4.3.1 Cluster setup and datasets

Unless otherwise stated, we used 100 nodes, each with 4 quad-core processors (16 physical cores) and 32GB memory. This configuration is similar to Amazon EC2 c4.4xlarge instances (16 physical cores, 30GB memory). The nodes are connected by 1Gbps Ethernet as well as a 20Gbps Infiniband IP over an IB interface. Most experiments were conducted via the 1Gbps Ethernet; those that were conducted over IB are noted. We use several real and synthetic datasets (see Table 4.1 for details).

4.3.2 Performance metrics:

We compare machine learning implementations using three metrics: (1) *objective function value* versus *total data samples operated upon*[3], abbreviated **OvD**; (2) *total data samples operated upon* versus *time (seconds*), abbreviated **DvT**; and (3) *objective function value* versus *time (seconds)*, referred to as **convergence time**. The goal is to achieve the best objective value in the least time (i.e. fast convergence).

[3] Machine learning algorithms operate upon the same data point many times. The total data samples operated upon exceeds N, the number of data samples.

Data set(size)	#machines	YahooLDA	SchMP-LDA
NYT(0.5GB)	25	38	43
NYT(0.5GB)	50	79	62
PubMed(4.5GB)	25	38	60
PubMed(4.5GB)	50	74	110
ClueWeb(80GB)	25	39.7	58.3
ClueWeb(80GB)	50	78	114
ClueWeb(80GB)	100	151	204

Table 4.2: Static SchMP: DvT for topic modeling (million tokens processed per second).

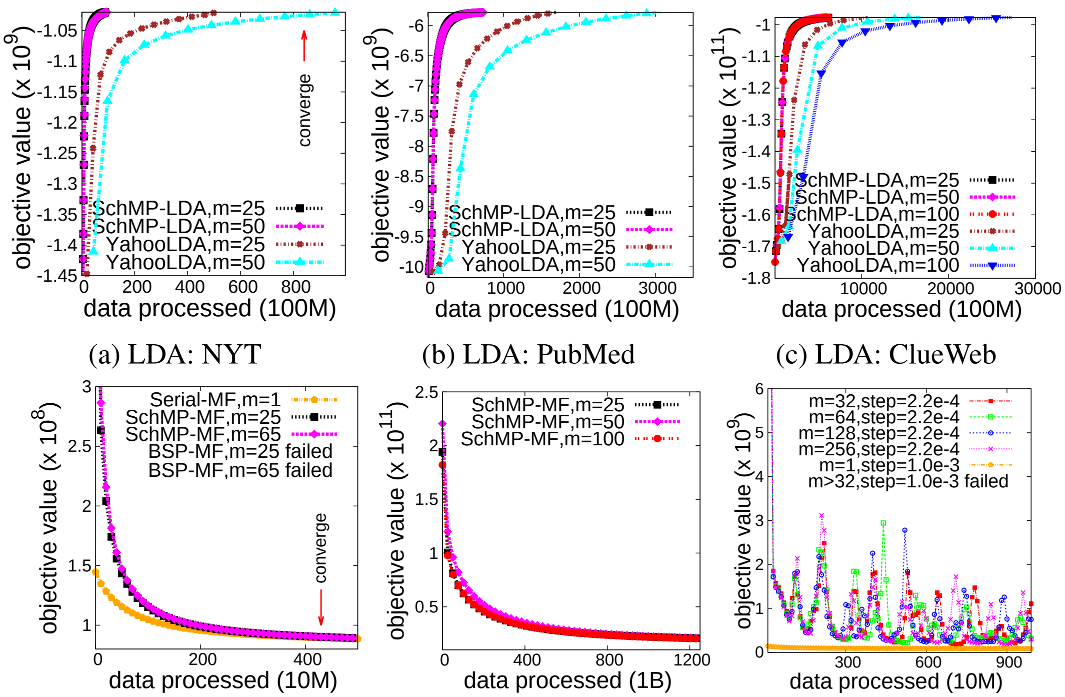
OvD is a uniform way to measure machine learning progress per iteration across different machine learning implementations, as long as they use identical parameter update equations. This is always the case, unless otherwise stated. Similarly, DvT measures machine learning iteration throughput across comparable implementations. Note that high OvD and DvT imply good (i.e. small) machine learning convergence time. Measuring OvD or DvT alone (as is sometimes done) is *insufficient* to show that an algorithm converges quickly.

4.3.3 Machine learning programs and baselines:

We evaluate the performance of LDA (a.k.a. topic model) and MF (a.k.a collaborative filtering). STRADS uses Algorithm 5 (**SchMP-LDA**) for LDA, and a scheduled version of the Stochastic Gradient Descent (SGD) algorithm for MF (**SchMP-MF**). For baselines, we used **YahooLDA**, and **BSP-MF** – our own implementation of the classic BSP SGD for MF. Both are data-parallel algorithms, meaning that they do not use SchMP schemes. These baselines were chosen to analyze how SchMP affects OvD, DvT, and convergence time. Later we will compare convergence time benchmarks against the GraphLab system, which does use model parallelism.

To ensure a fair comparison, YahooLDA was modified to (1) dump the model state at regular intervals for later objective (log-likelihood) computation[4] and (2) keep all local program state in memory, rather than streaming it off a disk. All LDA experiments were performed on the 20Gbps Infiniband network, such that bandwidth was not a bottleneck for the parameter server used by YahooLDA. Note that in LDA OvD and DvT measurements, we consider each word token as one data sample.

[4] With overhead less than 1% of total running time.



(a) LDA: NYT (b) LDA: PubMed (c) LDA: ClueWeb

(d) MF: Netflix (e) MF: x256 Net. (f) BSP-MF: Net.

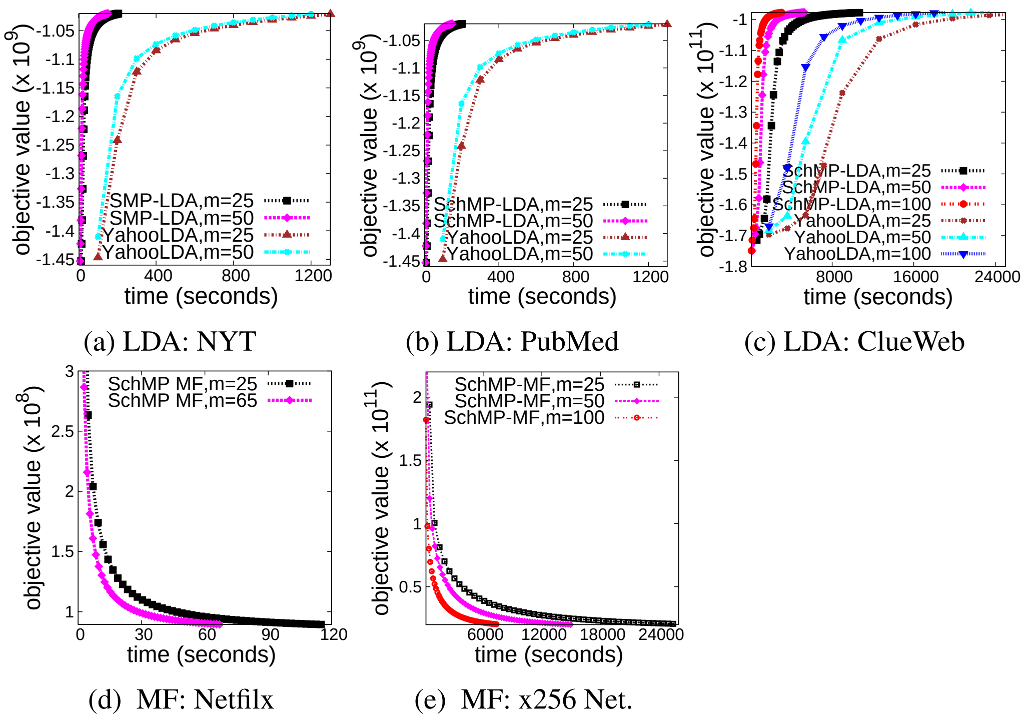
Figure 4.3: Static SchMP: OvD. (a-b) SchMP-LDA vs YahooLDA on two data sets; (c-d) SchMP-MF vs BSP-MF on two data sets; (e) parallel BSP-MF is unstable if we use an ideal sequential step size; m denotes number of machines.

4.3.4 Performance evaluations

Static SchMP has high OvD: For LDA, YahooLDA's OvD decreases substantially from 25 to 100 machines, whereas SchMP-LDA maintains the same OvD (Figures 4.3a, 4.3c). For MF, Figure 4.3f shows that BSP-MF is sensitive to step size[5]; if BSP-MF employs the ideal step size determined for serial execution, it does not properly converge on ≥ 32 machines. In contrast, SchMP-MF can safely use the ideal serial step size (Figures 4.3d,4.3e) and approaches the same OvD as serial execution within 20 iterations.

STRADS Static Engine has high DvT: For LDA, Table 4.2 shows that SchMP-LDA enjoys higher DvT than YahooLDA. We speculate that YahooLDA's lower DvT is primarily due to lock contention on shared data structures between application and parameter server

[5] A required tuning parameter for SGDMF implementations; higher step sizes lead to faster convergence, but step sizes that are too large can cause algorithm divergence/failure.



(a) LDA: NYT

(b) LDA: PubMed

(c) LDA: ClueWeb

(d) MF: Netfilx

(e) MF: x256 Net.

Figure 4.4: Static SchMP: convergence times. (a-b) SchMP-LDA vs YahooLDA; (c-d) SchMP-MF with varying number of machines m.

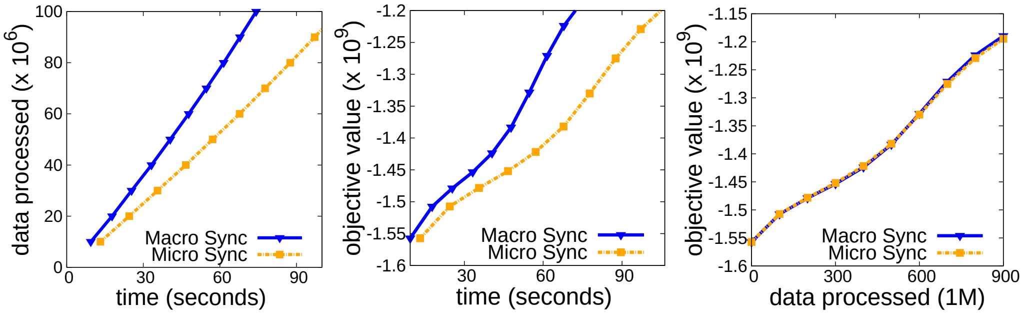
threads (which the STRADS Static Engine tries to avoid).

Static SchMP on STRADS has low convergence times: Thanks to high OvD and DvT, SchMP-LDA's convergence times are not only lower than YahooLDA, but also scale better with increasing machine count (Figures 4.4b, 4.4c). SchMP-MF also exhibits good scalability (Figure 4.4d, 4.4e).

4.3.5 Evaluations of static engine optimizations

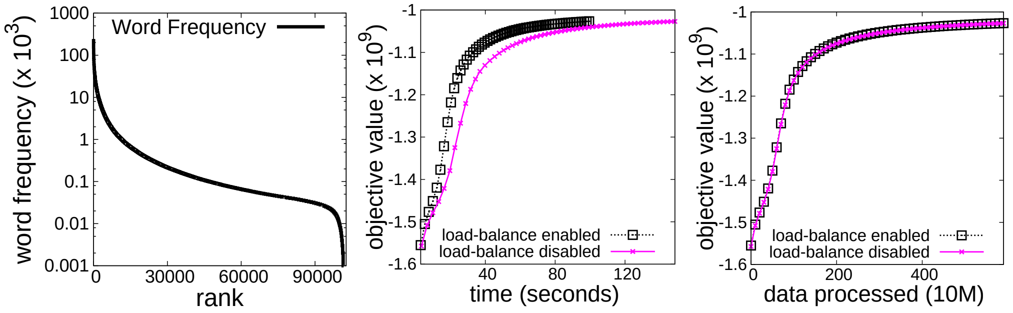
The STRADS Static Engine achieves high DvT (i.e iteration throughput) via two system optimizations: (1) reducing synchronization costs via the ring topology; (2) using a job pool to perform load balancing across **workers**.

Reducing synchronization costs: Static SchMP programs (including SchMP-LDA and SchMP-MF) do not require all parameters to be synchronized across all machines, and this motivates the use of a ring topology. For example, consider SchMP-LDA Algorithm 5: the



(a) DvT (b) Time (c) OvD

Figure 4.5: Static Engine: synchronization cost optimization. (a) macro synchronization improves DvT by 1.3 times; (b) it improves convergence speed by 1.3 times; (c) This synchronization strategy does not hurt OvD.



(a) NYTimes (b) Time (c) OvD

Figure 4.6: Static Engine: Job pool load balancing. (a) Biased word frequency distribution in NYTimes data set; (b) by dispatching the 300 heaviest words first, convergence speed improves by 30 percent to reach objective value -1.02e+9; (c) this dispatching strategy does not hurt OvD.

word-rotation `schedule()` directly suggests that **a worker** can pass parameters to their ring neighbor, rather than broadcasting to all machines; this applies to SchMP-MF as well.

STRADS's Static Engine implements this parameter-passing strategy via a ring topology, and only performs a global synchronization barrier after all parameters have completed one rotation (i.e. P iterations) — we refer to this as "Macro Synchronization". This has two effects: (1) network traffic becomes less bursty, and (2) communication is effectively overlapped with computation; as a result, DvT is improved by 30% compared to a naive implementation that invokes a synchronization barrier every iteration ("Micro Synchronization", Figure 4.5a). This strategy does not negatively affect OvD (Figure 4.5c), and hence time to convergence improves by about 30% (Figure 4.5b).

Job pool load balancing: As mentioned in sec:static-system-challenge, uneven workloads are common in Static SchMP programs: Figure 4.6a shows that the word distribution in LDA is highly skewed, meaning that some SchMP-LDA `update()` jobs will be much longer than others. Hence, STRADS dispatches the heaviest jobs first to the update threads. This improves convergence times by 30% on SchMP-LDA (Figure 4.6b), without affecting OvD.

4.3.6 Comparison against other frameworks

GraphLab: We compare SchMP-MF with GraphLab's SGD MF implementation, on a different set of 8 machines — each with 64 cores, 128GB memory. On Netflix , GL-SGDMF converged to objective value 1.8e+8 in 300 seconds, and SchMP-MF converged to 9.0e+7 in 302 seconds (i.e. better objective value in the same time). In terms of DvT, SchMP-MF touches 11.3m data samples per second, while GL-MF touches 4.5m data samples per second.

Comparison against single-core LDA: We compare SchMP-LDA with a single-core LDA implementation (Single-LDA)[6] on PubMed. Single-LDA converges in 24.6 hours while SchMP-LDA takes 17.5 minutes and 11.5 minutes on 25 machines (400 cores) and 50 machines (800 cores) respectively. Both Single-LDA and SchMP-LDA show similar OvD results. In DvT, Single-LDA processes 830K tokens per second while SchMP-LDA processes 70M tokens on 25 machines (175K tokens per core), and 107M tokens on 50 machines (133K tokens per core). The locking contention on a shared data structure within a machine accounts for the reduced per-core efficiency of SchMP versus Single-LDA. Even so, the distributed approach of SchMP achieves substantial speed-up gains (84 times on 25 machines, 128 times on 50 machines) over Single-LDA. We leave further machine-level optimizations, such as relaxed consistency on shared data structures within a machine, as future work.

Bösen: We compare SchMP-LDA against an implementation of LDA on a recent parameter server, Bösen[88], which prioritizes model parameter communication across the network, based on each parameter's contribution to algorithm convergence. Thus, Bösen improves convergence rate (OvD) over YahooLDA while achieving similar token processing throughput (DvT). On the NYT data with 16 machines[7], SchMP-LDA and Bösen are 7 and 3 times faster, respectively, than YahooLDA, and SchMP-LDA is about 2.3 times

[6] For fair comparison, Single-LDA implements the same sampling algorithm and the same data structure of SchMP-LDA, and is lock-free. We use C++11 STL library for implementing the sampling algorithm routine from scratch without third-party library.

[7] For fair comparison, we set the stopping log-likelihood value to -1.0248e+09 for all experiments: Bösen, YahooLDA, SchMP-LDA with 16 machines.

faster than Bösen. The SchMP-LDA improvement comes from the static model-parallel `schedule()` in Algorithm 5 (that avoids violating model dependencies in LDA), which the Bösen data-parallel LDA implementation does not have.

Chapter 5

Dynamic-SchMP & STRADS-Dynamic Engine

In this chapter, we explore a dynamic scheduling scheme (Dynamic-SchMP). First, we present dynamic scheduling algorithm that improves the statistical progress of machine learning algorithms by taking dependency structure and uneven convergence of model parameters into account. Then, we discuss system design challenges in running a Dynamic-SchMP program in a cluster and present STRADS Dynamic-Engine to address them. This chapter:

- Presents a generic two-phase scheduling algorithm that considers model dependency and dynamically changing convergence status of model parameters.
- Identifies two system challenges and presents STRADS Dynamic-Engine implementation that addresses the challenges.
- Quantifies the benefits of dynamic scheduling (Dynamic-SchMP) and STRADS Dynamic-Engine using three different metrics. Our evaluation shows that Lasso (l1-regularized regression) and logistic regression applications with Dynamic-SchMP on STRADS Dynamic-Engine improve training speed by an order of magnitude.

5.1 Dynamic-SchMP

This section introduces the dynamic scheduling scheme (Dynamic-SchMP) that considers a machine learning application's dynamically changing states (i.e. convergence dis-

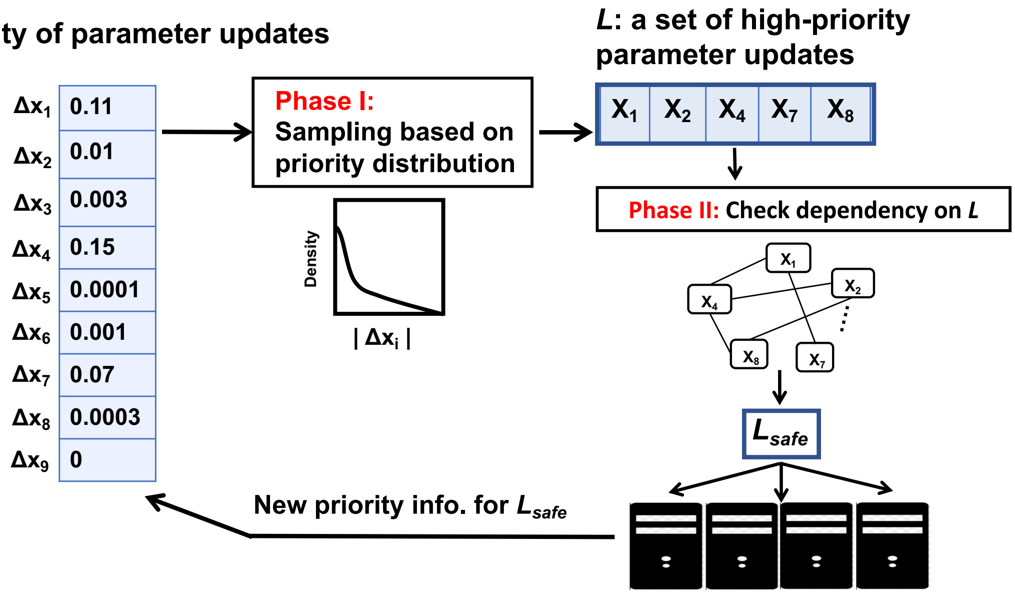

Priority of parameter updates
Δx1 0.11
Δx2 0.01
Δx3 0.003
Δx4 0.15
Δx5 0.0001
Δx6 0.001
Δx7 0.07
Δx8 0.0003
Δx9 0
Phase I:
Sampling based on priority distribution
Density
| Δxi |
L: a set of high-priority parameter updates
X1 X2 X4 X7 X8
Phase II: Check dependency on L
X1
X2
X4
X8
X7
Lsafe
New priority info. for Lsafe

Figure 5.1: Workflow of Two-Phase Scheduling in Dynamic-SchMP: Dynamic scheduler keeps track of priority information of individual model parameter updates. Priority of a model parameter update represent convergence distance of parameter values assoricated with the parameter update, and we approximate convergence distance by sum of delta of parameter values. Phase-I selects a set of model parameter update L based on the priority distribution. Phase-II builds model dependency graph for model parameters in L and checks on dependency structure. If a pair of parameters are found to have too strong dependency (larger than a threshold), one of them are set aside for next round. After dependency check, L is reduced to L_{safe} that degree of dependency on all possible pairs is less than a threahold.

tance of model parameters and runtime changes on model dependency structure), which Static-SchMP in Chapter 4 cannot capture, and generates scheduling plans at run time with affordable computation cost. To achieve such a scheduler, we combines two SchMP strategies in Sec 3.2.1, Dynamic Partitioning and Prioritization, and present a two-phase dynamic scheduling algorithm (Dynamic-SchMP). Note that dynamic scheduling is a general scheduling scheme that can support machine learning algorithms with variadic update function as well as Static-SchMP compatible algorithms with non-variadic update function we discussed in Chapter 4

5.1.1 Dynamic-SchMP

Dynamic-SchMP scheduler consists of two phases. Figure 5.1 depicts its workflow. The scheduler keeps track of priority information of individual parameter updates. Priority of a parameter update represents sum of convergence distances of model parameters associated with the parameter update in statistical parlance. The convergence distance of a model parameter at current time t is the difference between its value at current t and and its expected converged value at $t + n$. Because accurate estimation of convergence distance could be expensive or even impossible, in our study, we approximate it by measuring the delta of a model parameter per an update. The logic behind this is that the delta of model parameter value is getting smaller when the value of β_i is being closer to its converged value.

In Phase-I, the scheduler samples a subset of model parameter updates L based on updates' priorities where $|L| << $ the total parameter update count C. In Phase-II, the scheduler builds a dependency graph $\mathcal{G}_{\mathcal{L}}$ for L, where $e_{i,j}$ in $\mathcal{G}_{\mathcal{L}}$ represents degree of dependency[1] between two parameter updates i, j in L, and eliminates a edge that has larger dependency value than a user-defined threshold, by removing one of vertices of such edge. During this filtering process, the scheduler reserves victim updates for next round. Then, L_{safe}, in which dependency of any pairs of update in L_{safe} is less than the user defined threshold, is dispatched over a cluster for parallel execution. At the completion of execution of L_{safe}, the scheduler updates priorities of the parameter updates in L_{safe} based on the delta on parameter values that are associated with the parameter updates in L_{safe}.

5.1.2 Dynamic-SchMP example with Lasso

In this section, we apply Dynamic-SchMP to coordinate descent based lasso algorithm. Coordinate descent optimization method is the most common method for implementing Lasso (and a large number of its variations) but is known to be hard to parallelize using data-parallel approach unlike gradient descent method because coordinate descent method in many applications is often very sensitive to numerical inaccuracy and likely to make slow progress per iteration or fail to converge when being parallelized in data-parallel approach. We parallelize coordinate descent Lasso successfully using Dynamic-SchMP.

Lasso, or the ℓ_1-regularized least-squares regression, is used to identify a small set of important features from high-dimensional data. It is an optimization problem:

$$\min_\beta \quad \frac{1}{2}\sum_{i=1}^{n}\left(\mathbf{y}^i - \mathbf{x}^i\beta\right)^2 + \lambda\|\beta\|_1 \tag{5.1}$$

where $\|\beta\|_1 = \sum_{a=1}^{d}|\beta_a|$ is a sparsity-inducing ℓ_1-regularizer, and λ is a tuning parameter that controls the sparsity level of β. $\mathbf{X}$ is an N-by-M design matrix ($\mathbf{x}^i$ represents the i-th

[1] Defining a function that measure degree of dependency between two parameter updates is application specific but can be done easily by looking at machine learning algorithm.

Algorithm 6 SchMP Dynamic, Prioritized Lasso

```
$\mathbf{X}, \mathbf{y}$: input data
$\{\mathbf{X}\}^p, \{\mathbf{y}\}^p$: rows/samples of $\mathbf{X}, \mathbf{y}$ stored at worker $p$
$\beta$: model parameters (regression coefficients)
$\lambda$: $\ell_1$ regularization penalty
$\tau$: $\mathcal{G}$ edges whose weight is below $\tau$ are ignored
Function schedule($\beta, \mathbf{X}$):
    Pick $L > P$ params in $\beta$ with probability $\propto (\Delta\beta_a)^2$
    Build dependency graph $\mathcal{G}$ over $L$ chosen params:
        edge weight of $(\beta_a, \beta_b)$ = correlation($\mathbf{x}^a, \mathbf{x}^b$)
    $[\beta_{\mathcal{G}_1}, \ldots, \beta_{\mathcal{G}_K}]$ = findIndepNodeSet($\mathcal{G}, \tau$)
    For $p = 1..P$:
        $\mathbf{S}_p = [\beta_{\mathcal{G}_1}, \ldots, \beta_{\mathcal{G}_K}]$
    Return $[\mathbf{S}_1, \ldots, \mathbf{S}_P]$
Function update($p, \mathbf{S}_p, \{\mathbf{X}\}^p, \{\mathbf{y}\}^p, \beta$):
    For each param $\beta_a$ in $\mathbf{S}_p$, each row $i$ in $\{\mathbf{X}\}^p$:
        $R_p[a]$ += $x_a^i y^i - \sum_{b \neq a} x_a^i x_b^i \beta_b$
    Return $R_p$
Function aggregate($[R_1, \ldots, R_P], \mathbf{S}_1, \beta$):
    For each parameter $\beta_a$ in $\mathbf{S}_1$:
        temp = $\sum_{p=1}^{P} R_p[a]$
        $\beta_a = \mathcal{S}$(temp, $\lambda$)
```

row, $\mathbf{x}_a$ represents the a-th column), $\mathbf{y}$ is an N-by-1 observation vector, and β is the M-by-1 coefficient vector (the model parameters). The Coordinate Descent (CD) algorithm is used to solve Eq. (5.1), and thus learn β from the inputs $\mathbf{X}, \mathbf{y}$; the CD update rule for β_a is:

$$\beta_a^{(t)} \leftarrow \mathcal{S}(\mathbf{x}_a^\top \mathbf{y} - \textstyle\sum_{b \neq a} \mathbf{x}_a^\top \mathbf{x}_b \beta_b^{(t-1)}, \lambda), \tag{5.2}$$

where $\mathcal{S}(\cdot, \lambda)$ is a soft-thresholding operator [29].

Algorithm 6 shows SchMP Lasso that is scheduled the two-phase Dynamic-SchMP in Figure 5.1. In SchMP Lasso, a worker node stores a subset of design matrix $\mathbf{X}$ (which is common practice in parallel ML). However, the Lasso update Eq. (5.2) uses a feature/column-wise access pattern. Therefore, every worker $p = 1..P$ operates on the same scheduled set of L parameters, but using their respective data partitions $\{\mathbf{X}\}^p, \{\mathbf{y}\}^p$. Note that `update()` and `aggregate()` are a straightforward implementation of Eq. (5.2).

`schedule()` picks (i.e. prioritizes) L parameters[2] in β with probability proportional to their squared difference from the latest update (their "delta"). Parameters with larger

[2] In Lasso, an update operation corresponds to a model parameter so that we let a parameter β_i denote the parameter update on β_i.

delta are more likely to be non-converged. Next, it builds a dependency graph over these L parameters, with edge weights equal to the correlation[3] between data columns $\mathbf{x}^a, \mathbf{x}^b$. Finally, it eliminates all edges in $\mathcal{G}$ above a threshold $\tau > 0$ by removing a node per an eliminated edge. After the filtering process, all remaining nodes $\beta_{\mathcal{G}_k}$ do not have edges above a threshold and are thus safe to update in parallel.

Why is such a sophisticated `schedule()` necessary? See parallel Lasso with random parameter selection in Section 3.1. Figure 3.2 shows its progress, on the Alzheimer's Disease (AD) data [95]. The total compute to reach a fixed objective value goes up with more concurrent updates — i.e. progress per unit computation is decreasing, and the algorithm has poor scalability. Another reason is uneven parameter convergence. Figure 3.3 shows how many iterations different parameters took to converge on the AD dataset; $> 85\%$ of parameters converged in < 5 iterations, suggesting that the prioritization in Algorithm 6 should be very effective.

Default `schedule()` **functions:** The squared delta-based parameter prioritization and dynamic dependency checking in SchMP Lasso's `schedule()` (Algorithm 6) generalize to other regression problems — for example, we also implement sparse logistic regression using the same `schedule()`. STRADS allows ML programmers to re-use Algorithm 6's `schedule()` via a library function `scheduleDynRegr()`.

5.2 STRADS-Dynamic Engine

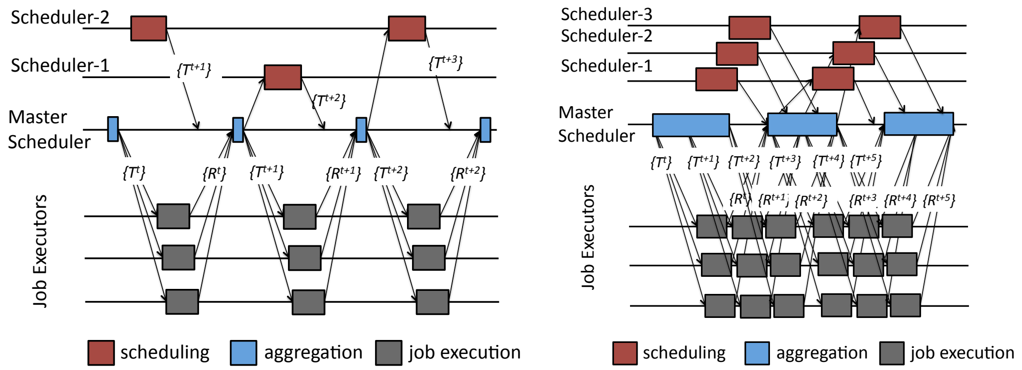
To achieve high update throughput of Dynamic-SchMP machine learning programs, we identify two system design challenges and present STRADS-Dynamic engine with three optimization solutions that trade "progress per update" from scheduling for update throughput. The logic behind this trade is to maximize progress per unit time by balancing update throughput and progress per update.

5.2.1 System design challenges

This section presents two major system design challenges in running Dynamic-SchMP applications in a cluster.

Scheduling Throughput: Update throughput of Dynamic-SchMP application is $min(S_{throughput}, U_{throughput})$ where $S_{throughput}$ is scheduling throughput by dynamic scheduler, and $U_{throughput}$ is iteration throughput by JobExecutors. In some machine learning problems, iteration might be

[3] On large data, it suffices to estimate the correlation with a data subsample.



(a) Non-pipelined

(b) Pipelined

Figure 5.2: Dynamic Engine pipelining: (a) Non-pipelined execution: network latency dominates; (b) Pipelining overlaps networking and computation.

short, hence parameter update throughput by JobExecutors is often larger than scheduling throughput of a single scheduler instance, in which JobExecutors have idle time waiting for schedule. Even worse, the parameter update throughput could be proportional to the number of JobExecutors, the scheduler design should be able to scale up with $U_{throughput}$.

Communication Latency: To dispatch scheduling plans and monitor dynamically changing priorities of parameter updates, Dynamic-SchMP incurs communication overheads between the scheduler and JobExecutors. In some machine learning programs, the latency of parameter update might be insufficient to amortize this communication latency, which bottlenecks update iteration throughput in the end.

5.2.2 STRADS Dynamic-Engine implementation

To address aforementioned challenges, we present distributed scheduler, in which scheduling throughput is scalable with the number of scheduler instances, and pipelining optimization, which overlaps newtork communication and computations of scheduling and parameter update.

Distributed Scheduler: To achieve scalable scheduling throughput, we implement a distributed scheduler that runs scheduling over multiple scheduler instances as shown in Figure 5.2a, 5.2b. In a distributed scheduler, model parameter updates are partitioned into S disjoint subsets where S is the number of scheduler instances and a scheduler instance runs independently on a subset of model parameter updates. Through this distributed design, Dynamic-SchMP scheduler with S instances easily increases scheduling through-

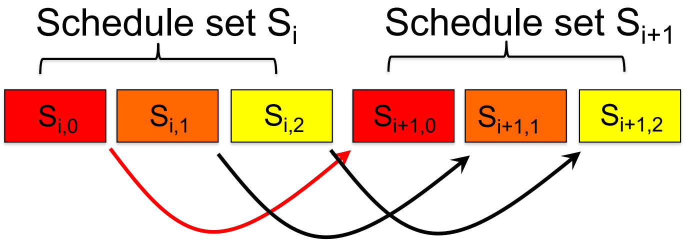


Figure 5.3: Reordering updates to mitigate side effect of pipelining: The dynamic engine reorders a schedule plan S in ascending order on updates' priorities and split S into three sub plans $S_{i,0}, S_{i,1}, S_{i,2}$. Red subplan is a group of highest priority updates, yello subplan is the opposite, and curves represent the availability of computation results at souce side for the computation at the destination side. When pipeline depth s is smaller than 3, the split and reordering optimization ensures the results of the red subplan $S_{i,0}$ is always available before starting the next red subplan $S_{i+1,0}$.

put by S times. In order to manage dispatch ordering and the feedback of updated parameter information, the dynamic engine has a coordinator node that pulls scheduling sets from S scheduler instances in a round robin manner and is responsible for running `aggregate()` function and pipelining tasks of schedule generation and update execution.

Pipelining: To address the aforementioned network communication latency challenge, Dynamic-Engine implements pipelining that overlaps network communication and computations of parameter update and scheduling; the dynamic engine will start additional iterations before waiting for completion of current iteration. The pipeline depth s (the number of in-flight iterations) can be set by the user. At iteration t, the dynamic engine starts the iterations $t+1, t+2, .., t+s$ before the current iteration t is completed. Figure 5.2a, 5.2b compare non-pipelined execution and pipelined execution. Although pipelining improves iteration throughput and overall converence speed, it may lower progress per iteration (update) due to (1) iteration t will not see the results from iteration $t-1$ to iteration $t-s$, where s is the pipeline depth; and (2) there might be dependencies between pipelined iterations because scheduling plans for iterations $t, t-1, .., t-s$ came from s different subsets of update operations that are generated by S independent scheduler instances. This does not lead to machine learning program failure because ML algorithms can tolerate some error and still converge — albeit more slowly. Pipelining is basically execution with stale parameters, $A^{(t)} = F(A^{(t-s)}, \{\Delta_p(A^{(t-s)}, S_p(A^{(t-s)}))\}_{p=1}^{P})$ where s is the pipeline depth.

Reordering updates: In this section, we present one more optimization technique that minimizes negative impacts of pipelining — certain loss on progress per iteration — by reordering parameter updates. The loss of statistical progress from pipelining is due to

ML app	Name	Workload	Feature	Input size (Disk)
Lasso	AlzheimerDisease (AD)	235M nonzero	463 sample, 0.5M feature	6.4 GB
Lasso	LassoSynthetic	2B nonzero	50K sample, 100M feature	45 GB
Logistic	LogisticSynthetic	1B nonzero	5K sample, 10M feature	29 GB

Table 5.1: Experiment data sets

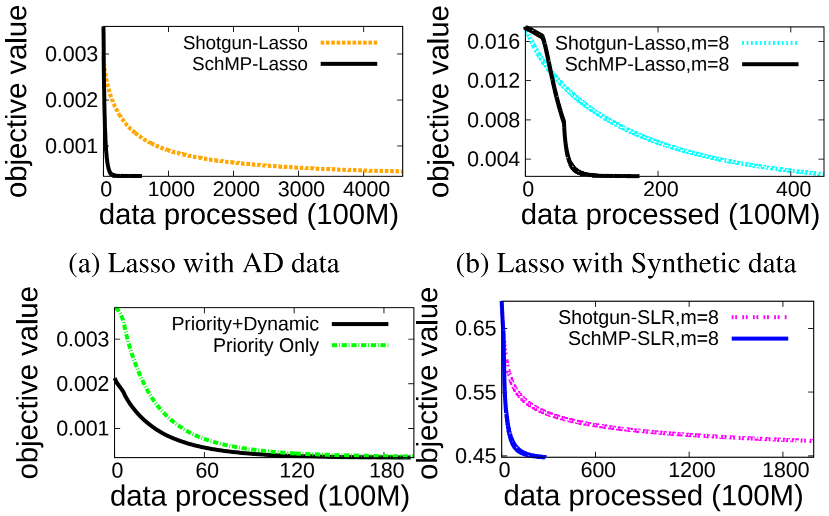
two factors:(1) the strength of dependencies among in-flight updates in the pipeline and (2) the magnitude of parameter changes by in-flight update operation in the pipeline. Here, we try to reduce the second factor, which is staleness associated with parameters that are modified by in-flight updates. First, for schedule set S_i with k update operations, the coordinator — the master scheduler in charge of pulling scheduling plans from scheduler instances and manages pipelining — reorders the update operations of S_i in ascending order of priority and splits S_i into n subsets, $S_{i,0}, S_{i,1}, .., S_{i,n-1}$. Then, the coordinator runs pipelining with subsets. This reordered subset pipelining could improve progress per iteration if the pipeline depth s is set to be smaller than n. Figure 5.3 illustrate an example where n is set to 3, and pipeline depth s is set to 2. When $S_{i+1,0}$, which is the most important updates of S_{i+1}, is being executed, the results of $S_{i,0}$, which is the most important update of S_i, is available without staleness. In other words, task reordering ensures that the most important update operations in the following schedule set $S_{i+1,0}$ can always see the results of $S_{i,0}$ when the pipeline depth s is less than n. Therefore, nemeric inaccuracy from pipelining can be mitigated.

5.3 Evaluation

Cluster setup and datasets: Unless otherwise stated, we used 8 nodes, each with 16 physical cores and 128GB memory. The nodes are connected by 1Gbps Ethernet. We use one real dataset and two synthetic datasets — see Table 5.1 for details.

Performance metrics: We compare ML implementations using three metrics: (1) *objective function value* versus *total data samples operated upon*, abbreviated **OvD**; (2) *total data samples operated upon* versus *time (seconds)*, abbreviated **DvT**; and (3) *objective function value* versus *time (seconds)*, referred to as **convergence time**. The goal is to achieve the best objective value in the least time — i.e. fast convergence.

Machine learning programs and baselines: We evaluate ℓ_1-regularized linear regression (Lasso) and ℓ_1-regularized Logistic regression (sparse LR, or SLR). STRADS uses Algorithm 6 (SchMP-Lasso) for the former, and we solve the latter using a minor modification to SchMP-Lasso[4] (called SchMP-SLR). To the best of our knowledge, there are no open-source distributed Lasso/SLR baselines that use coordinate descent, so we implement the Shotgun Lasso/SLR algorithm [12] (Shotgun-Lasso, Shotgun-SLR), which uses random model-parallel scheduling[5]



(a) Lasso with AD data

(b) Lasso with Synthetic data

(c) AD data

(d) SLR with Synthetic data

Figure 5.4: Dynamic SchMP: OvD. (a) SchMP-Lasso vs Shotgun-Lasso [12] on one machine (64 cores); (b) SchMP-Lasso vs Shotgun-Lasso on 8 machines; (c) SchMP-Lasso with & w/o dynamic partitioning on 4 machines; (d) SchMP-SLR vs Shotgun-SLR on 8 machines. m denotes number of machines.

5.3.1 Performance evaluations

Dynamic SchMP has high OvD: Dynamic SchMP achieves high OvD in both single-machine (Figure 5.4a) and distributed 8-machine (Figure 5.4b) configurations. Here, we compare SchMP-Lasso against random model-parallel Lasso (Shotgun-Lasso) [12]. In either case, Dynamic SchMP decreases the data samples required for convergence by an or-

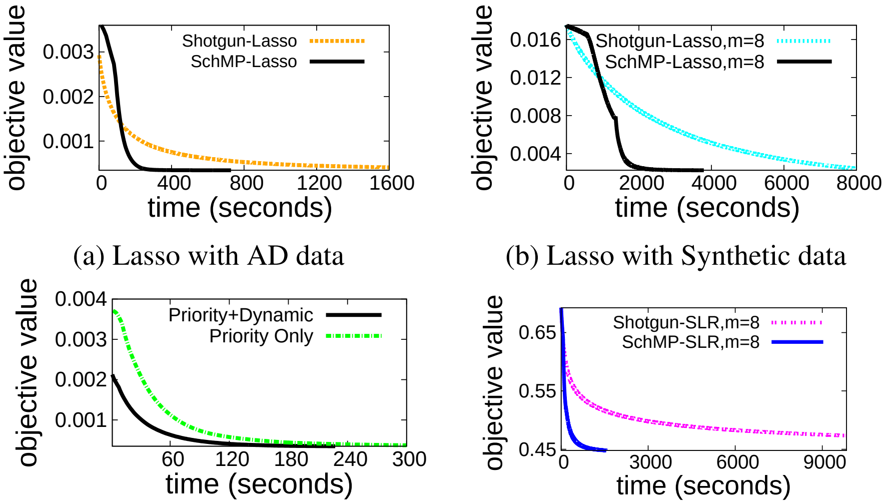
[4] Lasso and SLR are solved via the coordinate descent algorithm, hence SchMP-Lasso and SchMP-SLR only differ slightly in their update equations. We use coordinate descent rather than gradient descent because it has no step size tuning and more stable convergence [74, 72].

[5] Using coordinate descent baselines is essential to properly evaluate the DvT and OvD impact of SchMP-Lasso/SLR; other algorithms like stochastic gradient descent are only comparable in terms of convergence time.

nonzeros per column Application	1K	10K	20K
SchMP-Lasso 16 × 4 cores	125	212	202
SchMP-Lasso 16 × 8 cores	162	306	344
SchMP-LR 16 × 4 cores	75	98	103
SchMP-LR 16 × 8 cores	106	183	193

Table 5.2: Dynamic SchMP: DvT of SchMP-Lasso and SchMP-LR, measured as data samples (millions) operated on per second, for synthetic data sets with different column sparsity.

der of magnitude. Similar observations hold for distributed SchMP-SLR versus Shotgun-SLR (Figure 5.4d).



(a) Lasso with AD data

(b) Lasso with Synthetic data

(c) AD data

(d) SLR with Synthetic data

Figure 5.5: Dynamic SchMP: convergence time. Subfigures (a-d) correspond to Figure 5.4.

STRADS Dynamic Engine DvT analysis: Table 5.2 shows how the STRADS Dynamic Engine's DvT scales with increasing machines. We observe that DvT is limited by dataset density — if there are more nonzeros per feature column, we observe better DvT scalability with more machines. This is because the Lasso and SLR problems' model-parallel dependency structure limits the maximum degree of parallelization (number of parameters that can be correctly updated each iteration). Thus, Dynamic Engine scalability does not come from updating more parameters in parallel (which may be mathematically impossible), but from processing more data per feature column.

Dynamic SchMP on STRADS has low convergence times: Overall, both SchMP-Lasso and SchMP-SLR enjoy better convergence times than their Shotgun counterparts. The worst-case scenario is a single machine using a dataset (AD) with few nonzeros per feature column (Figure 5.5a). When compared with Figure 5.4a, SchMP DvT is much lower than Shotgun (Shotgun-Lasso converges faster initially), but ultimately SchMP-Lasso still converges 5 times faster. In the distributed setting (Figure 5.5b Lasso, Figure 5.5d SLR), the DvT penalty relative to Shotgun is much smaller and the curves resemble the OvD analysis (SchMP exhibits more than an order of magnitude speedup).

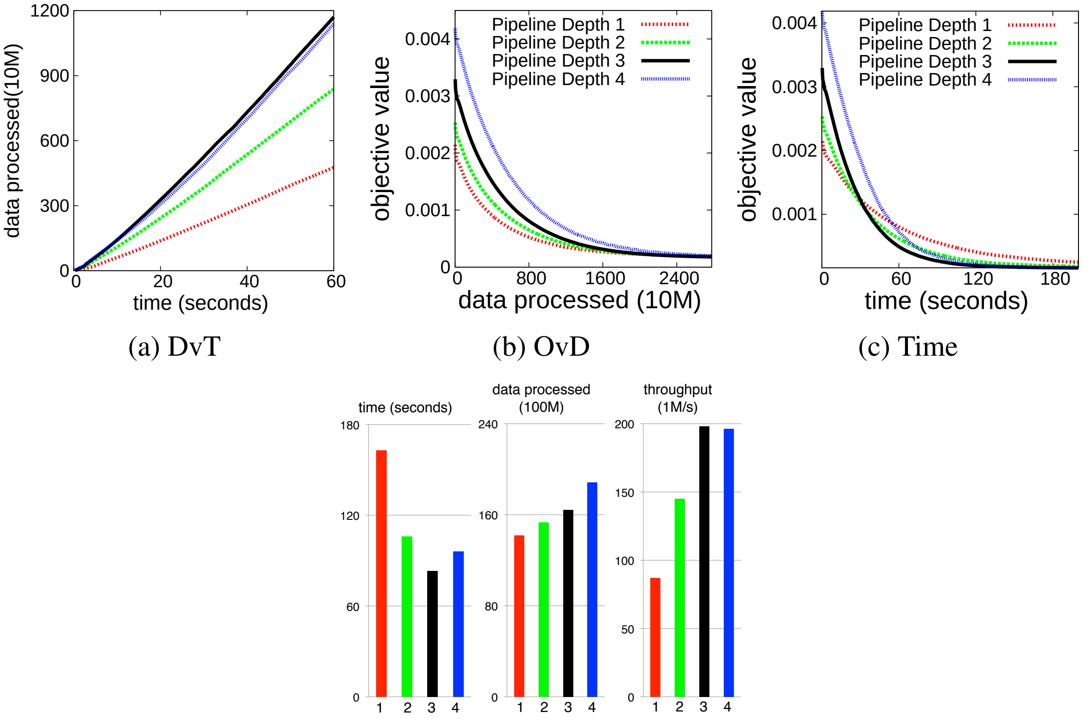
The evaluation of dynamic-schedule SchMP algorithms on the STRADS-Dynamic engine shows significantly improved OvD compared to random model-parallel scheduling. We also show that (1) in the single machine setting, Dynamic SchMP comes at a cost to DvT, but overall convergence speed is still superior to random model-parallel and (2) in the distributed setting, this DvT penalty mostly disappears.

5.3.2 Evaluation of dynamic engine optimizations

The STRADS Dynamic Engine improves DvT (data throughput) via iteration pipelining, while improving OvD via dynamic partitioning and prioritization in `schedule()`.

Impact of dynamic partitioning and prioritization: Figures 5.4c (OvD) and 5.5c (OvT) show that the convergence speedup from Dynamic SchMP comes mostly from prioritization — we see that dependency checking approximately doubles SchMP-Lasso's OvD over prioritization alone, implying that the rest of the order-of-magnitude speedup over Shotgun-Lasso comes from prioritization. Additional evidence is provided by Figure 3.3; under prioritization most parameters converge within just 5 iterations.

Pipelining improves DvT at a small cost to OvD: The STRADS Dynamic Engine can pipeline iterations to improve DvT (iteration throughput), at some cost to OvD. Figure 5.6c shows that SchMP-Lasso (on 8 machines) converges most quickly at a pipeline depth of 3, and Figure 5.6d provides a more detailed breakdown, including the time take to reach the same objective value (0.0003). We make two observations: (1) DvT improvement saturates at pipeline depth 3; (2) OvD, expressed as the number of data samples to convergence, gets proportionally worse as pipeline depth increases. Hence, the sweet spot for convergence time is pipeline depth 3, which halves convergence time compared to no pipelining (i.e. pipeline depth 1).



(a) DvT

(b) OvD

(c) Time

(d) Metrics at objective 3e-4

Figure 5.6: Dynamic Engine: iteration pipelining. (a) DvT improves 2.5× at pipeline depth 3, however (b) OvD decreases with increasing pipeline depth. Overall, (c) convergence time improves 2× at pipeline depth 3. (d) Another view of (a)-(c): we report DvT, OvD and time to converge to objective value 0.0003.

5.3.3 Comparison against other frameworks

We compare SchMP-Lasso/SLR with Spark MLlib (Spark-Lasso, Spark-SLR), which uses the SGD algorithm. In this experiment, we used 8 nodes with 64 cores and 128GB memory each. On the AD dataset (which has complex gene-gene correlations), Spark-Lasso reached objective value 0.0168 after 1 hour, whereas SchMP-Lasso achieved a lower objective (0.0003) in 3 minutes. On the LogisticSynthetic dataset (which was constructed to have few correlations), Spark-SLR converged to objective 0.452 in 899 seconds, while SchMP-SLR achieved a similar result.[6] This confirms that SchMP is more effective in the presence of more complex model dependencies.

[6] In the SchMP-Lasso/LR experiments, we did not include the overhead of checkpointing. We found that it is negligible ($<1\%$ of total execution time) and dominated by update computation time.

Chapter 6

Productivity of Developing Distributed ML

In this chapter, we present a case study that investigates the development cost of using high-level frameworks for developing distributed machine learning. Our case study shows that **a different mental model for programming** that a high-level framework requires a machine learning programmer to switch to from a familiar sequential programming model causes serious development overhead, and peculiarities of a high-level framework deliver **suboptimal training performance**. After the user study, we present an overview of our new approach STRADS-AP that allows a machine learning programmer to stay with a sequential programming model and simplifies distributed machine learning programming significantly while delivering performance comparable to hand-tuned distributed machine learning applications. In summary, this chapter:

- Conducts a case study that converts a sequential SGDMF code into distributed codes on Spark and STRADS-AP and compare development costs and runtime performance.

- Identifies two challenges that high-level frameworks impose for developing distributed machine learning.

- Presents an overview of STRADS-AP approach.

Algorithm 7 Pseudo code for SGDMF

```
$A$: a set of ratings. Each rating contains (i:user id, j:product id, r: rating)
$W$:$M \times K$ matrix; initialize $W$ randomly
$H$:$N \times K$ matrix; initialize $H$ randomly
for each rating r in $A$
        err = r.r - $W[r.i]H[r.j]$
        $\Delta$ $W$ = $\gamma$·(err*$H$[r.j] -$\lambda \cdot$ $W$[r.i] )
        $\Delta$ $H$ = $\gamma$·(err*$W$[r.i] -$\lambda\cdot$ $H$[r.j] )
        W[r.i] += $\Delta$ W
        H[r.j] += $\Delta$ H
```

6.1 Case Study

In the case study, we demonstrate that converting a sequential ML code into a high-level framework code requires large programming efforts and leads to poor performance that is an order of magnitude slower than STRADS-AP implementation, or a hand-tuned implementation.

As a concrete example, we choose Spark as the framework, and SGDMF (Stochastic Gradient Descent Matrix Factorization) as the algorithm — a popular recommendation system algorithm. As a baseline, first, we implement a sequential SGDMF in C++ by referring to the algorithm description in Algorithm 7. Then, we convert the sequential code into three different parallel codes — shared-memory OpenMP, Spark and STRADS-AP — and compare their performance.

6.1.1 SGDMF algorithm for recommendation system

This section summarizes matrix factorization (MF) model and stochastic gradient algorithm for optimizing matrix factorization model. Matrix factorization learns user's preferences over all productss from an incomplete rating dataset represented as a sparse matrix $A \in \mathbb{R}^{M \times N}$ where M and N are the number of users and products, respectively. It factorizes the incomplete matrix A into two low-rank $W \in \mathbb{R}^{M \times K}$ and $H \in \mathbb{R}^{N \times K}$ matrices such that $W \cdot H^T$ approximates A. Algorithm 7 iterates over the ratings in the matrix A. For each rating $r_{i,j}$, it calculates gradients $\Delta W[i]$, $\Delta H[j]$ and adds the gradients to $W[i]$, $H[j]$, respectively. The computed parameter values for the rating $r_{i,j}$ are immediately visible when computing the next rating, which is an example of asynchronous computation. Algorithm 7 represents a common way machine learning researchers present their

```
struct rating{
  int i,j; // i: user id, j: product id
  float s; // s: rating score
};
typedef rating T1;
typedef array<float, K> T2;
vector<T1> A = LoadRatings(Datafile_Path);
vector<T2> W(M);
vector<T2> H(N);
RandomInit(W);
RandomInit(H);
float gamma(.01f), lambda(.1f);
for(int iter=0; iter<maxiter; iter++){
  for(int k=0; k<A.size(); k++){
    const T1 &r = A[k];
    T2 err = r - W[r.i]*H[r.j];
    T2 Wd = gamma*(err*W[r.i]-lambda*H[r.j]);
    T2 Hd = gamma*(err*H[r.j]-lambda*W[r.i]);
    W[r.i] += Wd;
    H[r.j] += Hd;
  }
}
```

Figure 6.1: Sequential SGDMF: iterates over a vector of ratings, A. For a rating $A[k]$, which has rating score s for j-th product by i-th customer, it calculates gradients ΔW and ΔH for W_i and H_j, respectively and adds them to W_i and H_j, respectively. Note that the latest values of W_i and H_j are immediately visible for processing $A[k+1]$.

algorithm works in publication.

6.1.2 Sequential SGDMF

Sequential implementation of Algorithm 7 is a direct translation of the pseudocode as shown in Figure 6.1. We consider this sequential SGDMF as a baseline.

6.1.3 Shared-Memory SGDMF using OpenMP

As a intermediate step before moving on distributed SGDMF, we convert the sequential code in Figure 6.1 into a shared memory code using OpenMP[22]. We make two modifications to the sequential code as shown in Figure 6.2: annotate the inner loop with parallel-for pragma(line 17) ; and places mutexes(lines 20, 21, 27, 28) . OpenMP parallelizes the inner loop over loop indices using fork-join model where threads run the loop body with different loop indices and join at the completion of the loop. Use of mutexes

```
struct rate{
  int i, j;
  float s;
};
typedef rate T1;
typedef array<float, K> T2;
vector<T1> A = LoadRatings(Datafile_Path);
vector<T2> W(M);
RandomInit(W);
vector<T2> H(N);
RandomInit(H);
float gamma(.01f);
float lambda(.1f);
vector<mutex> WLock(M); // M: max user id
vector<mutex> HLock(N); // N: max product id
for(auto i(0);i<maxiter;i++){
  #pragma omp parallel for
  for(int k=0; k<A.size(); k++){
    const T1 &r = A[k];
    WLock(r.i).lock() // locks to avoid data race
    HLock(r.j).lock() //   on shared W,H matrices
    T2 err = r - W[r.i]*H[r.j];
    T2 Wd = gamma*(err*W[r.i]-lambda*H[r.j];
    T2 Hd = gamma*(err*H[r.j]-lambda*W[r.i];
    W[r.i] += Wd;
    H[r.j] += Hd;
    HLock(r.j).unlock()
    WLock(r.i).unlock()
    // Note that locks are released in reverse
    // ordering of obtaining to avoid deadlock
  }
}
```

Figure 6.2: Shared-Memory SGDMF: parallelizes update routine (loop body) using OpenMP primitives. To avoid race conditions on the shared parameters W and H, it places mutexes — HLock and WHlock — inside the loop and ensures serializability, which means its progress per iteration is comparable to that of sequential code.

inside the loop prevents data races on W and H submatrices and ensures serializability[1], which means its progress per iteration is comparable to that of sequential code.

6.1.4 Distributed SGDMF using Spark-Scala

We convert the sequential code into distributed Spark program. This conversion process requires significant programming effort as detailed below

[1] Serializability of a parallel execution means that the parallel execution has equivalent sequential execution.

```
val P = K  // number of executors
val ratings = sc.textFile(rfile, P).map(parser)
val blks=sc.parallelize(0 until P, P).persist()
val W = blks.map(a->Create_WpSubmatrix(a))
var H = blks.map(a->Create_HpSubmatrix(a))
var AW = ratings.join(W,P)
var AWH = AW.join(H,P).mapPartitions(a->ComputeFunc(a,0))
float gamma(.01f), lambda(.1f);
for(auto i(0);i<maxiter;i++){
  for(auto sub(0);sub<P;sub++){ // subiteration
    val idx = i*P + sub;
    if(idx > 0){
      AWH = AW(idx).join(H,P).
      mapPartitions(a->ComputeFunc(a,subepoch))
    }
    AW = AWH(idx).mapPartitions(x->separateAW_Func(x))
    H = AWH.map(x->separateH_and_Shift_Func(x))
  }
}
def ComputeFunc(it:Iterator to AWH){
  val tmp = ArrayBuffer[type of AWH]
  for(e <- it){
    val Ap = e.Ap
    val Wp = e.Wp
    val Hp = e.Hp
    for(auto r: Ap){
      if(r.2 not belong to Hp)
        continue //skip if not in Hp product indices
      val err = r.3 - Wp[r.1]*Hp[r.2]
      val Wd = gamma*(err*Wp[r.1]-lambda*Hp[r.2]
      val Hd = gamma*(err*Hp[r.2]-lambda*Wp[r.1]
      Wp[r.i] += Wd
      Hp[r.j] += Hd
    } // end of for(auto r ..
    tmp += Tuple2(e.key, ((Ap, Wp), Hp));
  } // end of for(e ..
  val ret = tmp.toArray
  ret.iterator
} // end of def update_Func
```

Figure 6.3: Spark-SGDMF: implements DSGD algorithm in [30]. It creates three RDDs, A, W, H for ratings, user matrix, and product matrix. A and W are joined onto AW based on user id. One iteration is divided into P subiterations. Each subiteration, it creates a temporary RDD AWH where a partition of AW get exclusive access to a partition of H using join operation, runs parameter update, and divides AWH into AW' and H' for next subiteration. Each subiteration, a AW partition is merged into a different H partition where partition is determined by exclusive range of product id.

Concurrency Control: Spark lacks concurrency control primitives. Since the inner loop of SGDMF leads to data dependencies when parallelized, we need to implement a scheduling plan for correct execution. Reasoning about concurrency control is application-specific and often requires a major design effort. For implementing a distributed SGDMF without concurrency issues, we implemented distributed stochastic gradient descent matrix factorization (DSGDMF) algorithm that was designed for implementing SGDMF on MapReduce system by Gemulla [30]. The scheduling code is at (lines 9-20), and the training code is at (lines 21 - 40) in Figure 6.3.

Molding SGDMF to Spark API: Even after obtaining a concurrency scheduling algorithm, implementing SGDMF in Spark requires substantial programming effort for the following reasons.

First, Spark operators such as *map*, operate on a single RDD object, while the inner loop body (= an update operation) in Figure 6.1(lines 14 – 21) accesses multiple objects: the input data A, and the parameter matrices W and H. To parallelize the inner loop with map we need to merge A, W, and H into a single RDD, requiring multiple *join* operations involving costly data shuffling.

Second, merging via *join* operator requires changes to data structures. Since the $join$ operator works only on the RDD[Key,Value] type, we have to replace the vectors A, W, H, in 6.1, with RDD[K,V] where V might be also key-value pair type.

Finally, data movement for concurrency control requires extra *join* and *map* operations. At the end of its every subiteration, DSGDMF moves H partitions among nodes, which requires two extra operations for every subiteration: (1) a *map* operation that separates H from the merged RDD and modifies the key field of H, (2) a *join* operation that remerges H and AW into AWH for the next subiteration, as shown in Figure 6.3 (lines 9-20.)

In summary, engineering the Spark implementation of SGDMF algorithm involves a large amount of *incidental* complexity that stems from the limitations of Spark API and its data abstractions. In addition to the loss in productivity, there is also a loss in efficiency. We will discuss this efficiency issue in Section 6.1.6.

6.1.5 Distributed SGDMF using STRADS-AP

Finally, we convert the sequential code into STRADS-AP code. This conversion is done almost mechanically by (1) replacing serial data structures with STRADS-AP's distributed data structures, and (2) replacing the inner loop with STRADS-AP's AsyncFor loop operator, as shown in Figure 6.4.

```
struct rate{int i;
  int j;
  float r
};
typedef rate T1;
typedef array<float, K> T2;
dvector<T1> &A = ReadFromFile(Datafile_Path, parser);
dvector<T2> &W = MakeDVector(M, RandomInit);
dvector<T2> &H = MakeDVector(N, RandomInit);
float gamma(.01f), lambda(.1f);
for(int  i=0;i<maxiter;i++){
  AsyncFor(0, A.size()-1, [gamma, lambda, &A,&W,&H](int k){
    const T1 rate &r = A[k];
    T2 err = r - W[r.i]*H[r.j];
    T2 Wd = gamma*(err*W[r.i]-lambda*H[r.j];
    T2 Hd = gamma*(err*H[r.j]-lambda*W[r.i];
    W[r.i] += Wd;
    H[r.j] += Hd;
  });
}
```

Figure 6.4: STRADS-AP SGDMF: implements distributed SGDMF by replacing vector with dvector and the inner for loop with AsyncFor – STRADS-AP's parallel loop operator that parallelizes loop bodies in isolated execution and ensures serializability. Note the similarity between STRADS-AP code and sequential code in Figure 6.1.

Unlike OpenMP and Spark codes, STRADS-AP code has no code for concurrency control (i.e. lock management in OpenMP or DSGDMF scheduling code in Spark). The runtime is responsible for addressing data conflicts on W and H matrices while executing loop body in a distributed setting, relieving users from writing error-prone locking code. With a little effort, STRADS-AP achieves efficient distributed parallelism, in addition to shared-memory parallelism. Note the similarity between the sequential code in Figure 6.1 and STRADS-AP code in Figure 6.4.

6.1.6 Performance cost

For performance comparison with hand-tuned distributed machine learning, we implement DSGDMF in [30] using MPI [28], which is more efficient at the cost of more programming efforts. MPI-SGDMF circulates the shared parameter H using point-to-point communication.

In our experiment, all distributed SGDMF implementations achieves proper concurrency control, making similar statistical progress per iteration. Therefore, our performance comparison focuses only on elapsed time for running 60 iterations, after which all implementations converge. We run experiments with Netflix dataset using up to 256 cores on

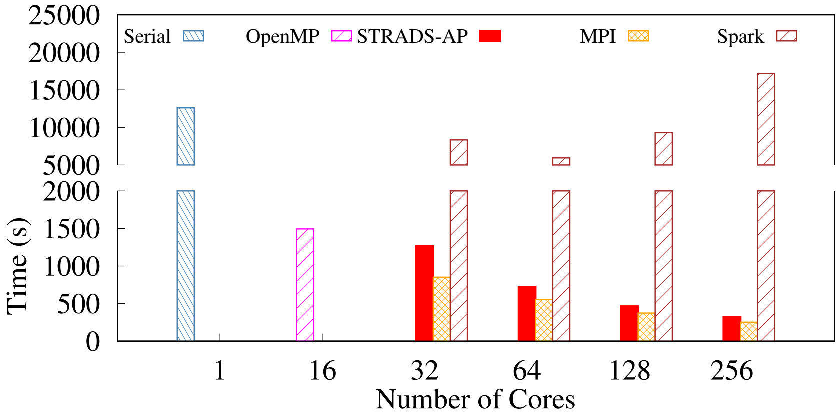


Figure 6.5: Time for 60 iterations with Netflix dataset[38], rank = 1000. STRADS-AP outperforms Spark by more than an order of magnitude (e) and continues to scale up to 256 cores, while Spark stops scaling at 64 cores. Hand-tuned MPI code is faster than STRADS-AP by 22% on 256 cores at the cost of a significantly longer programming and debugging effort.

16 machines that are connected via 40Gbps Ethernet.

As Figure 6.5 shows, the Spark is about 68× slower than MPI on 256 cores. In the same setting, STRADS-AP is slower than MPI by only 22%, whereas it is over 50× faster than Spark. The suboptimal performance of Spark implementation is due to aforementioned factors (Section 6.1.4 — invocation of expensive join operation every subiteration). STRADS-AP is 38.8 and 4.6 times faster than sequential and OpenMP, respectively.

6.1.7 Other high-level frameworks

Our findings of incidental complexity and suboptimal performance are not limited to the example of Spark and SGDMF. For example, PowerGraph provides concurrency control mechanisms, but *vertex centric* programming model requires users to redesign data structures to fit to a graph representation and express computations using GAS (Gather, Apply, Scatter) routines. TensorFlow provides super high-level programming model taking loss function and automates gradient update process but requires users to provide a data-flow graph that explicitly encodes data dependencies. Parameter Servers [25, 48, 19, 20, 88, 2, 21] abstract away the details of parameter communication through the key-value store interface but many other details of distributed parallel programming, such as data partitioning, parallelization of tasks, and the application-level concurrency control is left to the user.

	Requires Changing Programming Model	**Application-Level Concurrency Control**	**Hides Details of Distributed Programming**	**Fault Tolerance**
STRADS-AP	No	Yes	Yes	Yes(Checkpoint)
GraphLab	Yes (vertex-centric)	Yes	Yes	Yes(Checkpoint)
Spark	Yes (map/reduce/...)	No	Yes	Yes(RDD)
TensorFlow	Yes (data-flow)	No	Yes	Yes(Checkpoint)
Parameter Server	Yes (key-value)	No	Partly (parameter comm)	Yes(Replication)
MPI	Yes (message-passing)	No	Partly (communication)	No

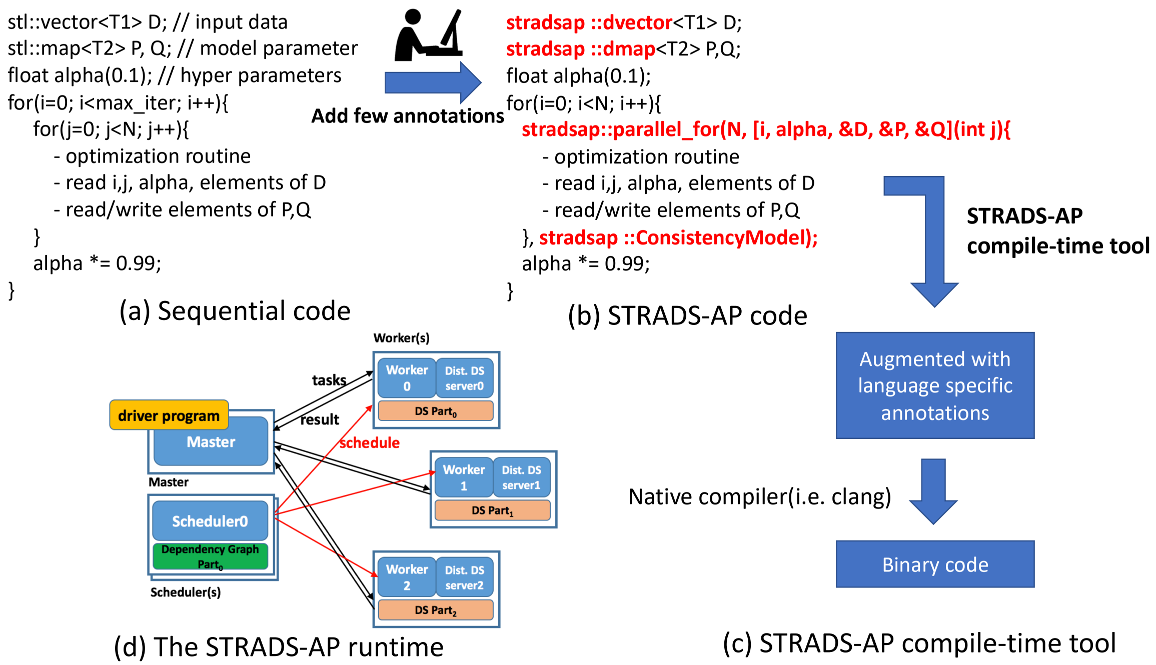
Table 6.1: Summary of features of frameworks used in distributed ML programming. For efficiency comparison, see evaluation section7.3

As Table 6.1 shows, STRADS-AP is the only framework that allows users to take their sequential code and automatically parallelize it to run on a cluster without sacrificing productivity or efficiency. STRADS-AP owes this flexibility to its familiar API and data structures that we will present in Chapter 7.

6.2 Overview of STRADS-AP

This section presents an overview of STRADS-AP framework. We believe that the complexity surrounding distributed ML programming as well as the inefficiency in execution are *incidental* and not *inherent*. That is, many sequential ML codes can be automatically parallelized to make near optimal use of cluster resources. To prove our point, we present STRADS-AP, a novel distributed ML framework that provides an API requiring minimally-invasive, mechanical changes to a sequential ML program code, and a runtime that automatically parallelizes the code on an arbitrary-sized cluster while delivering the performance of hand-tuned distributed ML programs.

STRADS-AP's API liberates machine learning programmers from the challenge of molding a sequential ML code to the framework's programming model. To achieve this, STRADS-AP API offers Distributed Data Structures (DDSs), a set of familiar containers, such as *vector* and *map*, allowing fine-grained read/write access to arbitrary elements, and two familiar loop operators. During runtime, these loop operators parallelize the loop bodies over a cluster following two popular ML parallelization strategies: asynchronous parallel execution, and synchronous parallel execution, with strong or relaxed consistency.



```
stl::vector<T1> D; // input data
stl::map<T2> P, Q; // model parameter
float alpha(0.1); // hyper parameters
for(i=0; i<max_iter; i++){
    for(j=0; j<N; j++){
        - optimization routine
        - read i,j, alpha, elements of D
        - read/write elements of P,Q
    }
    alpha *= 0.99;
}
```

(a) Sequential code

```
stradsap ::dvector<T1> D;
stradsap ::dmap<T2> P,Q;
float alpha(0.1);
for(i=0; i<N; i++){
  stradsap::parallel_for(N, [i, alpha, &D, &P, &Q](int j){
      - optimization routine
      - read i,j, alpha, elements of D
      - read/write elements of P,Q
  }, stradsap ::ConsistencyModel);
  alpha *= 0.99;
}
```

(b) STRADS-AP code

(c) STRADS-AP compile-time tool

(d) The STRADS-AP runtime

Figure 6.6: STRADS-AP workflow: (a) a machine learning programmer implements an ML algorithm in a sequential code; (b) Derives STRADS-AP parallel code with through mechanical changes; (c) STRADS-AP preprocessor adds more annotation to address language-specific constraints, and the source code is compiled by a native compiler; (d) The STRADS-AP runtime runs the binary in parallel on a cluster

STRADS-AP's workflow shown in Figure 6.6, starts with a machine learning programmer making mechanical changes to sequential code (Figure 6.6(a, b).) The code is then preprocessed by STRADS-AP's preprocessor and complied into a binary code by a C++ compiler (Figure 6.6(c).) Next, STRADS-AP's runtime executes the binary on nodes of a cluster while hiding the details of distributed programming (Figure 6.6(d).) The runtime system is responsible for (1) transparently partitioning DDSs that store training data and model parameters, (2) parallelizing slices of ML computations across a cluster, (3) fault-tolerance, and (4) enforcing strong consistency on shared data if required, or synchronizing partial outputs with relaxed consistency.

We implement STRADS-AP as a C++ library in about 16,000 lines of code.[2] We evaluate the performance on a moderate-sized cluster with four widely-used ML applications, using real data sets. To evaluate the increase in user productivity, we ask a group of students to convert a serial machine learning application to a distributed program using STRADS-AP, and we report our findings.

[2] Reported by CLOC tool, skipping blanks and comments.

Chapter 7

STRADS-AP API & Runtime System Implementation

In this chapter, we present a new framework STRADS-AP for simplifying distributed machine learning development. We benchmark STRADS-AP with three popular machine learning applications and conduct two user studies to evaluate potential productivity benefits of users. In summary. this chapter:

- Introduces STRADS-AP API in Section 7.1.
- Presents the STRADS-AP runtime system in Section 7.2.
- Evaluates training performance with three well-established machine learning applications (Word2vec, multi-class logistic regression, and SGDMF) in Section 7.3.1.
- Evaluates development productivity with a group of users in Section 7.3.2

7.1 STRADS-AP API

In this section, we present STRADS-AP API. We do not claim that STRADS-AP can automatically parallelize arbitrary machine learning programs. Instead, we restrict STRADS-AP to a group of machine learning programs with a specific structural pattern. First, we discuss this common structure of machine learning programs that STRADS-APtargets and present STRADS-APAPI that is not new but very similiar to sequential programming API while allowing a programmer to convert a sequential machine learning program — which has the common program structure — into a distributed program almost mechanically.

```
Create and initialize data structures D for input data
Create and initialize data structures P for model parameters
// .. run transformations on input data or parameter if necessary
Create and initialize hyper parameters V to control training
```

(a) Pretraining part

```
for(i=0; i<max_iter; i++){ // outer loop
   for(j=0; j<N; j++){     // inner loop
      // Computations for optimization happens here
      Read a part of input data D
      Read hyper parameters V and loop indexes i,j
      Read/writes to a part of model paraemters P
   }
   change hyper parameters
   if(stop condition is true)
     break;
}
```

(b) Training part

Figure 7.1: Machine learning applications targeted by STRADS are divided into two parts: (a) a pretraining part that creates data structures to store input data, model parameters, and hyper parameters; and (b) a training part with a nested loop structure that repeats a set of model parameter updates and a stopping condition check operation

7.1.1 Program structure of targeted machine learning applications

STRADS-AP targets machine learning applications with a common structural pattern consisting of two parts: (1) pretraining part that initializes the model and input data structures, and performs coarse-grained transformations; (2) training part that iteratively optimizes the objective function using nested loop(s) where inner loop(s) perform optimization computations—a pattern widely found in many ML algorithms.

To implement a STRADS-AP program, a user writes a simple *driver program* following the structure in Figure 7.1. In a driver program, a user declares hyper-parameters and invokes STRADS-AP data processing operators to create and transform DDSs in preprocessing part in Figure7.1(a), and then invokes STRADS-AP loop operators for optimization in training part in Figure 7.1(b). We describe each of these in the following sections.

Features	DDS	RDD	C++ STL Containers
Mutability	Mutable	Immutable	Mutable
Distribution	Yes	Yes	No
Reads	Coarse or fine-grained	Coarse or fine-grained	Coarse or fine-grained
Writes	Fine-grained	Coarse-grain	Fine-grained
Consistency	Automatic via Parallel_For	Trivial (immutable)	Up to app
Fault Recovery	Checkpoint	Lineage	Up to app
Size Change	Master-run code or through map/filter	Through map/filter	No constraint
Work Placement	Automatic based on data locality or app task scheduling	Automatic based on data locality	N/A

Table 7.1: Comparison of DDSs with Spark RDDs and C++ STL containers(sequential data structures

7.1.2 Distributed Data Structures (DDSs)

DDS[T] is a mutable in-memory container that partitions a collection of elements of type T over a cluster and provides a global address space abstraction with fine-grained read/write access and uniform access model independent of whether the accessed element is stored in a local memory or in the memory of a remote node. STRADS-AP offers three different types of distributed containers, dvector, dmap, and dmultimap, with a similar interface to their C++ STL counter parts.

These DDSs allow all threads running on all nodes to read and write arbitrary elements while unaware of details such as data partitioning and data placement. Support for distributed and fine-grained read/write access[1] gives STRADS-AP an important advantage over other frameworks. It allows reuse of data structures and routines from a sequential program by just changing the declaration of the data type. To better understand advantages of DDS, we compare DDS with RDD and STL sequential constainer in Table7.1. We describe the inner workings of DDSs in Section 7.2.3.

7.1.3 STRADS-AP operators

The two parts of machine learning programs, pretraining and training (7.1), have different workload characteristics. Pretraining is data-intensive, non-iterative, and embarrassingly-parallel, whereas training is compute-intensive and iterative, and the inner loop(s) may have data dependencies. STRADS-AP provides two sets of operators that allow natural expression of both types of computation.

[1] All DDSs support operator[] for fine-grained read/write access.

	Type	Description
Distributed Data Structures (DDSs)	dvector[T]	A distributed vector of type T elements
	dmap[K,V]	A distributed map of [K,V] element pairs of type K and V
	dmultimap[K,V]	A distributed multimap of [K,V] element pairs of type K and V
Data Processing Operators	DDS[T]& ReadFromFile(string **filename**, F **parser**)	Reads lines from **filename** and applies **parser** function to each line
	DDS[T2]& Map(DDS[T1] &**D**, F **UDF**)	Applies **UDF** to elements of **D** to generate DDS[T2].
	T2& Reduce(DDS[T1] &**D**, F **UDF**)	Reduces elements of **D** using **UDF** into a return value of type T2.
	void Transform(DDS[T] &**D**, F **UDF**)	Applies **UDF** to elements of **D** modifying them in-place.
Loop Operators	AsyncFor(int64 **S**, int64 **E**, F **UDF**)	Parallelizes **UDF** closure over indices [**S**, **E**] in isolated manner.
	SyncFor(DDS[T] **&D**, int **M**, F **UDF**, Sync **S**, bool **RE**)	Parallelizes **UDF** closure over minibatches of **D** each of size **M** using synchronization option **S** in data-parallel manner. **RE** parameter indicates whether to perform Reconnaissance Execution (7.2)

Table 7.2: A subset of STRADS-AP API—data processing operators for DDS transformation in pretraining, and loop operators for ML optimization in tra ining.

Data processing operators

As Table 7.2 shows, STRADS-AP provides operators for loading, storing, and creating DDSs. In addition, since MapReduce [26, 94] is an expressive API for embarrassingly-parallel computations, STRADS-AP provides Map, Reduce,Transform and Join operators. While these operators are a great fit for data processing that is typical in the pretraining part of an ML application, STRADS-AP puts no constraints on using them for expressing training computations.

Unlike previous implementations of these operators that generate many small tasks each of which process a chunk of data [26, 94, 13], we adopt a coarse-grained task approach to avoid the scheduling overhead [83]. STRADS-AP's runtime creates one process per machine taking into account data-placement, where each process launches as many threads as there are cores on the machine.

Loop operators for training

STRADS-AP provides loop operators shown in 7.2 to replace the inner loop(s) in the training part of machine learning programs in Figure 7.1. The loop operators take a user-defined closure as the loop body. The closure is C++ lambda expression that captures the specified DDSs and variables in the scope, and implements the loop body by reading from and writing to arbitrary elements of the captured DDSs. This allows to mechanically change the loop body of a sequential machine learning program to STRADS-AP code that is automatically parallelized.

STRADS-AP supports three models of parallelizing machine learning computations: (1) serializable asynchronous [53], (2) synchronous (BSP [85]), and (3) lock-free asynchronous (Hogwild![71]) within a node and synchronous across nodes (which we call Hybrid).

Currently, STRADS-AP offers two loop operators to support these models. A user can choose AsyncFor loop operator for serializable asynchronous model. For the remaining models a user can choose SyncFor operator and specify the desired model as an argument to the loop operator, as shown in 7.2. Other than choosing the appropriate loop operator, a user does not have to write any code for concurrency-control—STRADS-AP runtime will enforce the chosen model as described next.

AsyncFor parallelizes the loop over loop indices and ensures isolated execution of the loop bodies even if loop bodies have shared data. In other words, it ensures serializability: the output of the parallel execution matches the ordering of some sequential execution.

AsyncFor takes three arguments: the start index S, the end index $S + N$, and a C++ lambda expression F. It executes $N+1$ lambda instances, $F(S), F(S+1), \ldots, F(S+N)$ concurrently. At runtime, STRADS-AP partitions the index range $S \ldots S + N$ into P chunks of size C, and schedules up to P nodes to concurrently execute F with different indices. A node schedules multiple threads to run C lambda instances allowing arbitrary reads and writes to DDSs.

If the lambda expression modifies a DDS, then data conflicts will happen. Although ML algorithms are error-tolerant [36], some algorithms, like Coordinate Descent Lasso[81, 46], LDA[8, 91], and SGDMF[44, 30], converge slowly in the presence of numerical errors due to data conflicts. Following previous work [42], STRADS-AP runtime improves statistical progress by avoiding data conflicts using data conflict-free scheduling for lambda executions. Figure 6.4 shows an example use of AsyncFor operator for implementing SGDMF.

SyncFor parallelizes the loop over the input data. It splits input data into P chunks, where each chunk is processed by P nodes in parallel. Each node processes its data chunk, updating a local replica of model parameters.

SyncFor takes five arguments: the input data of type DDS[T], the size of a mini-batch M, a C++ lambda expression F, a synchronization option (BSP or Hybrid), and a flag indicating whether it should perform Reconnaissance Execution (Section 7.2.2). The runtime partitions the input data chunk of a node into L mini-batches of size M (typically L is much larger than the number of threads per node), and then schedules multiple threads to process mini-batches concurrently. A thread executes the lambda expression with a local copy of captured variables, and allows reads and writes only to the local copy while running F. At the end of processing a mini-batch, a separate per-node thread synchronizes the local copy of only those DDSs captured by reference across the nodes and synchronizes local threads, according to the sync option. Figure 7.2 shows an example use of SyncFor that reimplements Google's Word2vec model [34].

By default, SyncFor performs averaging aggregation of model parameters. Users can override this behavior by registering an application-specific aggregation function to a DDS through RegisterAggregationFunc() method.

7.2 STRADS-AP runtime system

In this section, we present core details of STRADS-APruntime system implementation: (1) the driver program execution; (2) Reconnaissance Execution; (3) DDS; (4) concurrency

```
typedef vector<word> T1;
typedef vector<array<float, vec_size>> T2;
dvector<T1> &inputD = ReadFromFile<T1>(path, parser);
dvector<T2> &Syn0 = MakeVector<T2>(vocsize, initrow1);
dvector<T2> &Syn1 = MakeVector<T2>(vocsize, initrow1);
float alpha = 0.025;
int W = 5, N = 10;
vector<int> &dtable = InitUnigramtable();
expTable &e = MakeExpTable();
for (int i = 0; i < maxiter; i++){
  SyncFor(inputD, mini-batchsize,[W, N, alpha, e, dtable, &Syn0, &Syn1](const vector<T1> &m){
    for (auto &sentence: m){
      //for each window in setence, pick up W words
      //  for each word in the window
      //    run N negative sampling using dist. table
      //    r/w to N rows of Syn0 and Syn1 tables
    }
  }, Hybrid, false);
}
```

Figure 7.2: Reimplementing Google's Word2vec model using STRADS-AP API.

Control; (5) preprocessor; and (6) STRADS-AP debugging facility.

7.2.1 Driver program execution model

The execution model of STRADS-AP follows the driver program model[94, 26]. In STRADS-AP, a programmer writes a visually straightline code for a sequential ML algorithm using STRADS-AP APIs as a driver program. In the driver program, machine learning programmers declare DDSs to store large input data and model paraemters and write ML computation code in the form of parallel loop operators. The statements in the driver program are classified into three categories: sequential statements, data processing statements, and loop statements. The runtime maintains a state machine with one state per category to keep track of the type of code to execute. A driver node starts the driver program in sequential state, and keeps sequential execution locally until the first invocation of a STRADS-AP operator. On STRADS-AP operator invocation, the state machine switches to the corresponding state and the runtime parallelizes the operator over multiple nodes. At the completion of the STRADS-AP operator, the runtime switches back to sequential state and continues running the driver program locally. The STRADS-AP runtime system consists of three types of nodes, and their roles are described in Figure 7.3.

The key challenges of STRADS-AP runtime design are: (1) full automation of concurrency control when parallelizing loop operators, and (2) reducing the latency of accessing DDS elements located on remote nodes. To address these challenges, STRADS-AP implements Reconnaissance Execution (RE).

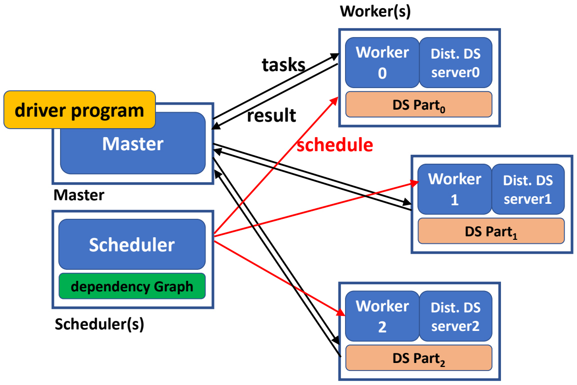


Figure 7.3: Driver program execution model: A driver program is a user-written straightline code that consists of sequential statements, data processing statement, and loop statements; Master node runs a driver program and launches STRADS-AP parallel operators over worker nodes; Scheduler nodes(s) generates dependency graph and make scheduling plans for worker nodes; A worker node run a set of worker threads that execute a slice of workload of a parallel operator and a DDS server thread.

7.2.2 Reconnaissance execution

The runtime system keeps the count of invocations of all loop operators in the driver program. On the first invocation of the loop operator, the runtime starts Reconnaissance Execution (RE)—a *virtual execution* of the loop operator. RE is a read-only execution that performs all reads to DDSs, and discovers read/write sets for individual loop bodies. A read/write *access record* of a loop body is a tuple of a DDS identifier, and a list of read/write element indices.

The runtime uses a read/write set for two purposes: (1) performing dependency analysis and generating data conflict-free scheduling plan for concurrent execution of loop bodies in AsyncFor operator, and (2) prefetching and caching of DDS elements on remote nodes for low-latency access during the *real execution*.

For the SyncFor operator, when the parameter access is sparse (that is, a small portion of parameters are accessed when processing a mini-batch), the runtime reduces amount of data transferred by referring to *access records* of RE. However, in applications with dense parameter access, (that is, most parameters are accessed when processing a mini-batch), RE does not help to improve the performance. Therefore, SyncFor operator's boolean RE parameter (7.2) allows users to skip RE and prefetch/cache all elements of DDSs captured by the corresponding lambda expression.

To reduce RE overhead, STRADS-AP runs it once per parallel loop operator in the driver program, and reuses read/write set for subsequent iterations. This optimization is based on two assumptions about ML workloads: (1) *iterativeness*—a loop operator is repeated many times until convergence, and (2) *static control flow*—read/write sets of loop bodies do not change over different iterations. That is, the control flow of the inner loop does not depend on model parameter values. Both assumptions are routinely accepted in ML algorithms [8, 91, 6, 2, 47, 44, 38, 90, 96, 67, 81, 12, 89, 29, 87, 51].

7.2.3 Distributed Data Structures

On the surface, a DDS is a C++ class template that provides index- or key-based uniform access operator. Under the hood, the elements of a DDS are stored on a distributed in-memory key-value store as key-value pairs. The key is uniquely composed of the table id plus the element index for *dvector*, and the table id plus the element key for *dmap/dmultimap*. Each node in a cluster runs a server of the distributed key-value store containing the elements of a DDS partitioned by the key hash. The implementation of DDS class template reduces the element access latency by prefetching and caching remote elements based on the access records generated by RE (Section 7.2.2).

The DDSs achieve fault-tolerance through checkpointing. At the completion of a STRADS-AP operator that runs on DDSs, the runtime makes snapshots of the DDSs that are modified or created by the operator. The checkpoint I/O time overhead is negligible because ML programs are compute-intensive, and the input data DDSs are not checkpointed (except once at creation), as they are read-only.

The traditional approach to checkpointing is to dump the whole program state onto storage during the checkpoint, and load the state from the last successful checkpoint during the recovery. Since an ML program may have an arbitrary number of non-DDS variables (like hyper-parameters), the traditional approach would require users to write boilerplate code for saving and restoring the state of these variables, reducing productivity and increasing opportunities for introducing bugs. Therefore, STRADS-AP takes a different approach to checkpointing that obviates the need for such boilerplate code.

Upon a node failure, STRADS-AP restarts the application program in *fast re-execution* mode. In this mode, when the runtime encounters a parallel operator *op* executing iteration i, it first checks to see whether a checkpoint for op_i exists. If yes, the runtime skips the execution of op_i and loads the DDS state from the checkpoint. Otherwise, it continues *normal execution*. Hence, the state of non-DDS variables are quickly and correctly restored without forcing the users to write extra code.

7.2.4 Concurrency control

STRADS-AP implements two concurrency control engines: (1) *serializable engine* for the AsyncFor operator, and (2) *data-parallel engine* for the SyncFor operator. Both engines use read/write set from Reconnaissance Execution (Section 7.2.2) for prefetching remote DDS elements, while serializable engine also uses it for making data conflict-free execution plans.

Serializable Engine for AsyncFor: In the serializable engine, a task is defined as the loop body with a unique loop index value i, which ranges from S to E, where S and E are AsyncFor arguments (Table 7.2). Serializable engine implements a scheduler module that takes the read/write set from RE, analyzes data dependencies, generates a dependency graph, and makes parallel execution plan that avoids data conflicts. To increase parallelism, serializable engine may change the execution order of tasks assuming that any serial reordering of loop body executions is acceptable. This assumption is also routinely accepted in ML computations [53, 66, 42].

The scheduler divides loop bodies into N *task groups*, where N is much larger than the number of nodes in a cluster, using an algorithm that combines the ideas of static scheduling from STRADS [42] and connected component-based scheduling from Cyclades [66]. The algorithm allows dependencies within a task group but ensures no dependency across task groups. At runtime, the scheduler places task groups on nodes, where each node keeps a pool of task groups.

To balance the load, serializable engine runs a greedy algorithm that sorts task groups in descending order of size, and assigns task groups to a node whose load is the smallest so far. Once task group placements are finalized, the runtime system starts the execution of the loop operator.

The execution begins by each node initializing DDSs to prefetch necessary elements from the key-value store into per-node DDS cache. Then each node creates a user-specified number of threads, and dispatches task groups from the task pool to the threads. All threads on a node access the per-node DDS cache without locking, since each thread executes a task group sequentially, and the scheduling algorithm guarantees that there will be no data conflicts across the task groups. In the case of an excessively large task group, we split it among threads and use locking to avoid data races, which leads to non-deterministic execution.

To reduce the scheduling overhead, serializable engine caches the scheduling plan and reuses it over multiple iterations based on the aforementioned assumptions (Section 7.2.2). Hence, the overhead of RE and computing a scheduling plan is amortized over multiple iterations.

Data-Parallel Engine for SyncFor: In data-parallel engine, a task is defined as the loop body with a mini-batch of D with size M, where D and M are SyncFor arguments (7.2). Hence, a single SyncFor call generates multiple tasks with different mini-batches. The engine places the tasks on nodes that hold the associated mini-batches, where nodes form a pool of assigned tasks.

Similar to serializable engine, the execution begins by each node initializing DDSs to prefetch necessary elements from the key-value store into per-node DDS cache, based on read/write set from RE, and continues by creating a user-specified number of threads.

Unlike serializable engine, the threads contain a per-thread cache, and are not allowed to access the per-node cache, since in this case there is no guarantee of data conflict-free access. When a node dispatches a task from the task pool to a thread, it copies the parameter values from the per-node cache to the per-thread cache.

Upon task completion, a thread returns the delta between the computed parameter values and the starting parameter values. The node dispatches a new task to the thread, accumulates deltas from all threads, and synchronizes per-node cache with the key-value store by sending the aggregate delta and pulling fresh parameter values.

The SyncFor operator allows users to choose among BSP and Hybrid (Section 7.1.3) models of parallelizing ML computations. The BSP [85] model is well-known, and our implementation follows previous work. Hybrid, on the other hand, is a lesser-known model [39]. It allows lock-free asynchronous update of parameters among threads within a node (Hogwild! [71]), but synchronizes across machines at fixed intervals. In the Hybrid model, a node creates a single DDS cache that is accessed by all threads without taking locks. When all of the threads complete a single task, which denotes a subiteration, the node synchronizes the DDS cache with the key-value store.

7.2.5 STRADS-AP preprocessor

While there exist mature serialization libraries for C++, none of them support serializing lambda function objects. The lack of reflection capability in C++, and the fact that lambda functions are implemented as anonymous function objects [58], makes serializing lambda challenging. We overcome this challenge by implementing a preprocessor that analyzes the source code using Clang AST Matcher library [18], identifies the types of STRADS-AP operator arguments, and generates RPC stub code and a uniquely-named function object for lambda expressions that are passed to STRADS-AP operators.

Dataset	Workload	Feature	Application	Purpose
Netflix	100M ratings	489K users, 17K movies rank=1000, data size=2.2GB	SGDMF	Recommendations
1Billion	1 billion words	Vocabulary size 308K, vector size=100, data size=4.5GB	Word2Vec	Word Embeddings
ImageNet	285K images	1K classes, 21,504 features, 24% sparsity, data size=21GB	MLR	Multi-Class Classification
FreeBase-15K	483K facts	14,951 entites, 1,345 relations, vector size=100, data size=36MB	TransE	Graph Embeddings

Table 7.3: Datasets used in our benchmarks.

The preprocessor also analyzes the source code to see if it declares DDSs of user-defined types. While DDSs of built-in types are automatically serialized using Boost Serialization library [9], for user-defined types the library requires adding a boilerplate code, which is automatically added by the preprocessor.

7.3 STRADS-AP evaluation

In this section, we evaluate STRADS-AP based on training performance and development productivity using a set of well known applications, such as SGDMF, word embedding[59, 60, 35], multi-class classification[7], and knowledge graph embedding[10, 87, 51, 11, 63, 64] as summarized in Figure 7.4. To further evaulate benefits of STRADS-AP, we conduct two user studies with a group of students that implement word embedding and knowledge graph embedding applications using STRADS-AP.

The main takeaway from our evaluation is that STRADS-AP improves the productivity significantly at the cost of reasonably small training speed. Our performance evaluation results show that STRADS-AP applications report smaller line counts than baseline distributed ones while outperforming a popular data-parallel framework (Spark) with a non-familiar programming model, achieving performance comparable to an ML-specialized framework (TensorFlow), and achieving $70 \sim 90$ percent of hand-tuned distributed ones (MPI-based applications). Our user study results presents that a programmer can easily convert the baseline sequential programs into STRADS-AP programs within few hours (less than 2 hours for both applications).

Application	Serial	OpenMP	MPI	STRADS-AP	TF	Spark
SGDMF	✓	✓	✓	✓		✓
MLR	✓	✓	✓	✓	✓	
Word2vec	✓	✓	✓	✓	✓	
TransE	✓			✓		

Table 7.4: ML programs used for benchmarking. Serial and OpenMP are single core and multi-core applications on a shared-memory machine, respectively, while the rest are distributed parallel applications. MPI applications use OpenMP for shared-memory parallelism within the nodes.

7.3.1 Productivity and performance evaluations

We evaluate STRADS-AP on (1) application performance, and (2) programmer productivity, using the following real world ML applications: SGDMF, Multinomial Logistic Regression (MLR)[7], Word2vec[60], and TransE [11], summarized in Table 7.4.

For performance evaluation, as a baseline we implement these applications as (1) a sequential C++ application, (2) a single-node shared-memory parallel C++ application using OpenMP, and (3) a distributed- and shared-memory parallel C++ application using OpenMP and MPI. We then compare the iteration throughput (time per epoch), and the statistical accuracy of Spark, TensorFlow (TF), and STRADS-AP implementations of these applications to those of baselines', while running them on datasets shown in Table 7.3. For productivity evaluation, we conduct two user studies on a group of students with Word2vec and TransE applications. As a measure of productivity, we count the lines of code produced, and measure the time it took students to convert a serial implementation of the algorithm into STRADS-AP implementation. All experiments were run on a cluster with 16 machines each with 64 GiB of memory and 16-core Intel Xeon E5520 CPUs, running Ubuntu 16.04 Linux distribution, connected with 40 Gbps Ethernet network.

Word2vec

Word2vec is a Natural Language Processing (NLP) model developed by Google that computes vector representations of words, called "word embeddings". These representations can be used for various NLP tasks, such as discovering semantic similarity. Word2vec can be implemented using two architectures: continuous bag-of-words (CBOW) or continuous skip-gram, the latter of which produces more accurate results for large datasets [34].

We implement the skip-gram architecture in STRADS-AP based on Google's open source multithreaded implementation [34] written in C. We make two changes to Google's implementation: (1) modify it to keep all the input data in memory to avoid I/O during

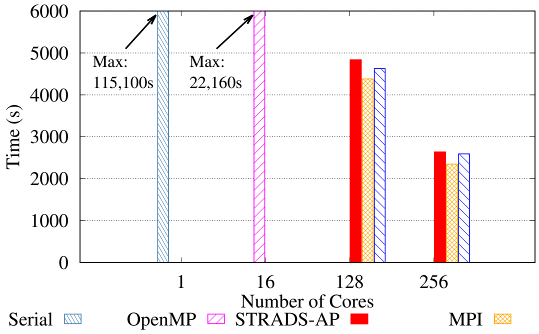


Figure 7.4: Time for 10 iterations for Word2Vec on 1 Billion word dataset[15] with vector size = 100, window = 5, negative sample count = 10.

training, and (2) replace POSIX threads with OpenMP. After our changes, we observe 6% increase in performance on 16 cores. We then run our improved implementation using a single thread for the serial baseline, and using 16 threads on 16 cores for the shared-memory parallel baseline.

Google recently released a highly-optimized multithreaded Word2vec [33] implementation on TensorFlow with two custom kernel operators written in C++. As of now, Google has not yet released a distributed version of Word2vec on TensorFlow. Therefore, we extend Google's implementation to run in a data-parallel distributed setting. To this end, we modify the kernel operators to work on partitions of input data, and synchronize parameters among nodes using MPI.

Cores	**Similarity**			**Analogy**		
	STRADS-AP	MPI	TF	STRADS-AP	MPI	TF
128	0.601	0.601	0.602	0.566	0.564	0.568
256	0.603	0.597	0.601	0.562	0.557	0.561
Serial	0.610			0.570		
OpenMP	0.608			0.571		

Table 7.5: The top table reports similarity test accuracy [27], and analogy test accuracy [60] for distributed Word2Vec implementations on 1 Billion word dataset, after 10 iterations. The bottom table shows respective values for the serial and OpenMP implementations.

Performance evaluation of Word2vec: Figure 7.4 shows the execution time of Word2vec for 10 iterations with 1 billion data set. On 256 cores (16 machines), MPI performs better than TensorFlow and STRADS-AP by 9.4% and 10.1%, respectively. The performance gap stems from the serialization overhead in TensorFlow and STRADS-AP. MPI imple-

Implementation	Word2vec	MLR	SGDMF
Serial	468	235	271
OpenMP	474	252	277
MPI	559	313	409
STRADS-AP	404	245	279
TensorFlow	646 (*)	155 (Python)	N/A
Spark	N/A	N/A	249 (Scala)

Table 7.6: Line counts of model implementations using different frameworks. Unless specified next to the lines counts, the language is C++. TensorFlow implementation of Word2vec has 282 lines in Python and 364 lines in C++.

mentation stores parameters in arrays of built-in types, and uses in-place MPI_Allreduce call to operate on the values directly. STRADS-AP outperforms serial and OpenMP implementation by 45× and 8.7×, respectively.

Table 7.5 shows the similarity test accuracy [27] and the analogy test [60] accuracy, after running 10 iterations. Using the accuracy of the serial algorithm as the baseline, we see that parallel implementations report slightly lower accuracy (within 1.1%) than the baseline due to the use of stale parameter values.

Productivity evaluation of Word2vec: Table 7.6 shows the line counts of Word2vec implementations in the first column. STRADS-AP implementation has 15% less lines than the serial implementation, which stems mainly from the coding style and the use of STRADS-AP's built-in text-parsing library. If we focus the comparison on the core of the program—the training routine—both implementations have around 100 lines, since STRADS-AP implementation takes the serial code and makes a few simple changes to it.

TensorFlow implementation, however, has three times more lines in the training routine. The increase is due to (1) splitting the training into two kernel operators to fit the dataflow model, (2) converting tensors into C++ Eigen library matrices and back, and (3) lock management.

While TensorFlow enables users to write simple models easily, it requires a lot more effort and knowledge, which most data scientists lack, to produce high-performance distributed model implementations with custom kernel operators. On the other hand, STRADS-AP allows ordinary users to easily obtain performance on par with the code that was optimized by Google, by making trivial changes to a serial implementation.

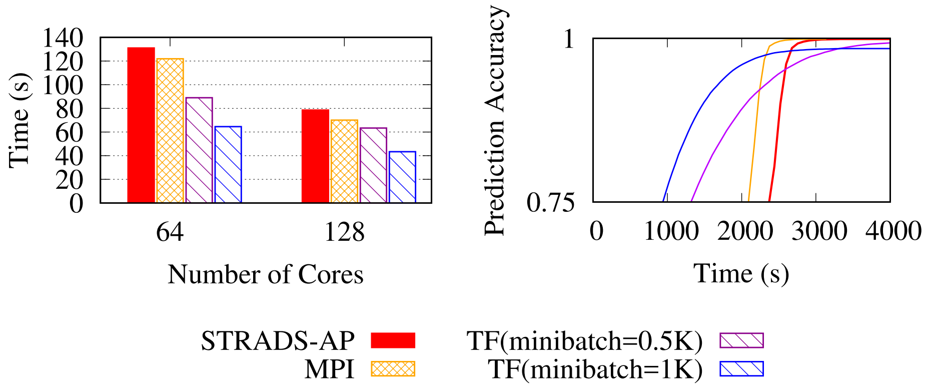


Figure 7.5: Left figure shows the time for a single iteration. We run TensorFlow implementation with a minibatch sizes of 500 and 1,000. STRADS and MPI implementations do not use vectorization, therefore, we run them with a minibatch size of 1. Serial and OpenMP implementations (omitted from the graph) also run with a minibatch size of 1, and take 3,380 and 467 seconds to complete, res pectively. Right figure shows the prediction accuracy as the training progresses. While each implementation runs for 60 iterations, the graph shows only the time until all of them converge to a stable value.

Multinomial Logistic Regression(MLR)

Multinomial Logistic Regression (MLR) [7] is a method for identifying the class of a new observation based on the training data consisting of observations and corresponding classes.

We implement a serial, OpenMP, MPI, TensorFlow, and STRADS-AP versions of MLR. Our distributed TensorFlow implementation uses parameter servers, and is based on the MNIST code in the TensorFlow repository. Similar to other implementations in our benchmark, our TensorFlow implementation preloads the dataset into memory before starting the training, and uses Gradient Descent optimizer.

Performance evaluation of MLR: Figure 7.5 shows the single iteration time on the 25% of ImageNet [73] dataset on the left, and the accuracy after 60 iterations on the right. TensorFlow makes heavy use of vectorization, which explains the 30% decrease in runtime when increasing the minibatch size from 500 to 1,000, and almost twice shorter runtime than MPI and STRADS implementations, which do not use vectorization.

On the other hand, as the right graph in Figure 7.5 shows, TensorFlow suffers in terms of accuracy. Unlike MPI and STRADS implementations that achieve 99.5% accuracy after about 2,800 seconds, the accuracy of TensorFlow remains under 98.4% even after 4,000 seconds. The difference in accuracy is due to STRADS and MPI implementations running with a minibatch size of 1, given that they do not use vectorization. A single iteration of TensorFlow with a minibatch size of 1 (for which it was not optimized) took about 6 hours, which we omitted from the graph. With vectorization support, STRADS can achieve on par performance with TensorFlow.

Productivity evaluation of MLR: Table 7.6 shows the line counts of MLR implementations in the second column. TensorFlow implementation has 38% and 50% fewer lines than STRADS-AP and MPI implementations, respectively, because while the latter implement large chunks of code to compute gradients and apply them to parameters, TensorFlow hides all of these under library function calls. On the other hand, most of the TensorFlow implementation consists of code for partitioning the data, setting up the cluster and parameter server variables.

Stochastic Gradient Matrix Factorization (SGDMF)

We already covered the implementation details in Section 6.1.6 and performance evaluation of solving Matrix Factorization algorithm using SGD optimization (SGDMF) in Figure 6.5. Therefore, we continue with the productivity evaluation.

Productivity evaluation of SGDMF: Table 7.6 shows line counts of SGDMF implementations in the third column. SGDMF implementation in Scala, even after including the line count for Gemulla's Strata scheduling algorithm (6.1.4), has about 15% fewer lines than STRADS-AP implementation. This is not surprising, given that functional languages like Scala tend to have more expressive power than imperative languages like C++. However, the difficulty of implementing the Strata scheduling algorithm is not captured well in the line count. Figuring out how to implement this algorithm using the limited Spark primitives, and tuning the performance so that the lineage graph would not consume all the memory on the cluster took us about a week, whereas deriving STRADS-AP implementation from the pseudocode took us about an hour. The line count of MPI is higher than serial code due to distributed SGDMF scheduling algorithm and manual data partitioning.

7.3.2 User Study

To further evaluate the productivity gains of using STRADS-AP, we conducted two more user studies.

User Study I: In the first study, as a capstone project we assigned a graduate student to implement a distributed version of Word2vec using STRADS-AP and MPI, after studying Google's C implementation [34]. The student had C and C++ programming experience, and had just finished an introductory ML course. After studying the reference source code, the student spent about an hour studying STRADS-AP API and experimenting with it. It then took him about two hours to deliver a working distributed Word2vec implemented with STRADS-AP API. On the other hand, it took the student two days to deliver a distributed Word2vec implemented with MPI. The MPI implementation was able to match STRADS-AP implementation in terms of accuracy and performance after two weeks of performance optimizations.

User Study II: In the second study, we conducted an experiment similar to a programming exam, with five graduate students. We provided the students with a two-page STRADS-AP API documentation, an example serial MLR code, and the corresponding STRADS-AP code. We then gave the students a serial C++ program written by an external NLP research group that implemented TransE [11] knowledge graph embedding algorithm, and asked them to produce a distributed version of the same program using STRADS-AP.

Table 7.7 shows the breakdown of times each student spent at different phases of the experiment, including the students' backgrounds, and the primary challenges they faced. While most students lacked proficiency in C++, they still managed to complete conversion in a reasonable time. Student 5, who was the most proficient in C++, finished the experiment in 1.5 hours, while Student 1 took 2.4 hours, most of which he spent in the last subtask debugging syntax errors, after breezing through the previous subtasks. The feedback from the participants indicated that (1) converting serial code into STRADS-AP code was straightforward because data structures, the control flow, and optimization functions in the serial program were reusable with a few changes, and (2) the lack of C++ familiarity was the main challenge. The list of reported mistakes included C++ syntax errors, forgetting to resize local C++ STL vectors before populating them, and an attempt to create a nested DDS, which STRADS-AP does not currently support. We evaluated the students' implementations by running them on FreeBase-15K[11] dataset for 1000 iterations with vector size of 50. The students' implementations were about 22× faster than the serial implementation on 128 cores, averaging at 45.3% accuracy, compared 46.1% accuracy of the serial implementation.

Subject	Major (Main PL)	C++ Skill	[T1]	[T2]	[T3]	[T4]	[T5]	Total	Challenges
Student 1	Data Mining (Python)	Low	0.25	0.1	0.25	0.1	1.7	2.4	Lack of C/C++ experience
Student 2	Data Mining (Java)	Low	0.3	0.2	0.1	0.2	1.4	2.2	Lack of C/C++ experience
Student 3	Machine Learning (Python)	Low	0.3	0.5	0.5	0.5	1	2.8	Lack of C/C++ experience
Student 4	Compilers (Java)	High	0.3	0.3	0.2	0.1	1.0	1.9	Lack of ML programming familiarity
Student 5	Systems (C++)	High	0.25	0.25	0.5	0.25	0.25	1.5	N/A

Table 7.7: The breakdown of times (in hours) of five students that converted the serial implementation of TransE [11] graph embedding algorithm to a distributed STRADS-AP implementation. We split the conversion task into five subtasks: [T1] understand the algorithm, [T2] understand the reference serial code, [T3] review STRADS-AP API guide, [T4] review the provided serial MLR code and the corresponding STRADS-AP code, [T5] convert the serial implementation to STRADS-AP implementation.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this dissertation, we made arguments regarding computational efficiency and programming productivity of distributed machine learning.

First, we argued that the performance of data-parallel machine learning is limited by fundamental trade-offs between system throughput and statistical progress per iteration. Though relaxed consistency models and parameter servers can improve training performance through the trade-offs, data-parallel machine learning still has residual staleness, which lowers progress per iteration resulting in lower efficiency.

To improve efficiency further, we proposed the model parameter update scheduling approach SchMP that (1) avoids data conflicts while running concurrent updates, which improves progress per iteration by eliminating staleness or reducing it further, and (2) prioritizes important parameter updates, which makes more progress per update by avoiding updates on already converged parameters. For different computational dependency structures, we presented two different parameter update scheduling schemes (Static/Dynamic SchMP). In STRADS system, we implement system optimizations, such as distributed scheduler for high scheduling throughput, pipelining for overlapping computation and communication, and ring overlay network for reducing synchronization overhead.

In our evaluations, we show that STRADS-Static/Dynamic engines achieve system throughput as high as data-parallel machine learning applications while preserving the benefits of parameter update scheduling – achieving progress per iteration close to or better than that of ideal sequential execution.

Second, we argued that we can simplify distributed machine learning programming with familiar sequential programming model without introducing a new programming model. Though high-level frameworks like MapReduce/GraphLab/Spark simplify distributed programming by hiding essential requirements of distributed computing from a machine learning programmer, these frameworks require a machine learning programmer to give up familiar sequential programming model and to switch to a different mental model for programming – functional programming model in MapReduce/Spark and vertex-centric programming model in GraphLab – which lowers development productivity.

To simplify distributed machine learning programming, we proposed a sequential-like interface, STRADS-AP API, which allows a machine learning programmer to convert sequential machine learning code into distributed code by making few mechanical changes. The API consists of DDSs (i.e. distributed STL containers) and two loop operators (i.e. AsyncFor and SyncFor) that allow a machine learning programmer to reuse data structures and computation routines from a sequential program with minimal changes. To show the feasibility of STRADS-AP API, we present the STRADS-AP runtime system that implements Reconnaissance Execution for profiling dependencies of updates and parameter access patterns, DDS containers with prefetching/caching capabilities, two concurrency control engines for running Async and Sync loop operators with strong or relaxed consistency models.

In our performance evaluations, we show that STRADS-AP machine learning applications achieve training performance comparable to that of hand-tuned distributed applications and counterparts on a ML specialized framework. In our productivity evaluations, we show that STRADS-AP API keeps line counts of applications close to those of sequential counterparts, which are less that those of hand-tuned distributed codes. Through two user studies, we show that converting a sequential machine learning program into a distributed program using STRADS-AP API is straightforward and can be done easily in less than three hours, which was much shorter than development time of hand-tuned distributed programming using MPI.

8.2 Future work

While this dissertation improves efficiency and productivity of general machine learning[1], there are several directions that could improve efficiency and productivity further for general machine learning.

[1] They are machine learning algorithms except neural network algorithms.

Higher Productivity: These days super high-level frameworks, such as TensorFlow and PyTorch, for neural network computing simplify machine learning programming significantly by supporting auto-differentiation and auto-gradient updates. We believe that these techniques can be applied to a general machine learning framework like STRADS-AP. The challenge is that general machine learning algorithms use a wider variety of numerical operators as compared to neural network algorithms which use a limited well-established set of operators. To address this problem, it's essential to build a library of operators for general machine learning – which would cover the broader range of operators than neural network library and require aggregated efforts from the machine learning and system communities.

Higher performance: As neural network computing gets popular, more clusters are harnessing GPUs. Most high-level machine frameworks except neural network frameworks do not support GPUs (or vector instructions of modern CPUs). Utilization of these advanced hardware features is left to a machine learning programmer, and these features are usually ignored for general machine learning programming. We think that it's essential to support these features for improving general machine learning performance. When building the aforementioned operator library, these advanced hardware features should be utilized.

Bibliography

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. 2, 2.2.5

[2] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J Smola. Scalable Inference in Latent Variable Models. In *WSDM*, 2012. 2.1.2, 2.2.4, 4.1.3, 6.1.7, 7.2.2

[3] Galen Andrew and Jianfeng Gao. Scalable training of l1-regularized log-linear models. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 33–40, New York, NY, USA, 2007. ACM. 5

[4] Apache Hadoop, http://hadoop.apache.org. 2.2.2

[5] Apache Mahout, http://mahout.apache.org. 2.2.2

[6] Arthur Asuncion, Max Welling, Padhraic Smyth, and Yee Whye Teh. On Smoothing and Inference for Topic Models. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pages 27–34, Arlington, Virginia, United States, 2009. AUAI Press. 7.2.2

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 1, 1.2, 7.3, 7.3.1, 7.3.1

[8] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003. 1, 1.2, 7.1.3, 7.2.2

[9] Boost. Boost C++ Library - Serialization. 7.2.5

[10] Antoine Bordes, Xavier Glorot, Jason Weston, and Yoshua Bengio. A semantic matching energy function for learning with multi-relational data. *Mach. Learn.*, 94(2):233–259, February 2014. 7.3

[11] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Durán, Jason Weston, and Oksana Yakhnenko. Translating Embeddings for Modeling Multi-relational Data. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 2787–2795, USA, 2013. Curran Associates Inc. (document), 1.2, 7.3, 7.3.1, 7.3.2, 7.7

[12] Joseph K. Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel Coordinate Descent for L1-Regularized Loss Minimization. In *ICML*, 2011. (document), 3.1.3, 3.2.1, 5.3, 5.4, 5.3.1, 7.2.2

[13] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert Henry, Robert Bradshaw, and Nathan. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010. 2, 7.1.3

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association. 1.1.1

[15] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. Technical report, Google, 2013. (document), 7.4

[16] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. 2

[17] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric P. Xing. Solving the straggler problem with bounded staleness. In *HotOS*. USENIX Association, 2013. 2.1.1

[18] Clang. AST Matcher,http://clang.llvm.org/docs/LibASTMatchersReference.html. 7.2.5

[19] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 37–48, Philadelphia, PA, June 2014. USENIX Association. 2, 2.1.3, 2.2.4, 6.1.7

[20] Henggang Cui, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky, Qirong Ho, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting Iterative-ness for Parallel ML Computations. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 5:1–5:14, New York, NY, USA, 2014. ACM. 2, 2.1.3, 2.2.4, 6.1.7

[21] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 4:1–4:16, New York, NY, USA, 2016. ACM. 2, 6.1.7

[22] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998. 6.1.3

[23] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth A. Gibson, and Eric P. Xing. High-performance distributed ML at scale through parameter server consistency models. In *AAAI*, 2015. 3.2.2

[24] Jeff Dean, David A. Patterson, and Cliff Young. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29, 2018. 1

[25] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1223–1231, USA, 2012. Curran Associates Inc. 6.1.7

[26] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008. 2, 2.1.1, 2.2.2, 7.1.3, 7.2.1

[27] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. Placing Search in Context: The Concept Revisited. In *Proceedings of the 10th international conference on World Wide Web*, pages 406–414. ACM, 2001. (document), 7.5, 7.3.1

[28] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994. 2, 2.1.1, 2.2.1, 6.1.6

[29] J. Friedman, T. Hastie, H. Hofling, and R. Tibshirani. Pathwise Coordinate Optimization. *Annals of Applied Statistics*, 1(2):302–332, 2007. 3.1.3, 5.1.2, 7.2.2

[30] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent. In *KDD*, 2011. (document), 6.3, 6.1.4, 6.1.6, 7.1.3

[31] Alexander Genkin, David D. Lewis, and David Madigan. Large-scale bayesian logistic regression for text categorization. *Technometrics*, 49(3):291–304, 2007. 5

[32] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, volume 12, page 2, 2012. 2, 2.2.3

[33] Google. TensorFLow Optimized Word2vec. 7.3.1

[34] Google. word2vec, https://code.google.com/archive/p/word2vec/. 1.2, 7.1.3, 7.3.1, 7.3.2

[35] Saurabh Gupta and Vineet Khare. Blazingtext: Scaling and accelerating word2vec using multiple gpus. In *Proceedings of the Machine Learning on HPC Environments*, MLHPC'17, pages 6:1–6:5, New York, NY, USA, 2017. ACM. 7.3

[36] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *NIPS*, 2013. 2.1.3, 7, 2.2.4, 3.2.2, 7.1.3

[37] Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multiclass support vector machines. *Trans. Neur. Netw.*, 13(2):415–425, March 2002. 5

[38] James Bennett and Stan Lanning and Netflix Netflix. The Netflix Prize. In *In KDD Cup and Workshop in conjunction with KDD*, 2007. (document), 6.5, 7.2.2

[39] Shihao Ji, Nadathur Satish, Sheng Li, and Pradeep Dubey. Parallelizing Word2Vec in Multi-Core and Many-Core Architectures. *CoRR*, abs/1611.06172, 2016. 7.2.4

[40] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM. 2

[41] George Karypis and Vipin Kumar. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995. 3.2.1

[42] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 5:1–5:16, New York, NY, USA, 2016. ACM. 3.3, 7.1.3, 7.2.4

[43] Kwangmoo Koh, Seung-Jean Kim, and Stephen Boyd. An interior-point method for large-scale l1-regularized logistic regression. *J. Mach. Learn. Res.*, 8:1519–1555, December 2007. 5

[44] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix Factorization Techniques for Recommender Systems. *Computer*, (8):30–37, 2009. 1, 1.2, 7.1.3, 7.2.2

[45] John Langford, Er J. Smola, and Martin Zinkevich. Slow learners are fast. In *In NIPS*, pages 2331–2339, 2009. 3.1.1

[46] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth Gibson, and Eric P. Xing. On model parallelism and scheduling strategies for distributed machine learning. In *NIPS*. 2014. 7.1.3

[47] Aaron Q. Li, Amr Ahmed, Sujith Ravi, and Alexander J. Smola. Reducing the Sampling Complexity of Topic Models. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 891–900, New York, NY, USA, 2014. ACM. 7.2.2

[48] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *OSDI*, 2014. 2, 2.1.3, 2.2.4, 3.2.2, 6.1.7

[49] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 19–27. Curran Associates, Inc., 2014. 2.1.3

[50] Chih-Jen Lin and Jorge J. Moré. Newton's method for large bound-constrained optimization problems. *SIAM J. on Optimization*, 9(4):1100–1127, April 1999. 5

[51] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 2181–2187. AAAI Press, 2015. 7.2.2, 7.3

[52] Jun Liu, Jianhui Chen, and Jieping Ye. Large-scale sparse logistic regression. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 547–556, New York, NY, USA, 2009. ACM. 5

[53] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012. 2, 2.2.3, 3.2.2, 3.2.3, 7.1.3, 7.2.4

[54] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010. 2, 2.2.3, 3.2.3

[55] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. 2

[56] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. *CoRR*, abs/1803.04014, 2018. 2.2.5

[57] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin,

Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, January 2016. 2.2.2

[58] Microsoft Developer Network. Lambda Expressions in C++, 2015. 7.2.5

[59] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013. 7.3

[60] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA, 2013. Curran Associates Inc. (document), 7.3, 7.3.1, 7.5, 7.3.1

[61] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013. 1.2

[62] MPICH2. `http://www.mcs.anl.gov/mpi/mpich2`. 2, 2.2.1

[63] Maximilian Nickel, Lorenzo Rosasco, and Tomaso Poggio. Holographic embeddings of knowledge graphs. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 1955–1961. AAAI Press, 2016. 7.3

[64] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 809–816, USA, 2011. Omnipress. 7.3

[65] OpenMPI. https://www.open-mpi.org/. 2, 2.2.1

[66] Xinghao Pan, Maximilian Lam, Stephen Tu, Dimitris Papailiopoulos, Ce Zhang, Michael I Jordan, Kannan Ramchandran, and Christopher Ré. Cyclades: Conflict-free Asynchronous Machine Learning. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2568–2576. Curran Associates, Inc., 2016. 7.2.4

[67] István Pilászy, Dávid Zibriczky, and Domonkos Tikk. Fast ALS-based Matrix Factorization for Explicit and Implicit Feedback Datasets. In *Proceedings of the Fourth ACM Conference on Recommender Systems*, RecSys '10, pages 71–78, New York, NY, USA, 2010. ACM. 7.2.2

[68] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010. 2, 2.2.4

[69] PyTorch. https://pytorch.org/. 2

[70] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. 2.2.5

[71] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*, 2011. 2.1.2, 3.1.1, 7.1.3, 7.2.4

[72] Peter Richtárik and Martin Takáč. Parallel coordinate descent methods for big data optimization. *arXiv preprint arXiv:1212.0873*, 2012. 4

[73] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115(3):211–252, December 2015. 7.3.1

[74] Chad Scherrer, Mahantesh Halappanavar, Ambuj Tewari, and David Haglin. Scaling up parallel coordinate descent algorithms. In *ICML*, 2012. 4

[75] Chad Scherrer, Ambuj Tewari, Mahantesh Halappanavar, and David Haglin. Feature clustering for accelerating parallel coordinate descent. In *NIPS*. 2012. 3.2.1

[76] Shai Shalev-Shwartz and Ambuj Tewari. Stochastic methods for ℓ_1 regularized loss minimization. In Léon Bottou and Michael Littman, editors, *Proceedings of the 26th International Conference on Machine Learning*, pages 929–936, Montreal, June 2009. Omnipress. 5

[77] Spark. Apache spark 2.3.1. 2.1.1

[78] Cheng tao Chu, Sang K. Kim, Yi an Lin, Yuanyuan Yu, Gary Bradski, Kunle Olukotun, and Andrew Y. Ng. Map-reduce for machine learning on multicore. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19*, pages 281–288. MIT Press, 2007. 2.2.2

[79] Choon Hui Teo, S. V. N. Vishwanathan, Alexander J. Smola, and Quoc V. Le. Bundle methods for regularized risk minimization. *Journal of Machine Learning Research*, 11:311–365, 2010. 5

[80] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, December 2013. 2.1.3

[81] R. Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. 1.2, 7.1.3, 7.2.2

[82] R. Tibshirani, M. Saunders, S. Rosset, J. Zhu, and K. Knight. Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(1):91–108, 2005. 1

[83] Ehsan Totoni, Subramanya R. Dulloor, and Amitabha Roy. A Case Against Tiny Tasks in Iterative Analytics. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 144–149, New York, NY, USA, 2017. ACM. 7.1.3

[84] Paul Tseng and Sangwoon Yun. A coordinate gradient descent method for nonsmooth separable minimization. *Math. Program.*, 117(1-2):387–423, 2009. 5

[85] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990. 7.1.3, 7.2.4

[86] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018. 2.2.5

[87] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 1112–1119. AAAI Press, 2014. 7.2.2, 7.3

[88] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Managed Communication and Consistency for Fast Data-parallel Iterative Analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 381–394, New York, NY, USA, 2015. ACM. 2, 2.2.4, 4.3.6, 6.1.7

[89] T.T. Wu and K. Lange. Coordinate Descent Algorithms for Lasso Penalized Regression. *The Annals of Applied Statistics*, 2(1):224–244, 2008. 7.2.2

[90] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon. Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems. In *ICDM*, 2012. 7.2.2

[91] Jinhui Yuan, Fei Gao, Qirong Ho, Wei Dai, Jinliang Wei, Xun Zheng, Eric P. Xing, Tie-Yan Liu, and Wei-Ying Ma. LightLDA: Big Topic Models on Modest Compute Clusters. In *WWW*, 2015. 7.1.3, 7.2.2

[92] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012. 2.1.1

[93] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. 2.1.1

[94] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM. 1.2, 2, 7.1.3, 7.2.1

[95] Bin Zhang, Chris Gaiteri, Liviu-Gabriel Bodea, Zhi Wang, Joshua McElwee, Alexei A Podtelezhnikov, Chunsheng Zhang, Tao Xie, Linh Tran, Radu Dobrin, et al. Integrated systems approach identifies genetic nodes and networks in late-onset Alzheimer's disease. *Cell*, 153(3):707–720, 2013. 3.1.3, 5.1.2

[96] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel Collaborative Filtering for the Netflix Prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008. 7.2.2

[97] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *NIPS*, 2010. 3.1.1