STRADS: A New Distributed Framework for Scheduled Model-Parallel Machine Learning

Jin Kyu Kim
jinkyuk@cs.cmu.edu

Thesis Proposal

Computer Science Department
Carnegie Mellon University

Committee
Garth A. Gibson (Co-chair)
Eric P. Xing (Co-chair)
Phillip B. Gibbons
Joseph E. Gonzalez

April 24, 2017
## Contents

1 Introduction .......................... 1

2 Part I: Parallel ML Challenges & SchMP .......................... 5
2.1 Challenges of Parallel Machine Learning .......................... 5
2.1.1 Example I: Challenges in Lasso .......................... 7
2.1.2 Example II: Challenges in LDA .......................... 8
2.2 Scheduled Model Parallel .......................... 10
2.2.1 Variations of Model Parallel .......................... 11
2.2.2 Programming Scheduled Model Parallel .......................... 12

3 Part II: Static-SchMP Challenges & STRADS-Static Engine 13
3.1 Static Scheduling and Challenges .......................... 13
3.1.1 Static Scheduling .......................... 14
3.1.2 LDA/SGD-MF with Static Scheduling .......................... 15
3.1.3 System Design Challenges .......................... 17
3.2 STRADS-Static Engine .......................... 17
3.2.1 Ring overlay-network for addressing synchronization overhead .......................... 18
3.2.2 Job Pool for multi-threading worker node .......................... 18
3.2.3 Prioritization in task dispatching .......................... 19
3.3 Selected Results .......................... 19

4 Part III: Dynamic-SchMP Challenges & STRADS-Dynamic Engine 21
4.1 Dynamic Scheduling and Challenges .......................... 22
4.1.1 Dynamic Scheduling .......................... 22
4.1.2 Lasso with Dynamic Scheduling .......................... 22
4.1.3 System Design Challenges .......................... 23
4.2 STRADS-Dynamic Engine .......................... 24
4.2.1 Distributed Scheduler .......................... 24
4.2.2 Pipelining .......................... 24
4.2.3 Task reordering .......................... 24
4.3 Selected Results .......................... 25

5 Proposed Work: Automatic Parallelization for STRADS 27
5.1 Problems and Motivation .......................... 27
5.2 Proposed Approach .......................... 28
5.2.1 Program structure of target MLDM applications .......................... 28
5.2.2 STRADS-AP Programming API .......................... 30
5.3 STRADS-AP Infrastructure Implementation .......................... 32
5.3.1 Runtime System .......................... 32
5.3.2 Compile-time System .......................... 34
5.4 Support programming model of other frameworks .......................... 35
5.5 Testing and Debugging Parallel ML on STRADS-AP

6 Related Work

7 Timeline
1 Introduction

Over the last two decades, the Internet and storage media technologies have improved at a breakneck pace, leading to massive collections of data. Data analytic techniques in the fields of machine learning and data mining (MLDM) have improved significantly. These improvements have caused quantum leaps in the quality of a wide range of technologies, from speech recognition to autonomous driving. Because of these factors, MLDM is a primary tool to analyze and explore big datasets. It supports high-level decision making and has extended the traditional data processing flow with an MLDM pipeline, as shown in Figure 1.

The data processing flow begins with the ETL stage, which extracts data from various sources, then transforms the data to address semantic mismatches by defining a structure. Finally, it loads the data into a central repository (a data warehouse). 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. NoSQL databases, such as BigTable [10], HBase and DynamoDB, were developed to maintain data in a data warehouse. The data in a data warehouse is used by MLDM applications and other traditional data warehouse applications, such as OLAP and visualization tools.

In this proposal, I exclusively focus on the MLDM application. Data analysis in MLDM consists of feature engineering, model training, and prediction. Feature engineering creates training data by extracting attributes of raw input data relevant to ML algorithms. Model training is an iterative process to search for the parameter values that best represent the training data. Training usually starts with randomly initialized parameter values and stops when the stopping condition is satisfied (i.e. 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.

Due to the iterative properties of MLDM algorithms, the training stage usually serves as a bottleneck for the entire MLDM pipeline. The exponential growth of training datasets prevents the completion of training in a timely manner on a single machine with limited computation resources.
Such a training constraint drives the industry toward distributed parallel MLDM.

To parallelize training computation, ML and systems communities have traditionally resorted to the **data-parallel** approach [28, 33, 13, 1]. Each machine receives a partition of data. Then, each machine iteratively scans the data partition and generates sub-updates that are applied to all ML model parameters (which are accessible with read/write operations from all machines), until convergence is reached. This straightforward way to parallelize ML training is possible because most ML algorithms adopt the so-called *i.i.d* assumption on data — that is, under a given estimate of the model parameters, data instances are independent and identically distributed. Therefore, all machines’ sub-updates can be aggregated easily. Many embarrassingly-parallel ML applications, such as gradient descent optimization algorithms, fit well into this approach.

More recently, an alternative strategy called **model-parallel** was introduced [30, 16, 19]. In the model-parallel approach, a machine is assigned a partition of model parameters and each worker updates a portion of model parameters using either a partition of data or the whole data set. The model-parallel approach is a promising way to achieve more effective parallel computation, particularly when the size of the model parameters is large. However, this approach introduces new subtleties: (1) an update of model parameters often has data dependencies with other model parameter updates (referred to as “model dependency”) and (2) different model parameters may require a different number of iterations to converge. Hence, the effectiveness of a model parallel algorithm is greatly affected by its parameter update scheduling strategy. Such a strategy specifies (1) which parameters are updated in parallel, (2) how serialized updates are sequenced, and (3) how computations of such updates are prioritized.

Traditionally, the ML community has focused on verifying the theoretical correctness of parallel ML algorithms while ignoring practical challenges in a distributed environment such as communication overhead and synchronization. In systems literature, the system community has developed distributed frameworks that automate the execution of parallel ML and exempt ML developers from low-level distributed programming burdens such as network communication, synchronization, and fault tolerance. However, many existing systems [17, 45, 3] treat ML applications as a black box and focus on correct execution by adhering to an overly strict synchronization scheme. This process is much like traditional computing that decreases parallelism to ensure correct execution of programs. It ignores ML’s unique properties such as error tolerance, model dependency, and uneven convergence, thus leaving major opportunities for performance improvements. Therefore, existing systems cannot provide stronger support for model-parallel algorithms.

In this thesis, I propose a holistic approach that takes both machine learning literature and system literature into account to improve the training speed of parallel ML. On the ML side, I propose the **Scheduled Model-Parallel (SchMP)** approach, which schedules independent model parameters for parallel execution and prioritizes model parameters based on their contributions to convergence per update. Instead of adhering only to improving statistical progress, SchMP has the flexibility to trade statistical progress for system throughput during scheduling to improve the overall time to answer. On the system side, I design and implement a new distributed framework, **STRADS**, that parallelizes SchMP ML applications in a distributed environment. The STRADS system enables the trading of a portion of statistical progress for gains in system throughput. During the execution of the schedule plan, STRADS intentionally compromises the quality of the schedule.
plan to a certain degree to enable system optimization techniques that improve system throughput. Compromising the schedule plan might reduce progress per update. However, system throughput gains compensate for this negative effect. Thus, this trading capability could help reduce the time to answer. Together, SchMP and STRADS implementation improve training speed by an order of magnitude. However, SchMP requires manual programming of scheduling functions. To eliminate this programming requirement, I will extend STRADS to automatically translate a sequential ML program into a STRADS executable parallel ML program by exploiting the unique properties of ML programs.

My thesis statement is:

*Using the ability to make trade-offs between statistical accuracy and system throughput in theoretical ML scheduling and its practice in a distributed environment allows the training speed of model parallel ML applications to be improved by an order of magnitude, and a high-level programming library with an infrastructure can automate ML scheduling for a wide range of MLDM applications.*

To support this thesis statement, I conducted three studies. (1) I investigated major challenges in parallel ML and proposed the SchMP (Schedule Model Parallel) approach for making faster convergence per update compared to non-scheduled parallel ML. (2) I designed and implemented a STRADS-Static engine that executes a static scheduling plan. (3) I designed and implemented a STRADS-Dynamic engine that schedules model parameters dynamically. To remove the programming burden of model parameter scheduling, I propose an extension of STRADS with a template library-based programming API that simplifies the development of SchMP ML programs.

- In my first study, I investigated major challenges in parallel ML with two popular ML applications: Lasso[40] and LDA topic modeling[7]. My study revealed that common data-parallel implementations of these algorithms 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, I propose scheduled model-parallel (SchMP), which schedules model parameters in a way that bounds the degree of dependency among concurrently updated model parameters to a threshold and prioritizes model parameters according to the contribution to convergence.

- In my second study, I explored Static-Scheduling for machine learning algorithms, in which a parameter update schedule can be planned prior to run time. Here, I address two questions: (1) What conditions of ML programs should be satisfied before choosing static-scheduling? and (2) What system optimizations should be supported to efficiently execute static-scheduled ML applications? On the ML side, I identified conditions for static-scheduling and presented a generic static-scheduling algorithm. On the system side, I presented a high-throughput STRADS-Static engine that implements two system optimizations: a ring overlay network and a strategy to address the load balancing problem.

- In my third study, I explored Dynamic-Scheduling for machine learning algorithms that
have an error-tolerant dependency structure\(^1\) and an uneven convergence rate\(^2\). This work is divided into two parts. On the machine learning side, I present a two-phase scheduling algorithm that improves statistical progress by respecting the dependency structure among parameter updates and considering the uneven convergence rate of model parameters. To maximize training speed, the dynamic scheduler intentionally 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 higher than the loss from reduced progress per update. On the system side, I present a STRADS-Dynamic engine that implements a distributed scheduler and pipelining of update operations to improve update throughput. The pipelining allows the \((t + 1)\)-th iteration\(^3\) 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\(^4\) 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.

Finally, I propose STRADS-AP(Automatic Parallelization) that relieves ML programmers of ML scheduling burdens and improves STRADS usability. This study focuses on building a C++ template library that consists of (1) distributed data structures (DDS), (2) data processing functions that operate on DDS, and (3) Parallel\_For function that concurrently updates model parameters while ensuring consistency on the model parameters. Once ML programmers write their applications using API of STRADS-AP, STRADS-AP run-time system internally records read/write accesses on the parameters for every parameter update operation before starting actual parameter update operations. During running actual parameter update operations STRADS-AP makes scheduling plans based on these read/write access records and execute only independent update operations in parallel. Therefore, STRADS-AP guarantees that parallel execution of update operations has an equivalent sequential execution.

\(^1\)An error-tolerant dependency structure is a unique property of ML computation; an ML algorithm can absorb a certain degree of 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.

\(^2\)An uneven convergence rate is a property of ML computation; different model parameters may take different number of updates to converge.

\(^3\)An iteration is a group of independent update operations.

\(^4\)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.
In this proposal, I present the preliminary results of the first three studies and a work plan for the proposed study. In Section 2, I investigate two unique challenges of model-parallel ML and propose Scheduled Model-Parallelism (SchMP). Section 3 describes the static scheduling scheme and STRADS-Static engine. A dynamic scheduling scheme and STRADS-Dynamic engine will be discussed in Section 4. In Section 5, I propose a research plan for STRADS-AP.

2 Part I: Parallel ML Challenges & SchMP

In this chapter, I will present two main challenges of parallel ML and show how these challenges affect the statistical progress of ML applications. Then, I will present the SchMP (Scheduled Model Parallel) approach, which addresses these challenges. In particular, this chapter:

- Explores two challenges in parallelizing ML in model-parallel approach: 1) model dependency and 2) uneven convergence.
- Investigates their effects on convergence rate with two well-established ML applications, l1-regularized sparse regression (Lasso) and LDA topic modeling, and
- Presents abstract expression of scheduled model parallel (SchMP) to address the challenges of parallelizing ML applications

The ML community has invented a range of machine learning models, most of which are reduced to optimization problems (tuning model parameters to minimize errors or maximize likelihood) for which algorithms are made. These algorithms are implemented in computer programs that generally take an iterative loop structure. Each iteration, the program repeats a fixed parameter update routine to search for better model parameter values that reduce errors or increase likelihood, until it reaches convergence, taking the following form:

\[ A^{(t)} = F(A^{(t-1)} + \Delta(D, A^{(t-1)})) , \]  

where index \( t \) refers to the current iteration, \( A \) are the model parameters, \( D \) is the input data, \( \Delta() \) is the model update function, and \( F \) is a summation function of \( A^{(t-1)} \) and \( \Delta() \). These iterative-convergent algorithms have several unique properties that we shall use or consider when improving ML training speed: tolerance to numerical errors in model parameter values, dependency structures that must be respected during parallel execution, and uneven convergence rates across model parameters.

2.1 Challenges of Parallel Machine Learning

The aforementioned data-parallel approach is based on the assumption that the collection of data samples – random variables in statics parlance – is independent and identically distributed (i.i.d.)
and enables 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), ML algorithms can converge as long as staleness is not arbitrary large, thanks to the error-tolerance of the ML algorithm. Theoretical correctness guarantees are found in [33, 28, 47]. The logic behind the correctness proof is that the potential errors from stale model parameters by sub-updates offset each other to be zero, or at least small enough to be tolerated when aggregated. However, the data parallel approach presents several challenges: 1) stale parameter values diminish convergence progress per computation, which requires more iterations to reach convergence than a serial algorithm, 2) it requires a large amount of memory to store a replication of model parameters on each worker, and 3) the strict synchronization aggravates the straggler problem.

In order to address these challenges, recent work [32, 29, 1, 13, 21] has adopted the parameter server approach, which provides abstraction of distributed shared memory in the form of key-value store. It also provides a flexible consistency model that aims to optimize communication overhead and the staleness of model parameters. Separate server machines hold a partition of model parameters, aggregate sub-updates from workers, and serve parameter read request from workers. Although the parameter server approach reduces the staleness problem, which improve progress per iteration, it cannot mitigate the staleness problem completely. Thus, parallel ML implementations on parameter server still take longer iterations to achieve convergence than ideal serial implementations. Furthermore, the demand for large memory in a worker machine is not resolved completely, as it is often necessary to cache substantial amount of model parameters to achieve reasonable performance.

Recently, the need for stronger explanatory power has driven the trend towards larger ML models with more parameters (i.e. hundreds of millions [27, 16]). This makes it highly desirable for the ML model to scale with the size of the aggregated memory in a cluster. As these big model problems emerge, the model-parallel approach is introduced in [16] to scale out big-model ML computing. Note that big-model problems don’t necessarily have large data. 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. If necessary, model repartitioning is conducted in run time. A model-parallel ML extends equation (1) to the following form:

\[ A(t) = A(t-1) + \sum_{p=1}^{P} \Delta_p(D, A(t-1), S_p(D, A(t-1))), \]  

where \( \Delta_p() \) is the model update function executed at parallel worker \( p \). The “schedule” \( S_p() \) outputs a subset of parameters in \( A \), which tells the \( p \)-th parallel worker which parameters it should work on 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 model-partitioning, or at random without considering data-dependency and uneven convergence. Since the data \( D \) is unchanging, I leave out \( D \) from the form for clarity:

\[ A(t) = A(t-1) + \sum_{p=1}^{P} \Delta_p(A(t-1), S_p(A(t-1))). \]  

Model parameters are partitioned over workers. Thus, the demand for memory to keep model-parameters on each worker is reduced to \( \frac{M}{P} \), where \( M \) is the size of model and \( P \) is the number
of workers. Unlike data samples, model-parameters are often dependent on each other – **model-dependency**. Updating dependent model parameters at the same time can cause numerical errors that tend to increase the number of iterations to reach convergence and cause computation to be less effective. Another important property of ML computation is the uneven convergence rate of model parameters, meaning that some model parameters require far more iterations than others. Thus, these few parameters bottleneck the entire ML training – **uneven convergence**.

In the following sections, I show examples of the model dependency challenge and the uneven convergence challenge with two well-established ML applications. To do this, I implement parallel versions of the Lasso and LDA topic modeling algorithms without considering model dependency and uneven convergence.

### 2.1.1 Example I: Challenges in Lasso

**Sequential Lasso:** Lasso is a $\ell_1$-regularized least-squares regression that is used to identify a small set of important features from high-dimensional data. Lasso is an optimization problem:

$$
\text{minimize } \frac{1}{2} \sum_{i=1}^{n} (y_i - h_\beta(x_i))^2 + \lambda \|\beta\|_1
$$

where $\ell_1$-regularizer $\|\beta\|_1 = \sum_{j=1}^{d} |\beta_j|$ and $\lambda$ is a preset user parameter that controls the number of zero entries in $\beta$. $X$ 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 $X$. $\beta$ is an m-by-1 coefficient vector. During the training phase, we learn the $\beta$ vector for a given $X, Y$ by solving equation 7. The update rule in Lasso is:

$$
\beta_j^{(t)} \leftarrow S(x_j^Ty - \sum_{k\neq j} x_j^T x_k \beta_k^{(t-1)}, \lambda),
$$

where $S(\cdot, \lambda)$ is a soft-thresholding operator [18]. Lasso is an inherently serial algorithm, since the operation of updating $\beta_k$ takes all $\beta_i$ where $i = 1, \ldots, n$ and $i \neq k$ in equation(5).

**Parallel Lasso:** Lasso can be parallelized by updating multiple coefficients in parallel. I implemented a parallel Lasso algorithm[9] that updates a set of randomly selected model parameters in each iteration.

**Model dependency challenge in Lasso:** In practice, the random selection of update parameters does not always fill the set $S$ (a set of model parameter to update in parallel per iteration) with independent model parameters, particularly when the design matrix $X$ is dense. 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 2 shows progress per data processed using parallel lasso with an Alzheimer’s Disease (AD) data set [46]. 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 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.
Figure 2: **Random Model-Parallel Lasso**: Objective value (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.

Figure 3: **Uneven Parameter Convergence**: The number of converged parameters at each iteration, with different regularization parameters $\lambda$. Red bar shows the percentage of converged parameters at iteration 5.

**Uneven convergence rate challenge in Lasso**: In many ML applications, the convergence rate of model parameters is uneven and has a biased distribution. For example, in e-commerce sales prediction, the design matrix $X$ for training is a huge, two dimensional matrix, where rows represent customers, and columns represent products. It is very natural that the popularity of products is uneven and skewed. 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 converge 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 shows histograms of the number of iterations to convergence for approximately a half million parameters in Lasso with the AD data set using three different regularization parameter ($\lambda$) values. It shows that about 85 percent, 95 percent, and 98 percent of parameters require less than five iterations to converge respectively with $\lambda$ 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.

### 2.1.2 Example II: Challenges in LDA

**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,
Parallel LDA: It is common to simply parallelize LDA over the document collection, while ignoring the dependency among tokens. Figure 4b 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 \( p \)). On the other hand, the word topic table \( W \) is globally shared since all workers access the table to make the probability vector for sampling. In the figure 4b, 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.

Figure 4: Gibbs sampling operator access on data structures.
Model dependency challenge in LDA: The parallel LDA in Figure 4b 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 topic table $V$, which diminishes progress (gains) per iteration. Figure 5 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.

2.2 Scheduled Model Parallel

Usually, convergence per update of sequential execution is ideal because sequential execution does not cause numeric errors that are usually introduced in parallel execution. Therefore, we often use the serial algorithms’ convergence rate per update as an ideal baseline. Scheduled model parallel (SchMP) has two goals. First, it aims to improve convergence per update of parallel ML so that it is close to that of sequential ML by scheduling model-dependency. Second, it tries 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), SchMP harnesses scheduling function $S$ with dependency checking and prioritization routines. Sometimes, it is neither practical nor possible to find a “perfect” parallel execution scheme for a ML 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 essen-
tially progress per iteration multiplied by iteration throughput, it is important to balance these two
considerations. Below, I explore this idea by explicitly discussing some variations within model
parallelism in order to expose possible ways by which model parallelization can be made more
efficient.

2.2.1 Variations of Model Parallel

I restrict my attention to model-parallel programs that partition $M$ model parameters across $P$
workers in an approximately load-balanced manner; highly unbalanced partitions are inefficient
and undesirable. Here, I introduce variations on model-parallelism, which differ on their parti-
tioning quality. Concretely, partitioning involves constructing a size-$M^2$ dependency graph, with
weighted edges $e_{ij}$ that measure the dependency between parameters $A_i$ and $A_j$. 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 $S_p(\cdot)$ in equation (3).

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 parallelization achieves the highest progress per iteration amongst load-
balanced model-parallel programs, but may incur a large one-time, or even every-iteration, parti-
tioning cost, which can greatly reduce iteration throughput.

**Random Model-Parallel:** At the other extreme is random model parallelization, in which a sched-
ule $S_p^{rand()}$ simply chooses one parameter at random for each worker $p$ [9]. 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 paralleliza-
tion, we may approximate $S_p^{ideal()}$ via a cheap-to-compute schedule $S_p^{approx()}$. A number of strate-
gies exist: one may partition small subsets of parameters at a time (instead of the $M^2$-size full
dependency graph), apply approximate partitioning algorithms [37] such as METIS [25] (to avoid
NP-hard partitioning costs), or use strategies that are unique to a particular ML program’s structure.

In this study, my goal is to explore strategies for efficient and effective approximate model
parallelization. I focus on ideas for generating model partitions and schedules:

**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 ML 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 parameters by
performing pair-wise dependency tests between a small number $L$ of parameters (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 parameters), 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-parallelism, 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 $S_p^{\text{dyn}}()$.

**Stale Input to Updating and Scheduling:** This is not a different type of model-parallelism, 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:

$$A^{(t)} = A^{(t-1)} + \sum_{p=1}^{P} \Delta_p(A^{(t-s)}, S_p(A^{(t-s)})),$$

(6)

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 ML algorithms are error-tolerant, they can still converge under stale model images (up to a practical limit) [21, 15]. 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 [30], while avoiding parameters that are already converged [29]. This is effective because 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). Prioritization can thus greatly improve progress per iteration, at a small cost to iteration throughput.

### 2.2.2 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 [31, 30], which “activates” a parameter whenever one of its neighboring parameters changes. I contrast this with scheduled model parallelism (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 $\Delta_p()$.

In the following chapters, I will introduce two scheduling schemes – static scheduling and
dynamic scheduling – based on Static Partitioning and Dynamic Partitioning as well as present two runtime execution engines that exploit Stale Input to Updating and Scheduling and Prioritization.

3 Part II: Static-SchMP Challenges & STRADS-Static Engine

In this chapter, I focus on static ML algorithms, in which schedule plans can be made prior to runtime. Because the scheduling plan is fixed prior to runtime, static-schedule functions tend to be computationally light, while dynamic scheduling is computationally intensive. However, static scheduling requires ML algorithms to meet some constraints. Here, I identify the static scheduling constraints that static ML algorithms should satisfy and present a general static scheduling algorithm on the ML side. On the system side, I implement a static engine that achieves high system throughput of static algorithms in a distributed environment.

If the targeting ML algorithms do not satisfy the constraints, the dynamic scheduling can serve them. Or, one might use a dynamic schedule on static ML algorithms 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, I separately provide two distinct but complete implementations of their scheduling. This chapter focuses on the STRADS-Static engine. The STRADS-Dynamic engine will be addressed in the following chapter. This chapter:

- Explores program structures of static ML algorithms that we can exploit to make schedule plans prior to runtime.
- Presents a generic static scheduling algorithm that takes dependency structure into account.
- Explains the design of the STRADS-Static engine that addresses two system challenges of static scheduling: (1) synchronization overhead and (2) the load balancing problem. The static engine reduces synchronization overhead by circulating model parameters along the ring overlay network of worker nodes. To address the load balancing problem, the static engine prioritizes heavy update operations when selecting a task to execute and communicate within a node.
- Quantifies the benefits of static scheduling and the STRADS-Static engine using three different metrics. My evaluation shows that LDA topic modeling with static scheduling on the static engine increases training speed by six times compared to Yahoo!LDA, which is a carefully designed distributed LDA on a fully asynchronous parameter server.

3.1 Static Scheduling and Challenges

As noted in Sec 2.2.1, static scheduling implements Static Partitioning. In each iteration (or subiteration), the partitioning of update operations (parameters) is hard-coded beforehand so that it incurs little cost for iteration throughput. The training data is partitioned over worker nodes. Shared
Algorithm 1 static scheduling where $k = 1$

M: the number of worker nodes
$P_i$: a parameter shard where $0 < i < M - 1$

while Until Convergence do
  for ($i = 0; i < M; i++$) {
    worker[0] = access permission to $P[(i + 0)\% M]$
    worker[1] = access permission to $P[(i + 1)\% M]$
    ...  
    worker[$M - 2$] = access permission to $P[(i + M - 2)\% M]$
    worker[$M - 1$] = access permission to $P[(i + M - 1)\% M]$
  }
end while

Algorithm 2 static scheduling for $k$-partite algorithm

M: the number of worker nodes
$P_0, P_1, ..., P_{k-1}$: K different types 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 $P_0[(i_0 + 0)\% M], P_1[(i_1 + 0)\% M], ..., P_{k-1}[(i_{k-1} + 0)\% M]$
        worker[1] = access to $P_0[(i_0 + 1)\% M], P_1[(i_1 + 1)\% M], ..., P_{k-1}[(i_{k-1} + 1)\% M]$
        ...  
        worker[$M - 1$] = access to $P_0[(i_0 + M - 1)\% M], P_1[(i_1 + M - 1)\% M], ..., P_{k-1}[(i_{k-1} + M - 1)\% M]$
      }
    }
  }
end while

parameters are partitioned by $L$ partitions, and are stored in a distributed key-value store. Each iteration is divided into subiterations. The main idea of static scheduling is that each node is scheduled such that one can see a sequence of $k$ partitions permuted from $L$ partitions, ensuring that each partition scheduled at a subiteration is assigned to a single worker node. Running all subiterations of one iteration ensures that all worker nodes go through all possible permutation sequences from $L^P_k$ at least once. Based on the dependency structure of the ML algorithm, there are subtle changes to the scheduling details. I will discuss them in the following sections.

3.1.1 Static Scheduling

To help ML programmers check whether their target ML applications can be statically scheduled, I present the properties of static ML algorithms. First, I categorize ML 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. Static scheduling only targets non-variadic algorithms, whereas variadic algorithms can be supported by dynamic scheduling. I further categorize non-variadic ML algorithms into two cases, according to the dependency graph when $k$ is fixed: (1) $k$-partite algorithms and (2) non-$k$ partite algorithms. In this proposal, I limit my scope to $k$-partite algorithms, where a single update operation accesses $k$ different parameters from $k$ different sets of shared parameters. For simplicity of explanation, I present an “Algorithm 1” for $k = 1$ case and “Algorithm 2” for general cases.
Algorithm 3 SchMP Static-schedule Topic Modeling

$U, V$: doc-topic table, word-topic table (model params)
$N, M$: number of docs, vocabulary size
{$z \choose p}, \{w \choose p}$: topic indicators and token words stored at worker $p$
$c$: persistent counter in schedule() Function schedule():

For $p = 1..P$:

// "word-rotation" schedule

$x = (p - 1 + c) \mod P$
$S_p = (xM/P, (x + 1)M/P)$  // $p$’s word range

c = c + 1

Return $[S_1, \ldots, S_P]$ Function update($p, S_p, \{U \choose p}, V, \{w \choose p}, \{z \choose p}$):

[lower,upper] = $S_p$  // Only touch $w_{ij}$ in range

For each token $z_{ij}$ in $\{z \choose p}$:

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 $[R_1, \ldots, R_P]$

Static Scheduling for ML Algorithms with $k = 1$: Algorithm 1 is the static scheduling algorithm for the case of $k = 1$, where each update operation 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 chunks, where $M$ is the number of 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 exclusive access to the $(i + m)\%M$-th partition. 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 $k$-Partite Algorithm: In $k$-partite algorithms, shared parameters consist of $k$ different independent sets $P_0, P_1, .., P_{k-1}$. An update operation accesses $k$ shared parameters, each of which comes from $k$ sets. From the machine learning perspective, the sets of different parameters represent different types of objects. In Algorithm 2, each parameter set is partitioned into $M$ disjoint chunks, and one iteration is divided into $M^k$ subiterations. Algorithm 2 runs $k$ nested loops to perform scheduling. Every $M$ subiteration, a for-loop statement at level $k - 1$ runs a full pass and increments the index at level $k - 2$. After $M^k$ subiterations, the algorithm finishes one full pass of a for-statement at level-0, which is equal to the amount of work in one iteration.

3.1.2 LDA/SGD-MF with Static Scheduling

I present two static scheduling examples: LDA using Algorithm 1 and SGD-MF using Algorithm 2 with $k=2$. SGD-MF could be scheduled using Algorithm 1, and it could achieve higher performance than SGD-MF using $k=2$ Algorithm 2 because it has less synchronization overhead. However, I intentionally scheduled SGD-MF using Algorithm 2 with $k=2$ as a showcase of Algorithm 2.

LDA scheduling example: For simplicity, I will not show the details of update() and aggregate(). Instead, I focus on how schedule() controls which token assignments $z_{ij}$ are being updated.
Algorithm 4 SchMP Static-schedule SGD-MF

$M$: the number of worker nodes
$W, H$: user/product summary vector table (model params)
$W_0, W_1, \ldots, W_{M-1}$: partitions of W table
$H_0, H_1, \ldots, H_{M-1}$: partitions of H table
${\{R\}}_p$: a set of ratings at worker $p$ (randomly partitioned)
$r_{ij}$: a single rating by user $i$ in $W$ for a product $j$ in $H$
$l_0, l_1$: persistent counters in schedule()

Function $\text{schedule}()$:

For $p = 1..M$: // “W rotation at inner loop”

\[
S_p \leftarrow \{\}
\]
\[
x = (p - 1 + l_1) \mod M
\]
\[
S_p \leftarrow W_x // \text{put one partition of W}
\]
\[
l_1 = l_1 + 1
\]
\[
l_0 = l_1 / M
\]

For $p = 1..M$: // “H rotation at outer loop” schedule

\[
x = (p - 1 + l_0) \mod M
\]
\[
S_p \leftarrow H_x // \text{put one partition of H}
\]

Return $[S_1, \ldots, S_P]$

Function $\text{update}(p, S_p, {\{R\}}_p)$:

\[
[i\text{-lower}, i\text{-upper}, j\text{-lower}, j\text{-upper}] = S_p \quad // \text{Only touch } r_{ij} \text{ in the range}
\]

For each rating $r_{ij}$ in $\{R\}_p$:

\[
\text{If } r_{ij} \in \text{range}(i\text{-lower}, i\text{-upper}, j\text{-lower}, j\text{-upper})
\]
\[
\text{new}W_i, \text{new}H_j = \text{ProcessOneRate}(r_{ij}, W[i], H[j])
\]
\[
W[i] = \text{new}W_i
\]
\[
H[j] = \text{new}H_j
\]

Return $R_p$

by which worker. Algorithm 3 shows SchMP LDA implementation, which uses a static “word-
rotation” schedule. It rotates a shared word table using Algorithm 1 and partitions the documents over workers. The word-rotation schedule partitions the rows of $V$ (word-topic table), so that workers never touch the same rows in $V$ (each worker skips over words $w_{ij}$ associated with not-assigned rows). The partitioning is “rotated” $P$ times, so that every word $w_{ij}$ in each worker is touched exactly once after $P$ invocations of $\text{schedule}()$.

One might ask why $\text{schedule}()$ is useful, being that a common strategy is to have workers sweep over all their $z_{ij}$ every iteration [1]. However, as I exhibited in Sec 2.1.2, this causes concurrent writes to the same rows in $V$, breaking model dependencies.

SGD-MF example: When one does not pre-partition the rating data by user-id or product-id, one update operation of SGD-MF will touch two shared parameters, one from $W$ and one from $H$. Therefore, we can consider this SGD-MF algorithm as a $k$-partite algorithm with $k = 2$. In this section, I will present a static schedule plan for this non-prepartitioned SGD-MF using Algorithm 2. One should assume that the ratings of training data are randomly partitioned so that the ranges of user-id and product-id of ratings in a worker node are not constrained. Each $W$ and $H$ parameter of SGD-MF is partitioned into $M$ chunks $\{W_0, W_1, \ldots, W_{M-1}\}$ and $\{H_0, H_1, \ldots, H_{M-1}\}$, respectively. The loop at level-0 runs over $W$, and the loop at level-1 runs over $H$. At each subiteration, a worker is given exclusive access to a pair of $W_i$ and $H_j$. After $M^2$ subiterations, each node goes through all possible pairs of $W$ and $H$ partitions. Again, this static scheduling avoids concurrent write
problems completely, as it did in the previous LDA example.

3.1.3 System Design Challenges

In the static scheduling schemes of Algorithms 1 and 2, I observed that the schemes share a common network communication pattern that circulates the shared parameters along a ring of worker nodes. In this section, I identify the common communication pattern that we can exploit to improve system throughput of statically scheduled ML algorithms and present three other system throughput challenges.

Ring communication pattern: Static scheduling schemes in Algorithms 1 and 2 rotate shared parameter partitions along a ring of worker nodes. For example, Algorithm 1 starts with $M$ parameter partitions, each of which is assigned to a worker. In each subiteration, the scheduler shifts mapping of a worker and a parameter partition by one click in the clock-wise direction of the ring.

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

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 ~ 64 cores), instead of making a worker node per a core. However, a worker node with many cores is more likely to suffer from high lock contention and high cache miss ratio. I will address these problems by grouping update operations according to a shared model parameter and sequentially executing a group of update operations 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 is dedicated to a thread and executed sequentially. However, that grouping technology might cause a load balancing problem among threads within a node. In ML applications, the distribution of workload per model parameter is uneven. For example, the word distribution of LDA is highly skewed, meaning that updating some model parameters associated with frequent words takes much longer than others. Therefore, certain threads with particularly heavy jobs will cause a straggler problem. I will address this problem using a heavy-job prioritization technique, which prioritizes heavy jobs when selecting a task to execute and communication in a worker node.

3.2 STRADS-Static Engine

Figure 6 shows the overall architecture of a static engine. The input training data is partitioned over worker nodes, and shared parameters are stored in a distributed KV store. Each worker
node executes parameter update operations, accessing the KV store while simultaneously being in charge of a partition of key range as a KV server.

3.2.1 Ring overlay-network for addressing synchronization overhead

To improve iteration throughput, I exploit the ring-based data movement pattern of static scheduling. I implement a ring overlay network on worker nodes and let worker nodes rotate parameters along the ring network at fine granularity. When running the inner-most 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. 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 the \( k = 1 \) algorithm, the synchronization count per iteration is reduced from \( M \) to 1. In the \( k=2 \) bipartite algorithm, the synchronization count is reduced from \( M^2 \) to \( M \).

3.2.2 Job Pool for multi-threading worker node

A worker node consists of multiple update threads and a parameter manager thread. The parameter manager maintains two job pools, a ready pool and a done pool. Upon receiving a parameter \( p_i \) from a ring neighbor, the parameter manager creates a task with update operations that can run when a parameter \( p_i \) arrives and puts the task in the ready pool. In run time, update threads poll and pull available tasks from the ready pool. On obtaining a task, an update thread sequentially executes all update operation of the task. Then, the completed task is put into the done pool and finally \( p'_i \) is passed onto the next ring neighbor by the parameter manager.
Table 1: Data sets used in the evaluation.

<table>
<thead>
<tr>
<th>ML app</th>
<th>Data set</th>
<th>Workload</th>
<th>Feature</th>
<th>Raw size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDA</td>
<td>PubMed</td>
<td>737M tokens</td>
<td>8.2M documents, 141K words, 1K topics</td>
<td>4.5GB</td>
</tr>
<tr>
<td>LDA</td>
<td>ClueWeb</td>
<td>10B tokens</td>
<td>50M webpages, 2M words, 1K topics</td>
<td>80 GB</td>
</tr>
<tr>
<td>MF</td>
<td>Netflix</td>
<td>100M ratings</td>
<td>480K users, 17K movies (rank=40)</td>
<td>2.2 GB</td>
</tr>
<tr>
<td>MF</td>
<td>x256 Netflix</td>
<td>25B ratings</td>
<td>7.6M users, 272K movies (rank=40)</td>
<td>563 GB</td>
</tr>
</tbody>
</table>

3.2.3 Prioritization in task dispatching

Due to the skewed workload distribution of ML, tasks with popular words or products are far heavier than others, and thus take much longer to process. Because tasks in the ready and done pools are served in FIFO (First In, First Out), these heavy tasks will cause a load balancing problem, meaning that a few threads with these heavy tasks will bottleneck iterations. To mitigate this problem, the parameter manager prioritizes heavy tasks when placing a new task on the ready pool and pulling done tasks from the done pool. In other words, it sorts tasks in the ready/done pool based on their workload.

3.3 Selected Results

I compare SchMP ML programs implemented on STRADS against existing parallel execution schemes — either a well-known publicly-available implementation, or if unavailable, my own implementation — as well as sequential execution. I 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 ML program completion times (i.e. fewer seconds to convergence).

Cluster setup and datasets: Unless otherwise stated, I 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. I use several real and synthetic datasets (see Table 1 for details).

Performance metrics: I compare ML implementations using three metrics: (1) objective function value versus total data samples operated upon\(^5\), 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).

\(^5\)ML algorithms operate upon the same data point many times. The total data samples operated upon exceeds \(N\), the number of data samples.
Figure 7: 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.

Table 2: Static SchMP: DvT for topic modeling (million tokens processed per second).

<table>
<thead>
<tr>
<th>Data set(size)</th>
<th>#machines</th>
<th>YahooLDA</th>
<th>SchMP-LDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClueWeb(80GB)</td>
<td>25</td>
<td>39.7</td>
<td>58.3</td>
</tr>
<tr>
<td>ClueWeb(80GB)</td>
<td>50</td>
<td>78</td>
<td>114</td>
</tr>
<tr>
<td>ClueWeb(80GB)</td>
<td>100</td>
<td>151</td>
<td>204</td>
</tr>
</tbody>
</table>

otherwise stated. Similarly, DvT measures ML iteration throughput across comparable implementations. Note that high OvD and DvT imply good (i.e. small) ML convergence time. Measuring OvD or DvT alone (as is sometimes done) is insufficient to show that an algorithm converges quickly.

ML programs and baselines: I evaluate the performance of LDA (a.k.a. topic model) and MF (a.k.a collaborative filtering). STRADS uses Algorithm 3 (SchMP-LDA) for LDA, and a scheduled version of the Stochastic Gradient Descent (SGD) algorithm$^6$ for MF (SchMP-MF). For baselines, I used YahooLDA, and BSP-MF – my 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 I 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$^7$ 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, I consider each word token as one data sample.

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 7a, 7b). For MF, Figure 7e

$^6$Due to space limits, I could not provide a full Algorithm figure. SchMP-MF divides up the input data such that different workers never update the same parameters in the same iteration.

$^7$With overhead less than 1% of total running time.
Figure 8: Static SchMP: convergence times. (a-b) SchMP-LDA vs YahooLDA; (c-d) SchMP-MF with varying number of machines $m$.

shows that BSP-MF is sensitive to step size; if BSP-MF employs the ideal step size determined for serial execution, it does not properly converge on $\geq 32$ machines. In contrast, SchMP-MF can safely use the ideal serial step size (Figures 7c,7d) and approaches the same OvD as serial execution within 20 iterations.

STRADS Static Engine has high DvT: For LDA, Table 2 shows that SchMP-LDA enjoys higher DvT than YahooLDA. I speculate that YahooLDA’s lower DvT is primarily due to lock contention on shared data structures between application and parameter server 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 8a, 8b). SchMP-MF also exhibits good scalability (Figure 8c, 8d).

4 Part III: Dynamic-SchMP Challenges & STRADS-Dynamic Engine

In this study, I explore a dynamic scheduling scheme on the ML side. My goal is to improve the statistical progress of machine learning algorithms that have a dependency structure among update operations. I also present the STRADS-Dynamic engine to improve the system progress of dynamically scheduled ML programs on the system side. This study:

- Explores a generic two-phase scheduling algorithm that considers the update operation dependency and uneven convergence speed of model parameters.

- Designs a system engine that (1) parallelizes the scheduling function to scale out for achieving high schedule throughput and (2) pipelines scheduling, network communication, and update computation to hide latencies of communication and scheduling. The combination of these two optimizations substantially improves system throughput.

- Quantifies the benefits of dynamic scheduling and using the system engine via three different metrics. My evaluation shows that machine learning programs with dynamic scheduling on the STRADS-Dynamic engine reduce training speed by an order of magnitude.

---

*A required tuning parameter for SGD MF implementations; higher step sizes lead to faster convergence, but step sizes that are too large can cause algorithm divergence/failure.*
4.1 Dynamic Scheduling and Challenges

This section introduces the dynamic scheduling scheme, which reflects run-time changes of parameter priorities and generates schedule plans during runtime. Note that dynamic scheduling is a general scheduling scheme that can support static ML algorithms with non-variadic update function, such as in the previous chapter, as well as ML algorithms with variadic update function.

4.1.1 Dynamic Scheduling

The dynamic scheduling scheme implements two strategies of SchMP introduced in Sec 2.2.1: dynamic partitioning and prioritization. In this scheme, the scheduler selects \( L \) parameters, designated as \( S_L \), by conducting sampling based on the distribution of individual parameters’ convergence distance. Because the parameter with longer convergence distances are assigned higher probability during sampling, updating the selected \( L \) parameters yields the greatest convergence progress. I approximate the convergence distance of a parameter \( p_i \) using \( \Delta p_i \) of the latest update operation. Then, the scheduling scheme builds a dependency graph by performing pair-wise dependency measurements between \( L \) parameters in \( S_L \). When the dependency of two parameters \( p_i \) and \( p_j \) in \( S_L \) is larger than the prefixed threshold, the scheduler evicts either \( p_i \) or \( p_j \) from \( S_L \) as a victim. The victim is reserved for the following iteration.

The dynamic scheduling scheme reduces the complexity of scheduling per iteration to \( O(L^2 + L) \), where \( L \) is far less than the total number of parameters \( M \) and improves progress per iteration. However, it may suffer from low iteration throughput in a distributed environment. Because only a small number of parameters are updated in each iteration, the time spent updating \( L \) parameters with the \( P \) workers may not be able to amortize the latency of network communication and the cost of scheduling computation. I will address this system design problem by exploiting Staleness Input to Update and Scheduling idea as discussed in Sec 2.2.1.

4.1.2 Lasso with Dynamic Scheduling

Lasso, or the \( \ell_1 \)-regularized least-squares regression, is used to identify a small set of important features from high-dimensional data. It is an optimization problem:

\[
\min_{\beta} \frac{1}{2} \sum_{i=1}^{n} (y^i - x_i^T \beta)^2 + \lambda \|\beta\|_1
\]

where \( \|\beta\|_1 = \sum_{a=1}^{d} |\beta_a| \) is a sparsity-inducing \( \ell_1 \)-regularizer, and \( \lambda \) is a tuning parameter that controls the sparsity level of \( \beta \). \( X \) is an \( N \)-by-\( M \) design matrix (\( x_i^T \) represents the \( i \)-th row, \( x_a \) represents the \( a \)-th column), \( y \) is an \( N \)-by-1 observation vector, and \( \beta \) is the \( M \)-by-1 coefficient vector (the model parameters). The Coordinate Descent (CD) algorithm is used to solve Eq. (7), and thus learn \( \beta \) from the inputs \( X, y \); the CD update rule for \( \beta_a \) is:

\[
\beta_a^{(t)} \leftarrow S(x_a^T y - \sum_{b \neq a} x_a^T x_b \beta_b^{(t-1)}, \lambda),
\]

where \( S(\cdot, \lambda) \) is a soft-thresholding operator [18].

Algorithm 5 shows an SchMP Lasso that uses dynamic, prioritized scheduling. It expects that each machine locally stores a subset of data samples (which is common practice in parallel ML).
Algorithm 5 SchMP Dynamic, Prioritized Lasso

\(X, y\): input data
\(\{X\}^p, \{y\}^p\): rows/samples of \(X, y\) stored at worker \(p\)
\(\beta\): model parameters (regression coefficients)
\(\lambda\): \(\ell_1\) regularization penalty
\(\tau\): \(G\) edges whose weight is below \(\tau\) are ignored

**Function** schedule(\(\beta, X\)):
- Pick \(L > P\) params in \(\beta\) with probability \(\propto (\Delta \beta_a)^2\)
- Build dependency graph \(G\) over \(L\) chosen params:
  - edge weight of \((\beta_a, \beta_b) = \text{correlation}(x^a, x^b)\)
- \([\beta_{G_1}, \ldots, \beta_{G_K}] = \text{findIndepNodeSet}(G, \tau)\)
- For \(p = 1..P\):
  - \(S_p = [\beta_{G_1}, \ldots, \beta_{G_K}]\)
- Return \([S_1, \ldots, S_P]\)

**Function** update(\(p, S_p, \{X\}^p, \{y\}^p, \beta\)):
- For each param \(\beta_a\) in \(S_p\), each row \(i\) in \(\{X\}^p\):
  - \(R_p[a] += x^i_a y^i - \sum_{b \neq a} x^i_a x^i_b \beta_b\)
- Return \(R_p\)

**Function** aggregate(\([R_1, \ldots, R_P]\), \(S_1, \beta\)):
- For each parameter \(\beta_a\) in \(S_1\):
  - \(\text{temp} = \sum_{p=1}^P R_p[a]\)
  - \(\beta_a = S(\text{temp}, \lambda)\)

However, the Lasso update Eq. (8) 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 \(\{X\}^p, \{y\}^p\). Note that update() and aggregate() are a straightforward implementation of Eq. (8).

schedule() picks (i.e. prioritizes) \(L\) parameters in \(\beta\) with probability proportional to their squared difference from the latest update (their “delta”). Parameters with larger 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\(^9\) between data columns \(x^a, x^b\). Finally, it removes all edges in \(G\) below a threshold \(\tau > 0\), and extracts nodes \(\beta_{G_k}\) that do not have common edges. All chosen \(\beta_{G_k}\) are thus pairwise independent and safe to update in parallel.

### 4.1.3 System Design Challenges

This section presents two major system design challenges for implementing dynamic scheduling in a distributed environment. The first challenge is the low scheduling throughput caused by high computational complexity. The time complexity of dynamic scheduling is \(O(L + L^2)\), which ultimately limits scheduling throughput. The second challenge is the latencies of network communication and scheduling. In many dynamic scheduling algorithms, latency of parameter update operation is short. Therefore computation time by worker is often insufficient to amortize latencies of network communication and scheduling. These challenges make it hard to achieve high system throughput of dynamic scheduling algorithms in a distributed environment although dynamic

---

\(^9\)On large data, it suffices to estimate the correlation with a data subsample.
scheduling improves statistical progress on the ML side.

4.2 STRADS-Dynamic Engine

To achieve high update throughput of ML programs with dynamic scheduling, I explore three optimization techniques that trade “progress per update” from scheduling. The logic behind this trade is to maximize progress per unit time by balancing update throughput and progress per update.

4.2.1 Distributed Scheduler

To address the low scheduling throughput challenge, the STRADS-dynamic engine implements a distributed scheduler, in which the total set of update operations is partitioned into \( N \) disjoint subsets, where \( N \) is the number of scheduler instances. Each scheduler instance runs independently on a subset of update operations. A distributed scheduler with \( N \) instances easily increases scheduling throughput by \( N \) 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 \( N \) scheduler instances in a round robin manner.

4.2.2 Pipelining

To address the aforementioned latency challenge, the dynamic engine implements pipelining that overlaps network communication, update computation, and scheduling computation. At iteration \( t \), the dynamic scheduler will start the iterations \( t + 1, t + 2, \ldots, t + s \) before the current iteration \( t \) is completed. The pipeline depth \( s \) can be tuned by user. The pipelining technique improves update throughput. However, it could lower progress per update because (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 due to the fact that scheduling plans for iterations \( t, t - 1, \ldots, t - s \) came from \( s \) different subsets of update operations managed by \( s \) independent scheduler instances. Thus, there might be dependencies between iterations. This does not crash the whole computation of the machine learning algorithm, but it does reduce statistical progress per update.

4.2.3 Task reordering

The pipelining technique can cause a loss of statistical progress depending on two factors: (1) the strength of dependencies among in-flight update operations in the pipeline and (2) the magnitude of parameter changes by in-flight update operations in the pipeline. Here, I focus on reducing the second factor, which represents data inconsistency associated with parameters of in-flight update operations. For schedule set \( S_i \) with \( k \) update operations \( op_0, op_1, \ldots, op_{k-1} \), the coordinator reorders the operations in ascending order of priority of operations and divides \( k \) update operations into \( M \) subiterations. Then, the coordinator starts to pipeline the subiterations. The most important update operations are placed in the first subiteration \( S_{i,0} \) of the iteration \( S_i \) from the \( i \)-th scheduler. Thus, task reordering guarantees that the most important update operations in the following schedule set
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 3: Experiment data sets

![Graphs](a) AD data (b) Synthetic data (c) Synthetic data

Figure 9: Dynamic SchMP: OvD. (a) SchMP-Lasso vs Shotgun-Lasso [9] on one machine (64 cores); (b) SchMP-Lasso vs Shotgun-Lasso on 8 machines; (c) SchMP-SLR vs Shotgun-SLR on 8 machines. \(m\) denotes number of machines.

\(S_{i+1}\) from the \((i + 1)\)-th scheduler can always see the results of \(S_{i,0}\) when the pipeline depth \(s\) is less than \(M\). Therefore, parameter inconsistency due to pipelining can be reduced.

### 4.3 Selected Results

**Cluster setup and datasets:** Unless otherwise stated, I used 8 nodes, each with 16 physical cores and 128GB memory. The nodes are connected by 1Gbps Ethernet. I use one real dataset and several synthetic datasets — see Table 3 for details.

**Performance metrics:** I compare ML implementations using three metrics: (1) objective function value versus total data samples operated upon\(^{10}\), 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.

**ML programs and baselines:** I evaluate \(\ell_1\)-regularized linear regression (Lasso) and \(\ell_1\)-regularized Logistic regression (sparse LR, or SLR). STRADS uses Algorithm 5 (SchMP-Lasso) for the former, and I solve the latter using a minor modification to SchMP-Lasso\(^{11}\) (called SchMP-SLR). To the best of my knowledge, there are no open-source distributed Lasso/SLR baselines that use coordinate descent, so I implement the Shotgun Lasso/SLR algorithm [9] (Shotgun-Lasso, Shotgun-SLR), which uses random model-parallel scheduling\(^{12}\).

**Dynamic SchMP has high OvD:** Dynamic SchMP achieves high OvD in both single-machine (Figure 9a) and distributed 8-machine (Figure 9b) configurations. Here, I compare SchMP-Lasso

---

\(^{10}\)ML algorithms operate upon the same data point many times. The total number of data samples operated upon exceeds \(N\), the number of data samples.

\(^{11}\)Lasso and SLR are solved via the coordinate descent algorithm, hence SchMP-Lasso and SchMP-SLR only differ slightly in their update equations. I use coordinate descent rather than gradient descent because it has no step size tuning and more stable convergence [36, 35].

\(^{12}\)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.
Table 4: **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.

<table>
<thead>
<tr>
<th>Application</th>
<th>1K</th>
<th>10K</th>
<th>20K</th>
</tr>
</thead>
<tbody>
<tr>
<td>SchMP-Lasso 16 × 4 cores</td>
<td>125</td>
<td>212</td>
<td>202</td>
</tr>
<tr>
<td>SchMP-Lasso 16 × 8 cores</td>
<td>162</td>
<td>306</td>
<td>344</td>
</tr>
<tr>
<td>SchMP-LR 16 × 4 cores</td>
<td>75</td>
<td>98</td>
<td>103</td>
</tr>
<tr>
<td>SchMP-LR 16 × 8 cores</td>
<td>106</td>
<td>183</td>
<td>193</td>
</tr>
</tbody>
</table>

Figure 10: **Dynamic SchMP**: convergence time. Subfigures (a-d) correspond to Figure 9.

against random model-parallel Lasso (Shotgun-Lasso) [9]. In either case, Dynamic SchMP decreases the data samples required for convergence by an order of magnitude. Similar observations hold for distributed SchMP-SLR versus Shotgun-SLR (Figure 9c).

**STRADS Dynamic Engine DvT analysis**: Table 4 shows how the STRADS Dynamic Engine’s DvT scales with increasing machines. I observe that DvT is limited by dataset density — if there are more nonzeros per feature column, I 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 10a). When compared with Figure 9a, 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 10b Lasso, Figure 10c 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. I 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 Proposed Work: Automatic Parallelization for STRADS

This chapter presents a proposed work that aims to improve the usability of STRADS by providing a template library and a STRADS-AP infrastructure that automates parameter scheduling.

5.1 Problems and Motivation

In previous chapters, I presented STRADS model-parallel ML programs to improve training speed by an order of magnitude. However, these performance gains are at the cost of extra programming complexities when writing ML schedules. Unfortunately, these programming complexities (i.e. identifying a dependency structure and finding non-conflicting update operations for concurrent execution) can be too much of a burden for ML programmers, who usually write serial ML programs mainly for testing new ML models and algorithms on a single machine. These ML programmers will not be familiar with parallel programming nor the internal design of STRADS. In addition to the burden of scheduling, partitioning input data might also be a non-trivial (yet mundane) task for ML programmers and might lower ML programmers’ productivity.

To improve STRADS’s usability and ML programmers’ productivity, I propose STRADS-AP (STRADS Automatic Parallelization) template library that consists of 1) Distributed Data Structures (DDS), 2) template-based data processing functions that operate on DDS in parallel, and 3) a template-based Parallel_For function that trains model parameters. In STRADS-AP, ML programmers store the training data and model parameters on DDS, specify data processing functions to process data in DDS, and use Parallel_For function to specify model parameter update computation that might have a dependency structure. Then, STRADS-AP infrastructure (runtime and compile-time systems) is responsible for partitioning data, scheduling model parameter update computation for consistency, and managing communication between nodes. This allows ML programmers with little experience in model parameter scheduling and distributed systems to easily get the benefits of a scheduled model-parallel approach and to utilize the resources of a cluster well.

To demonstrate that the API of STRADS-AP library is expressive enough to support many MLDM applications, I will evaluate the coverage of the library API. For the coverage evaluation, I conduct two sets of experiments: (1) a depth experiment and (2) a breadth experiment. In the depth experiment, I select matrix factorization model, a very popular model for building recommendation systems, and apply STRADS-AP to three popular algorithms to solve this model: Alternating Least Square (ALS)[23], Coordinate Descent (CD)[43], and Stochastic Gradient Descent (SGD)[26]. These three algorithms adopt different optimization approaches that require different parallelization strategies when being hand-coded. In the breadth test, I select and apply

---

13Professionals in the MLDM field can be categorized into three groups: (1) ML users who utilize pre-made ML libraries to solve domain specific problems; (2) ML programmers who invent new ML algorithms and write serial programs; and (3) ML system programmers who have parallel programming experience and reimplement ML programs for scaling out in a distributed system.

14These are distributed implementations of container classes, such as map and vector. Elements of DDS are distributed across a cluster. DDS supports element access by index and allows any node of the cluster to access an element on a remote node.
for (i = 0; i < M; i++) { // outer loop
    for (j = 0; j < N; j++) { // inner loop
        loop body
        - read a part of input data D
        - read variables V, and indices i, j
        - read/write a part of parameters P
    }
}
}

change control variables V
if (check_stop_condition)
    break;

Figure 11: The structure of MLDM programs expressed in STRADS-AP’s programming API: is divided into two parts: (1) a pre-training part that defines data structures to store input data, model parameters, and hyper parameters; and (2) a training part with a nested loop structure that repeats a set of parameter update operations and a stopping condition check operation that defines convergence.

STRADS-AP to five different serial ML codes that solve five different ML models: Latent Dirichlet Allocation (LDA)[7], Mixed Membership Stochastic Blockmodels (MMSB)[2], Support Vector Machine (SVM)[12, 22], Lasso[40, 9], and Logistic Regression (LR)[38].

To demonstrate that STRADS-AP programs can achieve comparable performance to SchMP applications having manually-written schedule routines and other distributed ML implementations, I will evaluate performance of STRADS-AP programs. For the performance evaluation, I set up two benchmarks: (1) a scalability benchmark and (2) a speed benchmark. In the scalability benchmark, I analyze training speed improvements of STRADS-AP codes of LDA and SVM, each with 25, 50, and 100 nodes. In the speed benchmark, I compare the performance of STRADS-AP LDA topic modeling and ALS matrix factorization against parallel implementations from three sources: (1) our own library that implements these algorithms as standalone applications, (2) Spark MLlib, and (3) GraphLab.

5.2 Proposed Approach

This section presents the program structure of target MLDM applications and the STRADS-AP’s programming API.

5.2.1 Program structure of target MLDM applications

Figure 11 describes a structure of MLDM applications that are expressible through the API of STRADS-AP (Table 6), and Table 5 shows a list of notations related to Figure 11. I divide the structure of ML applications into (1) a pre-training part and (2) a training part. The pre-training part loads input data, defines data structures D to store input data, and runs data-processing functions on D. Then, the pre-training part defines and initializes data structures P to store model...
Notation | Description
--- | ---
\(D\) | immutable data structures to store input data samples
\(P\) | mutable data structures to store model parameter values
\(V\) | hyper parameters to control training (i.e. learning rate)
i, j | index variables of the outer and inner loops, respectively
\(\hat{f}\) | the loop body of the inner loop
\(f_{i,j}\) | the execution of \(\hat{f}\) with index variables \(i, j\) for the outer and inner loops.
i-th iteration | the completion of the inner loop which executes \(f_{0,i}, f_{1,i}, ..., f_{N-1,i}\) once with the outer loop index \(i\).
\(r_{i,j}\) | an address set of read accesses on \(D, P\) by \(f_{i,j}\)
\(w_{i,j}\) | an address set of write accesses on \(P\) by \(f_{i,j}\)
\(rw_{i,j}\) | a union of \(r_{i,j}\) and \(w_{i,j}\)

Table 5: Notations related to Figure 11.

parameters and control variables \(V\) to store hyper parameters. Usually, computations in the pre-training part can be independently parallelized and are executed one time before the parameter training starts. The **training part** consists of a nested loop structure that iteratively trains model parameters. Three components of the training part include:

**Outer Loop:** Each pass of the outer loop triggers the inner loop, which runs to completion until the outer loop finishes \(M\) iterations or the stopping condition is satisfied. The body of the outer loop consists of two parts: (1) an inner loop statement and (2) a routine that changes control variables and checks the stopping condition.

**Inner Loop:** When the inner loop runs, it executes \(f\), the inner loop body, \(N\) times with the loop index \(j = 0, 1, ..., (N-1)\), which is equal to the amount of work per one iteration in MLDM parlance. \(f_{i,j}\) denote an execution of the loop body \(f\) with the outer loop index \(i\) and the inner loop index \(j\). \(f_{i,j}\) updates the values of a set of model parameters by reading a set of input data, a set of model parameters, and control variables. The memory addresses for these read and write operations on input data \(D\) and model parameters \(P\) by \(f_{i,j}\) are denoted as read set \(r_{i,j}\) and write set \(w_{i,j}\), respectively, and \(rw_{i,j}\) is the union of \(r_{i,j}\) and \(w_{i,j}\).

**Stopping condition checking routine:** To determine whether to continue or not, an ML program measures the quality of trained parameter values regularly (i.e. usually every iteration). Quality is commonly represented by an objective value or likelihood value. After checking the stopping condition, an ML program usually tunes the hyper parameters such as learning rate or regularization coefficient.

The program structure shown in Figure 11 is commonly found across many real MLDM applications. To support this argument, I will present an example in the next section and show that the applications written in GraphLab and Parameter Server programming models can be expressed in the structure of Figure 11.
### Distributed Data Structures (DDS)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dvector&lt;T&gt;</td>
<td>a distributed vector of T type elements</td>
</tr>
<tr>
<td>dmap&lt;K,V&gt;</td>
<td>a distributed map of &lt;K,V&gt; pairs</td>
</tr>
<tr>
<td>dmultimap&lt;K,V&gt;</td>
<td>a distributed multimap of &lt;K,V&gt; pairs</td>
</tr>
<tr>
<td>dist&lt;T&gt;</td>
<td>a distributed list of T type elements</td>
</tr>
</tbody>
</table>

### Data Processing Functions

<table>
<thead>
<tr>
<th>Function Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS&lt;T&gt;&amp; load(F parser, string fn)</td>
<td>reads lines from a file and applies the parser function to each line.</td>
</tr>
<tr>
<td>DDS&lt;T2&gt;&amp; map(DDS&lt;T1&gt;&amp; input, F udf)</td>
<td>applies udf to elements of the input and generates a new T2 type obj.</td>
</tr>
<tr>
<td>T2&amp; reduce(DDS&lt;T1&gt;&amp; input, F udf)</td>
<td>combines the elements of the input into a return value of T2 type obj.</td>
</tr>
<tr>
<td>void transform(DDS&lt;T&gt;&amp; input, F udf)</td>
<td>applies udf to elements of the input and modifies contents of the input</td>
</tr>
</tbody>
</table>

### Parallel Loop

- **Parallel for**
  - `[closure](int j){user code}` is a lambda function f(int j).
  - `Parallel_for iterates the lambda function f(j) with j = I'[0] to I'[N-1] where I' = a permutation of I sequence [0, 1, ..., (N-1)].`

---

**Table 6:** Programming API of STRADS-AP. T represent a type of a data structure, and F represent a type of a function. Regarding Parallel_for, the lambda function f is a unnamed function object with a function call method defined as `void operator()(int i){user code}` and private members that correspond to the closure.

#### 5.2.2 STRADS-AP Programming API

STRADS-AP provides a template library that allows ML programmers to express ML applications that follow the structure of Figure 11. The template library provides three types of programming constructs:

1. **A set of distributed data structures (DDS):** includes distributed implementations of vector, map, multimap, and list. DDS store a collection of elements. By default, the elements of DDS are partitioned across a set of machines. In the example of N machines, the elements of a vector DDS are partitioned into N chunks of consecutive index range, and the elements of a map DDS are partitioned by hashing on the key that determines the location of a pair <Key,Value> by `HashFunction(Key) mod N`. Each type of DDS provides a set of API such as add, delete, size, subscript operator, and iterator. ML programmers should store input data D and model parameters P in DDS.

2. **A set of data processing functions:** create DDS and process data in DDS across a set of machines. The set of functions includes load, store, create, sort, sum, transform, max, min, map, and reduce functions. The common header of data processing functions is `RT& function(T &ds, F UDF, Args..)` where T represents a type of DDS, F represents the type of user defined function, and Args represents extra parameters, if necessary. A data processing function applies the user-defined function (udf) to each element of the input DDS. When the user-defined function is applied to an element, it generates a new element or modifies the element in place. RT is the return type of the function and can be a DDS type, primitive type, or user-defined data structure type. Note that the data processing functions can be easily parallelized because the data processing functions cause read-only accesses on an input DDS and generate elements of a new DDS independently (or modify elements of the input DDS independently). These functions are
Declare and initialize input data structures $D$ by loading input data file
Declare and initialize parameter structures $P$
Declare and initialize control variables $V$ to control training (i.e. learning rate)

for $i = 0; i< M; i++$ { // outer loop
  modify control variables $V$
  if (check_stop_condition) break;
}

- read/write a part of parameters $P$
}

for (i = 0; i< M; i++ ) { // outer loop
  stepsize= sizesize*(0.9);
  float objective = RMSE( D,W,H); // see appendix for detail
  row &h = H[D[j].movieid];
  float r = D[j].rate;
  gradient_update (w,h,r);
}

(a) a program structure with Parallel_For

(b) SGD-MF example

Figure 12: Programming Model Example: (a) the inner loop of Figure 11 can be easily replaced with Parallel_For(); (b) SGD-MF (Stochastic Gradient Descent Matrix Factorization) is expressed using STRADS-AP programming API.

used mainly for the pre-training part and stopping condition checking\(^{15}\).

(3) Parallel_For: Parallel_for(int $N$, F $f$) is a template function that have two arguments: (1) an integer as a max iteration count and (2) a lambda function[24] (defined as [closure](int $j$){user defined code}) as a loop body. The Parallel_For function iterates the lambda function $f$ (int $j$)\{ . . . \} for $j = I'[0]$ to $I'[N-1]$ where $I'$ = a permutation of $I$ sequence \([0, 1, ..., (N-1)]\). The closure is a list of variables that are available from the point that the Parallel_For is called. ML programmers should use Parallel_For function to write the inner loop of Figure 11. This can be done straightforwardly as shown in Figure 12a. As an example, I present SGD-MF\(^{16}\) application using STRADS-AP’s API in Figure 12b. To facilitate parallel execution of Parallel_For, STRADS-AP has a constraint regarding the lambda function argument that all closure variables except DDS must be immutable, meaning that STRADS-AP infrastructure does not need to consider non-DDS closure variables for consistency during scheduling. The details of checking this constraint depend on the lambda function syntax of a programming language. As an example, I will present the implementation details of this constraint checking for C++ language in Section 5.3.2.

For an invocation of Parallel_For(N, $F$ fn(int $j$)), STRADS-AP runtime system concurrently executes $N$ lambda function instances, $fn(0)$, $fn(1)$, ..., $fn(N−1)$, across a cluster while ensuring isolated execution of the lambda function instances. Details of isolated execution will be discussed

\(^{15}\)Often, the calculation of the objective value to check stopping condition is as expensive as training and must be parallelized. Fortunately, the objective value calculation can be independently parallelized because it causes only read accesses on input data and model parameters. Therefore, the routine of objective value calculation can be implemented easily using the data processing functions in Table 6.

\(^{16}\)SGD-MF stands for stochastic gradient descent matrix factorization.
5.3 STRADS-AP Infrastructure Implementation

This section presents an implementation of STRADS-AP infrastructure targeted to the distributed computing environment and C++ language. The infrastructure implementation is divided into two parts: a runtime system and a compile-time system. At the core of the runtime system is the virtual iteration technique that automatically identifies dependencies of MLDM computations in a Parallel_For function and generates a dependency graph. After a virtual iteration is completed, STRADS-AP starts actual execution of the Parallel_For function. During the actual execution, the runtime system automatically performs scheduling for consistency by running STRADS scheduler (the static-scheduler in Section 3 or dynamic-scheduler in Section 4) based on the dependency graph from the virtual iteration. The compile-time system is a standalone tool that handles C++-specific details of serialization/deserialization of function type arguments.

5.3.1 Runtime System

STRADS-AP runtime consists of a master node, worker nodes, and scheduler nodes as shown in Figure 13.

A **master node**: executes a programmer’s driver program\(^{17}\) that defines DDS and invokes parallel functions, such as a data processing function or a Parallel_For function, across a set of worker nodes.

A **worker node**: runs a worker thread and a DDS server thread. A worker thread is responsible for running a subtask of a parallel function as well as executing a virtual iteration of a Parallel_For function. The virtual iteration will be discussed later. A DDS server stores a partition of each

---

\(^{17}\)A driver program is a source code that an ML programmer has written using the API of STRADS-AP.
distributed data structure and provides global access to DDS clients such as worker threads, a
master node, and scheduler nodes. As a result, DDS clients can access elements of DDS through
the subscript operator $[ ]$ or the iterator method of DDS.

**The scheduler node:** generates a dependency graph based on read/write access records from
worker nodes during virtual iteration, makes scheduling plans, and forwards the plans to worker
nodes. Worker nodes run STRADS-Static engine as described in Section 3 or STRADS-Dynamic
engine as described in Section 4 to execute the Parallel_For according to the scheduling plans.

**Virtual Iteration:** is a one-time execution of a Parallel_For function for system purposes, which is
transparent to ML programmers and does not affect the state of input data, model parameters, and
control variables. In details, each instantiation of a Parallel_For function maintains the invocation
count. On the first invocation, a Parallel_For function activates the recording option for DDS that
are accessed by the lambda function $f$ with $j = I'[0]$ to $I'[N-1]$. The master node assigns each worker thread $k$ a unique range $R_k$ of $j$. Then, each worker thread $k$ executes $f^v_j$ for $v = 0$ and $j \in R_k$ while recording $r^v$ and $w^v$. During the virtual iteration, writes on DDS and control variables are not committed. Hence, we assume that changes in values of DDS and control variables can not impact $rw_i$ of $f_i$ and dependencies. On completion of the virtual iteration, the scheduler gathers the DDS read/write records of $f^v_0, f^v_1, ..., f^v_{(N-1)}$. For the sake of simplicity, I denote $f^v_0, f^v_1, ..., f^v_{(N-1)}$ as parameter update operations. Based on these read/write records, the scheduler generates a dependency graph among parameter update operations and makes scheduling plans that execute only independent update operations in parallel. Two update operations $f^v_a$ and $f^v_b$ are defined to be independent if the intersection of $w^v_a$ and $rw^v_b$, and the intersection of $w^v_b$ and $rw^v_a$ are both empty. The scheduler divides $N$ update operations into $N'$ subsets of independent update operations and executes each subset across a set of workers. Because the overhead of running a virtual iteration, generating a dependency graph, and making scheduling plans is high, STRADS-AP reuses a dependency graph and scheduling plans when the same Parallel_For is invoked in the following iteration of the outer loop, with an assumption that $r^v_j$ and $w^v_j$ of $f^v_j$ for all $j$ do not change for the superscript $v = 0, 1, 2, ..., M$. In other words, this assumption means that the set of read/write accesses to input data and model parameter of each $f_j$ does depend on a data set but does not depend on the values of model parameters and control variables.

**Scheduling:** When a virtual iteration is completed for a Parallel_For, the runtime system starts
actual execution of the Parallel_For function. For the actual execution, a scheduler generates a
schedule plan for consistency based on the dependency graph from the virtual iteration. To perform
scheduling, STRADS-AP reuses static scheduler in Section 3 or dynamic scheduler in Section 4
with minor modifications$^{19}$.

$^{18}$Note that a lambda function represents the loop body of a Parallel_For function as shown in Figure 12a.

$^{19}$To support STRADS-AP, the dynamic scheduler is modified to disable the priority-based sampling scheme and
perform dependency scheduling ensuring that all parameter update operations are scheduled once per invocation of
Parallel_For function.

33 24/04/2017
Figure 14: **Processing user’s code by the compile-time system:** (a) user’s source code; (b) After scanning the user’s source code, the compile-time system generates a function class \( LFC_0 \) against the lambda function \( LF_0 \) and modify \( ST_0 \) into \( ST_0' \)

### 5.3.2 Compile-time System

This section presents C++-specific details of implementation that are necessary to enable parallel execution of C++ template-based data processing functions and Parallel_For function. (Readers who are unfamiliar with C++ may skip this subsection and continue to Section 5.4). C++ lacks reflection capability\(^{20}\). Recently, C++ 11 standards added RTTI (Run-Time Type Identification) feature as reflection, but it is limited to type examination. This limited reflection capability makes it difficult to implement routines to serialize/deserialize function type arguments\(^{21}\) of parallel template functions. Serialization/deserialization of function type argument is necessary for communication between a master and worker nodes when a master dispatch a data processing function or Parallel_For invocation to worker nodes. To address this challenge, I present a compile-time system. It is a standalone tool that parses an user’s source code and automatically generates serialization/deserialization codes for parallel template functions in Table 6. I present details of serialize/deserialize code generation with an example of Parallel_For function. The code generation for Parallel_For consists of two steps:

**Scanning input code:** The compile-time system scans the source code to identify types of declared variables\(^{22}\) and find Parallel_For statement(s). Parallel_For statements in the source code are de-

---

20In programming languages, reflection is the ability to examine and modify its data structure and behavior in runtime.

21A argument of a function type consists of a function body and a closure.

22To facilitate parsing the source code and identifying types of variables, I use GCC-XML and CastXML third party tools that parse C++ code and make an abstract syntax tree in XML.
noted as $ST_k$, and the lambda function of $ST_k$ is denoted as $LF_k$. For each $LF_k$, the compile-time system checks whether or not the closure of $LF_k$ satisfies the constraint that all closure variables except DDS must be immutable. Under C++ lambda function syntax, this constraint can be translated into three specific rules on the closure composition: (1) $[\&]$ and $[=]$ expressions, which capture all automatic variables used in the body of the lambda function, are not allowed; (2) only STRADS-AP’s distributed data structures can be captured by reference; and (3) a mutable specifier is not allowed, meaning that the lambda function is not allowed to change the parameters captured by value\(^\text{23}\). If every $LF_k$ satisfies these three rules on the closure composition, the compile-time system starts the next step. Otherwise, it stops with an error message.

**Generate lambda function class:** The lambda function in C++ is defined as an unnamed class by C++ compiler at compile time, and there is no way to serialize and deserialize an instance of this lambda function class. To address this problem, the compile-time system generates a function class $LFC_k$ for each $LF_k$. This reverse engineering can be done mechanically. The variables in the capture list are copied as class members and the body of the lambda function $LF_k$ is copied as a function call operator(). Then, the compile-time system modifies $ST_k$ of the user source code into $ST'_k$, which replaces $LF_k$ of $ST_k$ with declaration of $LFC_k$. The figure 14 shows an example of this processing with $k = 0$. When a Parallel For $ST_k$ is invoked at runtime, the function creates an instance of $LFC_k$ that stores the closure of $LF_k$. Then, STRADS-AP runtime serializes\(^\text{24}\) this $LFC_k$ instance and sends RPC request with this serialized instance to workers. Note that this code generation and modification to the user code are done by the compile-time system without human intervention.

### 5.4 Support programming model of other frameworks

To facilitate reuse of MLDL application codes from GraphLab [30] and parameter server [13], I will integrate the porting layers that runs GraphLab and PS applications on top of STRADS-AP.

**Porting Layer for Parameter Server applications:** To facilitate porting parameter server applications, STRADS-AT provides a parameter server (PS) programming interface with two extra annotation APIs: BEGIN and END. In a PS application code, ML programmers are required to specify the beginning and the end of an update operation using these APIs. STRADS-AP infrastructure defines an invocation of a pair of BEGIN and END as an update operation and schedules them for consistency in runtime. Additionally, ML programmers are required to ensure that reordering update operations does not affect the correctness of the application and is acceptable. PS applications fall into two classes: (1) a BSP algorithm that updates all parameters at one time (i.e. data-parallel gradient descent) and (2) a “partial at one time” algorithm that updates a subset of parameters at one time (i.e. parallel CGS LDA). Because the first type of algorithm does not have a dependency structure, STRADS-AP targets only the second type of PS applications for porting.

---
\(^{23}\)In the lambda function without the mutable specifier, the passed-by-value closure arguments cannot be modified, but the function is allowed to modify the passed-by-reference closure arguments. This is a part of the C++ lambda function syntax, and violation of this syntactic constraint will cause a compile-time error.

\(^{24}\)The compile-time system adds serialization and deserialization methods to $LFC_k$ using the Serialization and
**Algorithm 6** GraphLab Execution Model (Algorithm 2 in [30])

Input: Initial vertex set \( T = \{v_1, v_1, \ldots, v_n\} \)
Input: Data Graph \( G = (V, E, D) \)

while Until \( T \) is not Empty do
   \( v \leftarrow \text{RemoveNext}(T) \)
   \((T', S_v) \leftarrow f(v, S_v)\)
   \( T \leftarrow T \cup T' \)
end while

**Algorithm 7** GraphLab Execution on STRADS-AP

Input: Initial vertex set \( T = \{v_1, v_1, \ldots, v_n\} \)
Input: temporary vertex set \( T_{tmp} = \phi \)
Input: Data Graph \( G = (V, E, D) \)

Store \( D \) into DSM (KV store and parallel container)
Feed \( G \) and consistency model to scheduler as dependency graph

while (true) do
   \( k = |T| \)
   parallel update \( (i=1; i < k; i++) \{
      \( v \leftarrow \text{RemoveNext}(T) \)
      fill variable set \( R_i \) of \( D \) associated with \( v \) and \( S_v \) with values in DSM
      \((T', S_v) \leftarrow f(v, S_v)\) // \( T' \) is a set of new vertices added by \( f \)
      store \( R_i \) into DSM
      \( T_{tmp} \leftarrow T_{tmp} \cup T' \)
   \} \)
   \( T \leftarrow T_{tmp} \)
   if(\( T = \phi \)) break
end while

**Porting Layer for GraphLab Applications:** Algorithm 6 shows the GraphLab [30] execution model of a vertex program. For vertex \( v \) in \( G \), GraphLab calls the vertex program that takes variables in \( S_v \) (where \( S_v \) is the scope of \( v \)), executes the application update routine, updates the variables in \( S_v \), and adds the vertex to a vertex set for the next iteration, if necessary. GraphLab repeats the running of the vertex program \( f \) for a subset of a vertex set \( T = \{v_1, v_1, \ldots, v_n\} \) until the program converges (i.e. vertex set \( T \) becomes empty). The GraphLab execution model in Algorithm 6 can be translated into STRADS-AP code in Algorithm 7. GraphLab provides the DSM (Distributed Shared Memory) abstraction in the form of a distributed graph. STRADS-AP replaces the distributed graph with distributed data structures (distributed containers). STRADS-AP places the vertex program in the inner loop. When triggering vertex \( v_i \), STRADS-AP materializes variables of \( S_{v_i} \) with values of distributed data structures, runs the vertex program, and updates the distributed data structures with \( S'_{v_i} \). Note that STRADS scheduler ensures that dependent vertices are not executed in parallel. The GAS model of PowerGraph splits a vertex program \( f \) into “gather”, “apply”, and “scatter” functions. It then runs the “gather” and “scatter” functions for a vertex \( v_i \) in multiple machines to maximize parallelism. STRADS-AP is able to support PowerGraph’s GAS model by replacing the call to the function \( f \) in Algorithm 7 with three calls to the “gather”, “apply”, and “scatter” functions of PowerGraph. STRADS-AP targets asynchronous execution model of GraphLab and PowerGraph.

Deserialization library of Boost v1.61.

24/04/2017
Figure 15: Discrepancy-Free Dual Testing: (1) Forward-Test from serial to parallel that runs a parallel execution with trace $S_{\text{trace}}$ from a serial execution and compares its outcome against the serial outcome $S_{\text{baseout}}$; and (2) Backward-Test from parallel to serial that runs a serial execution with trace $P_{\text{trace}}$ from a parallel execution and compares its outcome against the parallel outcome $P_{\text{baseout}}$.

5.5 Testing and Debugging Parallel ML on STRADS-AP

Debugging distributed ML programs is non-trivial and often time-consuming because of the non-deterministic characteristics of distributed computation from thread interleaving and process interleaving. To facilitate debugging, Spark and Hadoop frameworks ensure deterministic computation if all user-defined functions are deterministic. STRADS-AP also supports deterministic computation if all user-defined functions for parallel functions are deterministic. To achieve deterministic computation, STRADS-AP implements a deterministic scheduler that uses a globally-predefined seed value for random number generation.

In addition to debugging, testing distributed ML programs can also be non-trivial when an underlying framework implements ML-specific optimizations, such as reordering of update operations and stale synchronization[21] on shared parameters. In this case, objective values of serial baseline implementation and parallel implementation on a framework usually have a numerical difference. The common testing custom is to check the magnitude of the discrepancy between these two objective values and give an approval if the magnitude is small enough. The problem with this custom is that it is not easy to ensure that this discrepancy truly originates from the ML-specific optimizations and not from any bug in the parallel implementation. To address this problem, STRADS-AP provides a new testing method called “discrepancy-free dual testing” that enables to compare outcomes of a serial execution and a parallel execution without discrepancy in a test mode. The new testing method consists of two separate tests as shown in Figure 15: (1) forward-test from serial to parallel and (2) backward-test from parallel to serial.

In the forward-test, the serial execution is the baseline. The serial execution runs on a single node and stores an execution trace $S_{\text{trace}}$ and an outcome $S_{\text{baseout}}$ (i.e. a log of objective values per iteration) at the end of running. The parallel execution takes $S_{\text{trace}}$ as a program input, and the STRADS scheduler makes a parallel schedule plan whose execution is equivalent to the serial execution of $S_{\text{trace}}$. Finally, the parallel execution produces outcome $P_{\text{baseout}}$. To pass the
forward-test, $P_{out}$ must be equal to $S_{baseout}$. In brief, this testing procedure can be expressed as $P_{\text{execution}}(S_{\text{trace}}) = S_{\text{baseout}}$.

In the backward-test, the parallel execution is the baseline. The parallel execution runs on a cluster in a test mode and stores an execution trace $P_{\text{trace}}$ and an outcome $P_{\text{baseout}}$ at the end of running. The serial execution takes $P_{\text{trace}}$ as a program input, and the permutation of $j$ inside Parallel_For function reorders loop bodies corresponding to $P_{\text{trace}}$. Finally, the serial execution produces outcome $S_{\text{out}}$. To pass the backward-test, $S_{\text{out}}$ must be equal to $P_{\text{baseout}}$. In brief, this testing procedure can be expressed as $S_{\text{execution}}(P_{\text{trace}}) = P_{\text{baseout}}$.

Note that in the testing mode, STRADS-AP does not allow system optimizations such as pipelining and relaxed parameter scheduling. Because these optimization techniques introduce staleness to the parameter values, they may make it impossible to find a parallel execution plan that is equivalent to the serial execution.

6 Related Work

Data-Parallel Systems: Related work in data-parallel systems for parallel machine learning is categorized into two classes. First, dataflow frameworks — such as BSP[41], MapReduce[17][3], and Spark[44] — support various data processing operations based on a functional programming paradigm. Along this line, Mahout[4] on Hadoop[3] and, more recently, MLI[39]/MLlib/GraphX[42] on Spark[45] have been developed. However, these frameworks consider machine learning applications to be black boxes. They focus on correct execution by adhering to strict synchronization schemes, much like traditional computing, while ignoring the unique properties of ML and leaving huge opportunities for performance enhancements. These frameworks in particular suffer from high synchronization costs.

Second, several parameter server systems[1, 32, 29, 13, 21, 11] relax synchronization constraints by exploiting the error tolerance of ML programs in order to address the straggler and synchronization cost problems. Early PS systems were designed for a particular class of ML applications. Ahmed et al[1] implemented a fully asynchronous parameter server for a LDA topic modeling application. Dean et al[16] implemented a deep neural network specialized parameter server. More recent work [21, 29, 13, 11] has generalized the parameter server concept to support a wide range of ML algorithms. These recent PS systems support Stale Synchronous Parallel(SSP), which puts a bound on the staleness of parameter values. None of the above systems directly addresses the issue of conflict updates, which leads to slow convergence or algorithmic failure.

Model-Parallel System: Related work in model parallel systems for parallel machine learning falls into two classes as well. First, Dean et al[16] presented a DNN application specific model parallel approach that partitions a DNN model across workers and flows training data samples along the partitioned model. Each machine updates a portion of DNN model. This early work was implemented to train very large DNN models that do not fit into a single machine’s memory. However, it did not consider systematic approaches to handle data dependency and uneven convergence rate of model parameters.

Second, Gonzalez et al[31, 30, 19] presented the first systematic model-parallel approach called GraphLab. In GraphLab, ML’s computational dependencies are encoded by the user in an input
graph, so that the system may select disjoint subgraphs to process in parallel. GraphLab implements 2-Phase Locking (2PL) and chromatic engines to control concurrency among subgraphs. By using an input graph and execution engines, GraphLab addresses the issue of conflict updates and improves statistical progress per iteration. However, it is not able to make tradeoffs between statistical progress and system throughput in dependency scheduling and execution of programs. This inflexibility causes performance to degrade, particularly when the ML problem has a dense dependency structure. One might think of addressing this problem by pruning the edges of the input graph before running Graphlab; however, such preprocessing is often a non-trivial task because training input data is usually associated with edges. In contrast, STRADS explicitly separates the schedule from the update. Thus, STRADS scheduling has the flexibility to make trade-offs between statistical progress and system throughput according to a ML problem’s dependency structure. Moreover, STRADS execution engines also allow the trading of statistical progress for system throughput to maximize training speed.

Concurrency Control: For many decades, the DBMS community has intensively developed concurrency control machinery such as 2PL and optimistic concurrency control[20, 6]. However, this machinery cannot be an efficient solution for concurrency control for ML programs. This is because ML workloads have different properties from traditional DB workloads. For example, a ML problem usually consists of a very large number of transactions (millions to 10s of billions) and ML demands very high throughput (more than 10s of millions of transactions per second) in a cluster of commodity machines. Also, a ML problem has a relatively dense dependency structure. Thus, optimistic concurrency control will suffer from frequent transaction abortions. In STRADS-AT, I avoid building DBMS concurrency control machinery. Instead, I address the concurrency control problem by using virtual iteration and ML parameter scheduling.

Concurrent Programming Models: I limit the scope of discussion of concurrent programming models to imperative language. The most closely related work includes OpenMP[14], Cilk[8], and Intel’s Threading Building Blocks(TBB)[34]. These previous works follow the fork-join execution model and hide the details of concurrent programming, such as thread creation and barrier synchronization. However, these works require programmers to avoid concurrency errors such as deadlock and race condition. More recently, Berger et al introduced Grace[5], which prevents the concurrency errors without human efforts. Grace relies on the “threads-as-process” technique to provide memory isolation and executes threads speculatively. This approach lowers the bar for concurrent programming. However, it can not be an efficient solution for ML workloads because ML workloads often have dense dependency structures that cause frequent abortions of speculative execution. Moreover, all these works are limited to multi-thread computing on a single machine. In contrast, STRADS-AT targets distributed computing.

7 Timeline

- March 2017: completed
  Implement distributed data structures
  Implement Parallel_For and data processing functions

39
24/04/2017
April 2017:
Implement compile-time system
Implement list type DDS and more data processing function

May 2017:
Implement GraphLab and PS porting layer

June 2017:
Experiments / Preparing for paper submission

July – Oct 2017:
Dissertation writing and defense

References


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


