COP: Planning Conflicts for Faster Parallel Transactional Machine Learning

Faisal Nawab[1] Divyakant Agrawal[1] Amr El Abbadi[1] Sanjay Chawla[2,3]

[1]Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{nawab,agrawal,amr}@cs.ucsb.edu

[2]Qatar Computing Research Institute, HBKU
schawla@qf.org.qa
[3]University of Sydney, Sydney, NSW, Australia
sanjay.chawla@sydney.edu.au

ABSTRACT

Machine learning techniques are essential to extracting knowledge from data. The volume of data encourages the use of parallelization techniques to extract knowledge faster. However, schemes to parallelize machine learning tasks face the trade-off between obeying strict consistency constraints and performance. Existing consistency schemes require expensive coordination between worker threads to detect conflicts, leading to poor performance. In this work, we consider the problem of improving the performance of multi-core machine learning while preserving strong consistency guarantees.

We propose Conflict Order Planning (COP), a consistency scheme that exploits special properties of machine learning workloads to reduce the overhead of coordination. What is special about machine learning workloads is that the dataset is often known prior to the execution of the machine learning algorithm and is reused multiple times with different settings. We exploit this prior knowledge of the dataset to plan a partial order for concurrent execution. This planning reduces the cost of consistency significantly because it allows the use of a light-weight conflict detection operation that we call ReadWait. We demonstrate the use of COP on a Stochastic Gradient Descent algorithm for Support Vector Machines and observe better scalability and a speedup factor between 2-6x when compared to other consistency schemes.

1. INTRODUCTION

The increasingly larger sizes of machine learning datasets have motivated the study of scalable parallel and distributed machine learning algorithms [7, 16, 20–22, 24, 25, 27]. The key to a scalable computation is the efficient management of coordination between processing workers, or workers for short. Some machine learning algorithms require only a small amount of coordination between workers making them easily scalable. However, the vast majority of machine learning algorithms are studied and developed in the serial setting, which makes it arduous to distribute these *serial-based algorithms* while maintaining the algorithm's behavior and goals.

Distributing serial-based algorithms may be performed by encapsulating the algorithm within existing parallel and distributed computation frameworks. These frameworks are oblivious to the actual computation. Thus, the machine learning algorithms may be incorporated as-is without redesign. In this paper, we consider a framework of *transactions* [3, 10] for parallel multi-core execution of machine learning algorithms. A transaction may represent the processing of an iteration of the machine learning algorithm where workers run transactions in parallel. *Serializability* is the correctness criterion for transactions that ensures that the outcome of a parallel computation is equivalent to some serial execution. To guarantee serializability, transactions need to coordinate via consistency schemes such as locking [8] and optimistic concurrency control (OCC) [15].

Recently, coordination-free approaches to parallelizing machine learning algorithms have been proposed [7, 24, 25]. In these approaches, workers do not coordinate with each other thus improving performance significantly compared to methods like locking and OCC. Although these techniques were very successful for many machine learning problems, there is a concern that the coordination-free approach leads to "requiring potentially complex analysis to prove [parallel] algorithm correctness" [21]. When a machine learning algorithm, $\mathbb{A}$, is developed, it is accompanied by mathematical proofs to verify its theoretical properties, such as convergence. These proofs are typically on the serial-based algorithm. A coordination-free parallelization of a proven serial algorithm, denoted $\varphi_{cf}(\mathbb{A})$, is not guaranteed to have the same theoretical properties as the serial algorithm $\mathbb{A}$. This is due to overwrites and inconsistency that makes the outcome of $\varphi_{cf}(\mathbb{A})$ different from $\mathbb{A}$. Thus, guaranteeing the theoretical properties requires a separate mathematical analysis of $\varphi_{cf}(\mathbb{A})$, that although possible [6, 25], can be complex. Additionally, the theoretical analysis might reveal the need for changes to the algorithm to preserve its theoretical guarantees in the parallel setting [25].

Running parallel machine learning algorithms in a serializable, transactional framework bypasses the need for an additional theoretical analysis of the correctness of parallelization. This is because a serializable parallel execution, denoted $\varphi_{SR}(\mathbb{A})$, is equivalent to some serial execution of $\mathbb{A}$, and thus preserves its theoretical properties. We will call parallelizing with serializability, the *universal approach* because serial machine learning algorithms are applied to it

without the need of additional theoretical analysis or changes to the original algorithm.

In this work, we focus on the universal approach of parallelizing machine learning algorithms with serializable transactions. Consistency schemes like locking [8, 11], OCC [25], and others [2] incur a significant performance overhead. Traditional serializability schemes were designed mainly for database workloads. Database workloads are typically arbitrary, unrepeatable units of work that are unknown to the database engine prior to execution. This is not the case for machine learning workloads. Machine learning tasks are well defined. Most machine learning algorithms apply a single iterative task repeatedly to the dataset. Also, the dataset (*i.e.*, the machine learning workload) is typically processed multiple times within the same run of the algorithm, and is potentially used for different runs with different machine learning algorithms. Generally, machine learning datasets are also known, in offline settings, prior to the experiments. These properties of machine learning workloads make it feasible to plan execution. We call these the *dataset knowledge* properties.

We propose Conflict Order Planning (COP) for parallel machine learning algorithms. COP ensures a serializable execution that preserves the theoretical guarantees of the serial machine learning algorithm. It leverages the dataset knowledge properties of machine learning workloads to plan a partial order for concurrent execution that is serializable. It annotates each transaction (*i.e.*, a machine learning iteration) with information about its dependencies according to the planned partial order. At execution time, these *planned dependencies* must be enforced. Enforcing a planned dependency is done by validating that an operation reads or overwrites the correct version according to the plan. This validation is done using a light-weight operation that we call ReadWait. This operation is essentially an arithmetic operation that compares version numbers, which is a much lighter operation compared to locking and other traditional consistency schemes.

We present background about the problem, the system and transactional machine learning model in Section 2. Then, we propose COP in Section 3 followed by correctness proofs in Section 4. We present our evaluation in Section 5. The paper concludes with a discussion of related work and a conclusion in Sections 6 and 7.

2. BACKGROUND

In this section, we provide the necessary background for the rest of this paper. We introduce use cases of planning within machine learning systems in Section 2.1. Section 2.2 presents the transactional model we will use for machine learning algorithms.

2.1 Use Cases

We now demonstrate the opportunity and rewards of planning machine learning execution in three common models of machine learning systems. We revisit these use cases in the paper when appropriate to show how COP planning applies to them.

2.1.1 Machine Learning Framework

Machine learning and data scientists do not process a dataset only once in their process of analyzing it. Rather, the scientist works on a dataset continuously, experimenting with different methods and machine learning algorithms to discover what method works best with a dataset. Thus, the same dataset is being processed by many machine learning algorithms repeatedly. Figure 1 shows a typical flow diagram of a machine learning framework [12, 14, 27]. Multiple machine learning algorithms are applied to an input dataset to produce models of the dataset. Each machine learning algorithm may be applied multiple times with different configuration and parameters, such as the learning rate.

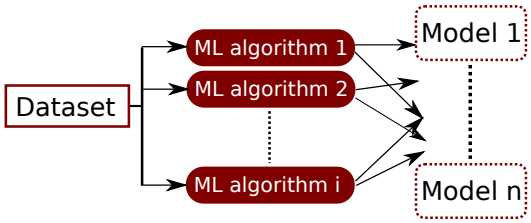


Figure 1: A flow diagram of a typical machine learning framework that employs a number of machine learning algorithms to learn models from a dataset

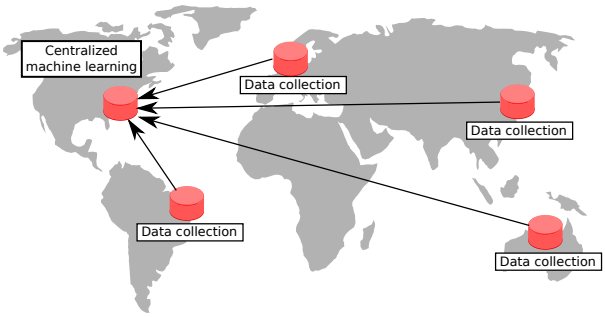


Figure 2: The current practice of machine learning of data collected across the world is to batch data at geo-distributed datacenters and send batches to a centralized location that performs the machine learning algorithm

In this model of a machine learning framework, the dataset is being processed many times, once for each generated model. This is an opportunity for COP to perform a planning stage that is then applied to all runs.

2.1.2 Global-Scale Machine Learning

Online machine learning is the practice of learning from data or events in real time. An example is a web application that collects user interactions with the website and generates a user behavior model using machine learning. Another example is applying machine learning to data collected by Internet of Things (IoT) and mobility devices. Typically, data is *born* around the world, collected at different datacenters, and then sent to a single datacenter that contains a centralized machine learning system. This case is shown in Figure 2 where there is a *central datacenter* for machine learning in North America and four other *collection datacenters* that collect and send data. This model has been reported to be the current standard practice of global-scale machine learning [4].

As data is being collected and batched at collection datacenters, there is an opportunity to generate a COP plan. This plan is then applied at the central datacenter for faster

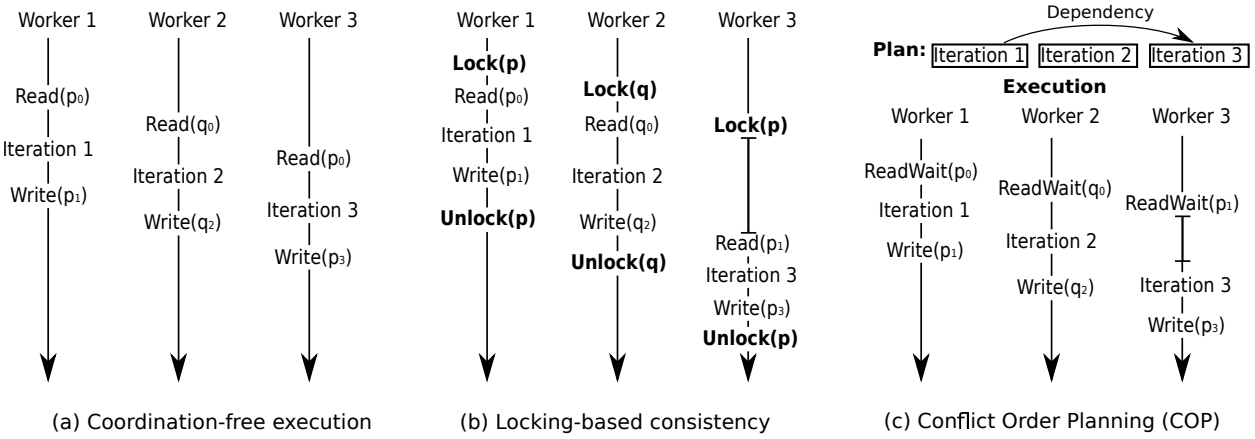


Figure 3: Execution of a machine learning algorithm by three workers with different consistency schemes. Each worker processes an iteration of the machine learning algorithm, where the first and third iterations read and update the same model parameter.

execution. A challenge in this model is that data is generated at different locations simultaneously and continuously. In such cases, COP plans for each batch individually at collection datacenters, and then batches are processed at the centralized datacenter in tandem.

2.1.3 Dataset Loading, Preprocessing and Execution

In addition to the opportunities for planning shown in use cases of machine learning systems, there is an opportunity for planning even in a single execution of a machine learning algorithm on a single dataset. This is because, typically, two tasks are performed prior to a machine learning algorithm execution: (1) Loading the dataset to main memory. Before execution, the dataset is stored in persistent storage, such as a disk. While loading the dataset from persistent storage, there is an opportunity to perform additional work to plan the execution. Our experiments demonstrate that planning while loading the dataset introduces a small overhead between 3% and 5% (Section 5.3).

Datasets are also typically preprocessed for various purposes such as formatting, data cleaning, and normalization [12]. Preprocessing is normally performed on the whole dataset, thus introducing an opportunity to plan execution while preprocessing is performed.

Even in the case of a dataset that is already preprocessed, loaded and ready to be learned, there is another opportunity to plan execution. A machine learning algorithm processes a dataset in multiple *rounds* on the dataset that we call epochs. Thus, planning during the first epoch will be rewarding for the execution of the remaining epochs.

In Section 3 we introduce COP planning algorithms and discuss their application to the various use cases we have presented.

2.2 Transactional Model of Machine Learning

A machine learning algorithm creates a mathematical model of a problem by iteratively learning from a dataset. The mathematical model of a machine learning algorithm is represented by *model parameters*, P, or parameters for short. For example, the mathematical model of linear regression takes the form $y = \sum_{i=1}^{n} \beta_i x_i + \epsilon$. The model parameters consist of the variables of the model, namely the vector of coefficients, β, and ϵ. The machine learning algorithm uses the dataset to estimate the parameter values that will result in the best fit to predict the dependent variable, y.

A dataset, $\mathbb{D}$, contains a number of samples, where the i^{th} sample is denoted $\mathbb{D}_i$. Each sample contains information about a subset of the parameters and the dependent variable corresponding to them. To distinguish between model parameters and parameter values in samples, we call the parameter values in samples *features*. For example, a dataset may contain information about movies. Each sample contains a list of the actors in a movie and whether the movie has a high rating. A mathematical model can be constructed to predict whether a movie has a high rating given the list of actors in it. Each parameter in the model corresponds to an actor. A sample contains a vector of feature values, where a feature has a value of 1 if the actor corresponding to it is part of the movie and 0 otherwise. A sample in the dataset also contains whether the movie has a high rating. Using the dataset, the mathematical model is constructed by estimating parameter values. These parameter values can then be utilized in the mathematical model to predict whether a new movie will have a high rating based on the actors in it.

Estimating model parameters is performed by iteratively learning from the dataset. Each iteration processes a single sample or a group of samples to have a better estimate of model parameters. An *epoch* is a collection of iterations that collectively process the whole dataset once. Machine learning algorithms run for many epochs until convergence. For example, Stochastic Gradient Descent (SGD) processes a single sample in each iteration. In an iteration, gradients are computed using a cost function to minimize the error in estimation. The gradients are then used to update the model parameters.

Machine learning algorithms are typically studied and designed for a serial execution where iterations are processed one iteration at a time. A straightforward approach to parallelizing a machine learning computation is to make workers process iterations concurrently, where each worker is responsible for the execution of a different iteration. Executing iterations concurrently may lead to conflicts among some of the updates from different workers, *e.g.*, updates from different workers to the same model parameters may overwrite each other. This means that the behavior of the algorithm no

Algorithm 1 Processing an iteration as a transaction

```
1: procedure PROCESS TRANSACTION T_i
2:     μ ← P.read(T_i.read-set)
3:     δ ← ML_computation(μ, T_i.sample, T_i.write-set)
4:     P ← δ
```

Algorithm 2 Parallel machine learning algorithm with Optimistic Concurrency Control

```
1: procedure PROCESS TRANSACTION T_i
2:     μ ← P.read_versioned(T_i.read-set)
3:     δ ← ML_computation(μ, T_i.sample, T_i.write-set)
4:     ATOMIC{
5:         P.validate(μ.versions)
6:         if not validated: abort or restart
7:         P ← δ
8:     }
```

longer resembles the intended serial execution of the machine learning algorithm.

Figure 3(a) illustrates the possibility of data corruption. Three workers are depicted processing three iterations of a machine learning algorithm concurrently. Each iteration reads a subset of the model parameters, computes new estimates of a subset of the model parameters, and finally writes them. In the figure, iterations 1 and 3 read and update the model parameter p and iteration 2 reads and updates the model parameter q. Iterations 1 and 3 read the same version of the parameter p, denoted p_0, and use it to calculate the new parameter value of p. Worker 1 writes the new state of p denoted p_1 and then worker 3 writes the new state of p denoted p_3. In this scenario, the work of worker 1 is overwritten by worker 3. Meanwhile, iteration 2 reads and updates parameter q, which does not corrupt the work of other iterations because it is not reading or writing parameter p.

Serializability can be the correctness criterion for parallel machine learning algorithms [21]. Serializability theory abstracts access to shared data by using the concept of a transaction [10] where a transaction is a collection of read and write operations on data objects. A data object is a subset of the shared state. The computation of an iteration i of a machine learning algorithm may be abstracted as a transaction, T_i, by considering reads of the model parameters as reads of data objects and writes to the model parameters as writes to data objects. We will denote the collection of model parameters by P, where $P[x]$ is the value of model parameter x. The parameters that are read by a transaction are denoted as $T_i.read\text{-}set$. Similarly, we will denote the parameters that are written by the transaction as $T_i.write\text{-}set$. The sample's data that is processed by iteration i is denoted by $T_i.sample$, where i is the id of the transaction. In the rest of the paper, we will use the terminology of transactions when appropriate, where a transaction is an iteration, and a data object is a model parameter.

The processing of a transaction follows the template in Algorithm 1 which is a transaction processing template that does not perform any coordination and is only serializable if run sequentially. The transaction template algorithm first reads the model parameters declared in the read-set and cache them locally as μ (line 2). Then, the read parameter values μ and the data sample information, $T_i.sample$, are used to compute new values of the parameters declared in the write-set (line 3). The new values are computed according to the used machine learning algorithms, and they are buffered locally as δ. Finally, the new parameter values, δ, are applied to the shared model parameters, P (line 4).

Serializability guarantees the illusion of a serial execution while being oblivious of the semantic computation performed within the transaction. Thus, it may be applied to machine learning algorithms. Serializability is achieved by ensuring that if some transactions conflict with each other, then they will not be executed concurrently. Detecting conflicts between concurrent transactions requires coordination among workers via different methods. These methods are diverse with different performance characteristics. We now present common transaction execution protocols that have been used in the context of machine learning algorithms, and we generalize them as transactional patterns that are oblivious to the machine learning algorithm. Readers familiar with transaction processing may skip to Section 2.3.

2.2.1 Locking

One of the most common methods for transaction management is *mutual exclusion* also known as lock-based protocols or pessimistic concurrency control [8]. In the rest of the paper, we will call it *Locking*. Locking is used in many parallel machine learning frameworks to support serializability [9, 19]. In this method, all read or written model parameters are locked during the processing of the transaction. These locks prevent any two transactions from executing concurrently if they access any common objects. Locking may be applied to the transactional pattern of Algorithm 1 by locking all data objects in the read-set and write-set at the beginning. These locks are released only after the transaction updates are applied to the shared model parameters.

Locking prevents conflicts such as the overwrite of worker 3 to worker 1's work in the scenario in Figure 3(a). The scenario with Locking is shown in Figure 3(b). Workers attempt to acquire a lock on the parameters they read or write before beginning the iteration. Worker 1 acquires the lock for p first and proceeds to compute and update the value of p before releasing the lock. Thus, it prevents worker 3 from overwriting its work because worker 3 will wait until it acquires the lock. Meanwhile, worker 2 acquires the lock for q and process iteration 2 because no other iteration is reading or updating q. This is a serializable execution because it resembles the serial execution of iteration 1, iteration 2, and then iteration 3. However, locking is an expensive operation that leads to a significant performance overhead even for iterations that do not need coordination, such as iteration 2.

2.2.2 Optimistic Concurrency Control

Optimistic concurrency control (OCC) [15] is an alternative to Locking. It performs better for scenarios with low contention, which made it more suitable for machine learning algorithms [21]. However, existing OCC methods for machine learning applications have been only proposed as specialized algorithms for domain-specific machine learning problems. Pan et. al. [21], for example, propose optimistic concurrency control patterns for DP-Means, BP-Means, and online facility location. Unlike these specialized OCC algorithms we present a generalized OCC pattern that can be applied to arbitrary machine learning algorithms.

A general OCC protocol [15] proceeds in three phases

shown in Algorithm 2 :

- *Phase I (Execution):* in the execution phase, the transaction's read-set is read from the shared model parameters (line 2). Model parameters in OCC are versioned, where the version number of a parameter is the id of the transaction that wrote it. The read parameter values and the sample information are then used in the machine learning computation (denoted ML_computation) to generate the updates to model parameters, δ (line 3). Note that during this phase no coordination or synchronization is performed.
- *Phase II (Validation):* in this phase we ensure that the read data objects were not overwritten by other transactions during the execution phase (lines 5-6). This is performed by reading the model parameters again after the computation and comparing the read versions to the current versions.
- *Phase III (Commit):* if the validation is successful, the updates, δ, are applied to the global model parameters (line 7).

One requirement for OCC to be serializable is to *perform the validation and commit phases atomically* (lines 4-8) [3]. To perform these two steps atomically, there are two typical approaches: (1) Execute these steps serially at a coordinator node. This, however, limits scalability, because it means that there is a dedicated worker that is doing the validation and commit for all iterations. Such a method can only be made efficient with domain knowledge about the machine learning problem, which means that the algorithm no longer becomes a general OCC scheme but rather a specialized OCC algorithm [21]. (2) The general approach for validation is to lock the write-set. This is different from Locking in two ways: locks are only held *after* the computation has been performed and only the data objects in the write-set are locked (data objects in the read-set are not locked). Thus, OCC outperforms Locking for cases when the contention is lower, and the write-set is significantly smaller than the read-set. This approach is adopted by recent state-of-the-art OCC transaction protocols in the systems and database systems community [29] and is the method we use in our evaluations.

2.3 Performance and Overheads of Consistency Schemes

Consistency schemes, such as Locking and OCC, incur overheads to ensure a serializable execution. These overheads are: (1) *Conflict detection overhead*: this is the overhead due to additional operations needed to detect conflicts, such as locks, atomic sections, and comparing versions. These overheads are incurred even in the absence of a conflict. (2) *Backoff overhead*: this is the wasted time that is incurred due to a detected conflict, such as waiting for a lock to be released, aborting due to deadlock, and failed validation.

For Locking, the conflict detection overhead is due to the operations to acquire and release locks. Even in the absence of conflict, these operations incur an overhead. The backoff overhead for Locking is the time spent waiting for acquired locks to be released. Deadlocks do not occur in our Locking algorithm. This is because locks are acquired in ascending order—locks with lower keys are acquired first. This is possible because the read and written data objects are declared at the beginning of the execution.

For OCC, the conflict detection overhead is due to the operations to acquire and release locks for the atomic section, and the overhead to validate the read-set. Unlike Locking, OCC locks are only for the data objects in the write-set. The backoff overhead for OCC is due to wasted processing time in the case of an abort and restart when validation fails.

3. CONFLICT ORDER PLANNING

In this section, we propose Conflict Order Planning (COP) for parallel machine learning that ensures a serializable execution while reducing the overhead of conflict detection. COP entails no use of locks or atomic blocks, which are expensive operations necessary for existing consistency schemes such as Locking and OCC.

3.1 Overview

COP leverages the dataset knowledge property of machine learning workloads: a machine learning algorithm processes a dataset of samples that is known prior to the experiment and is typically processed *multiple* times. This creates the opportunity to plan a partial order of execution to minimize the cost of conflict detection. Dataset knowledge is not manifested in traditional database systems. Thus, existing consistency schemes, such as Locking and OCC are designed with the assumption that they are oblivious of the dataset. The use of traditional database transactional methods leads to a lost opportunity as they do not exploit dataset knowledge. In this section, we propose COP algorithms that exploit dataset knowledge.

The intuition behind COP is to have a planned partial order of transactions prior to execution and then ensure that the partial order is followed during execution. We derive the planned partial order from an arbitrary starting serial order of transactions. For example, a dataset with n samples will be transformed to n transactions in some planned order $T_1, T_2, \ldots T_n$. We will represent this ordering by the relation $T_i <_o T_j$, where T_i is ordered before T_j. However, during execution, *the order is not enforced between every pair of transactions.* Rather, the order is only enforced for transactions that depend on each other. Thus, if T_2 does not depend on T_1 then a worker may start processing T_2 even if T_1 did not finish. Otherwise, processing T_2 must begin only after T_1 finishes. Thus, the enforced partial order is based on an initial serial order and the conflict relations between transactions.

DEFINITION 1. ***(Planned partial order)*** *There is a planned dependency—or dependency for short—from a transaction T_i to a transaction T_j if the planned order entails T_j reading or overwriting a write made by T_i. We denote this dependency by $T_i \rightsquigarrow_x T_j$ and it exists if all the following conditions are met:*

- *T_i writes the model parameter x ($x \in T_i.write\text{-}set$).*
- *T_j reads or writes the model parameter x ($x \in T_j.read\text{-}set \cup T_j.write\text{-}set$).*
- *T_i is ordered before T_j ($T_i <_o T_j$).*
- *There exists no transaction T_k that is both ordered between T_i and T_j and writes x ($\nexists T_k | x \in T_k.write\text{-}set \wedge T_i <_o T_k <_o T_j$).*

Enforcing the order between transactions that depend on each other is a sufficient condition to guarantee a serializable execution (see Section 4.1 for a correctness proof).

Algorithm 3 The COP partial order planning algorithm that is performed prior to the experiment.

```
1: Planned_version_list := A list to assign read and write versions initially all zeros
2: version_readers := A list to count the number of transactions that read a version
3: for T_i ∈ Dataset transactions do
4:     for r ∈ T_i.read-set do
5:         r.planned_version = Planned_version_list[r.param]
6:         version_readers[r.param]++
7:     for w ∈ T_i.write-set do
8:         w.p_writer = Planned_version_list[w.param]
9:         Planned_version_list[w.param] = i
10:        w.p_readers = version_readers[w.param]
11:        version_readers[w.param] = 0
12: Delete Planned_version_list and version_readers
```

COP enforces dependencies by versioning model parameters with the ids of the transactions that wrote them. A transaction only starts execution if the versions it depends on has been written. Consider applying COP to the scenario in Figure 3(a). The resulting execution is shown in Figure 3(c). Assume that the planned order is to execute samples 1, 2, and 3, in this order. The partial order consists of a single dependency from iteration 1 to iteration 3, because they both read and write p. Iterations use a special read operation called ReadWait that waits until the version it reads is written by the transaction that it depends on. Iteration 1 is planned to read the initial version of p, denoted p_0, because it is the first ordered iteration to read p. Likewise, iteration 2 is planned to read the initial version of q. Iteration 3 depends on Iteration 1, because they both read and write p. Thus, iteration 3 is planned to read the version of p that is written by iteration 1, denoted p_1. With this plan, workers 1 and 2 process iterations 1 and 2 concurrently after verifying that they have read their planned versions. Worker 3, however, waits until the version p_1 is written by worker 1 and then proceeds to process iteration 3. With COP, workers coordinate without the need of expensive locking primitives. Rather, workers only utilize simple arithmetic operations on the read or written parameter's version number to enforce the plan.

In the remainder of this section, we propose the COP planning algorithm that is used to find and annotate dependency relations between transactions (Section 3.2). Then we propose the COP transaction execution algorithm that enforces dependency relations (Section 3.3). We discuss the performance benefits of COP in Section 3.4.

3.2 COP Planning Algorithm

In this section, we present the COP planning algorithm in its basic form—planning prior to execution. Then, we discuss how it can be used to plan in conjunction with the first epoch and how it can be used in cases where there are multiple sources of data.

3.2.1 Basic COP Planning

We begin by presenting the basic COP planning strategy. Here, we assume that planning is performed before execution, either in offline settings or while loading the dataset. The objective of the planning algorithm is to annotate the dataset with the planned partial order information. This annotation includes the following:

Definition 2. ***(COP planning and annotation)*** *COP planning performs the following two annotations: (1) Read annotation: each read operation is annotated with the version number it should read, and (2) Write annotation: each write operation, w, is annotated with the id of the version it should overwrites, w', and the number of transactions that are planned to read the version w'.*

The read annotation's goal is to enforce the order during execution. The write annotation's goal is to ensure that a version is not overwritten until it is read from all the transactions that are planned to read it.

Algorithm 3 shows the steps to annotate transactions with the partial order information. The algorithm processes transactions one transaction at a time ordered by some arbitrary order—beginning with T_1 and ending with T_n.

In COP, each read operation in the read-set, r, contains both the read parameter to be read ($r.param$), and the planned read version number ($r.planned\_version$), *i.e.*, the read annotation. A planned version number k means that the transaction must read the value written by transaction T_k. Also, each write in the write-set, w, contains the parameter to be written ($w.param$), the number of transactions that read the previous version ($w.p\_readers$), and the transaction id of the transaction that it is overwriting ($w.p\_writer$), *i.e.*, the write annotation.

The planning algorithm tracks the planned version numbers in a list named *Planned_version_list* as dependencies are being processed. $Planned\_version\_list[x]$ contains the unique transaction id of the most recently planned transaction that writes x. All entries in the list are initialized to 0. Also, the number of version readers are maintained in a list named *version_readers*. At any point in the planning process, $version\_readers[x]$ contains the number of planned transactions that read the most recently planned written version of x. Both lists are only used within the planning algorithm and are deleted before the execution phase.

The planning of a transaction T_i proceeds by processing the read-set and then the write-set. Each read operation r in the read-set is annotated with a planned version from the *Planned_version_list* (lines 4-5). For example, consider the case where T_i reads model parameter x. Then, there is a read, r, with $r.param$ equals to x. At the time r is being planned, the corresponding value in the list, $Planned\_version\_list[x]$ contains the unique transaction id, k, of the last transaction, T_k, that wrote x. Thus, assigning the planned version of r to k is a way of encoding that the plan is for T_i to read the value of x that was written by T_k. Then, the corresponding number of version readers is incremented (line 6). After processing the read-set, the planning algorithm processes each write w in the write-set (lines 7-11). Each write is annotated with the previous writer's version number (line 8). Then, the corresponding entry in *Planned_version_list* is updated with the transaction id value i (line 9). Thus, reads of transactions ordered after T_i can observe that they are planned to read T_i's writes. Then, the write is annotated with the number of readers of the previous version (line 10). Finally, the corresponding entry in *version_readers* is reset. After all the operations are processed, the lists *Planned_version_list* and *version_readers* are deleted.

The outcome of the algorithm is read and write annotations

Algorithm 4 Parallel execution with COP

```
1: Global num_reads := initially all zeros
2: procedure PROCESS TRANSACTION T_i
3:     for r ∈ T_i.read-set do
4:         μ ← P.ReadWait(r)
5:         num_reads[r.param]++
6:     δ ← ML_computation (μ, T_i.sample, T_i.write-set)
7:     for w ∈ δ do
8:         w.version = i
9:         while w.p_readers ≠ num_reads[w.param] OR
       w.p_writer ≠ P[w.param].version do
10:            Wait
11:        num_reads[w.param] = 0
12:    P ← δ
```

of the whole dataset. The algorithm only requires a single pass on the dataset. In the evaluation section, we perform experiments to quantify the overhead of planning.

3.2.2 Alternative Planning Strategies

The basic COP planning algorithm, presented in the previous section, assumes that planning is performed prior to execution in offline settings or during dataset loading. We now show how to adapt the algorithm to plan in alternative planning scenarios. The first alternative is to plan during the first epoch of the machine learning algorithm's execution. The plan's objective is to annotate transactions with a partial order of a serializable execution. It is possible to execute the first epoch of the machine learning algorithm via a traditional consistency scheme (*e.g.*, Locking) and then annotate the dataset with the partial order of that epoch. Specifically, during the first epoch using Locking, the planning Algorithm 3 is performed for each transaction while all the locks of that transaction are held. Thus, each read is annotated with the read version and each write is annotated with the version it overwrites and the number of readers. After the first epoch, that has passed through the whole dataset, the remaining epochs are processed using COP with the annotated plan. The planning only adds a small overhead to the first epoch, as we discuss in the evaluation section.

Another alternative is to plan when the dataset is being generated online from multiple sources, in cases such as the global analytics scenario in Section 2.1.2. In such a scenario, planning can be done at each source for batches of samples using Algorithm 3. Then, at the centralized location, the machines learning algorithms process batches in tandem. The dependencies of a batch are transposed to previous batches. For example, consider two batches b_1 and b_2, where b_1 is processed in the centralized location prior to b_2. The transactions in b_2 that have dependencies on the initial version, according to Algorithm 3, are transposed to the most recent version written by b_1. For example, the first transaction that accesses x in b_2 will be annotated as reading the version 0. However, the centralized location will translate this as an annotation to wait for the last version written by b_1.

3.3 Planned Execution Algorithm

We present the COP execution algorithm (shown in Algorithm 4) that processes transactions in parallel according to a planned partial order. We associate each model parameter with a version number that corresponds to the transaction that wrote it, *e.g.*, $P[x].version$ is the current version number of model parameter x. A list of the number of version readers for model parameters, *num_reads*, is maintained and accessible by all workers. For example, a value for $num\_reads[x]$ of 3 means that so far, three transactions read the current version of x.

Dependencies between transactions are enforced by ensuring that read operations read the planned versions. T_i's read-set is read from the shared model parameters, P (lines 3-5). The ReadWait operation blocks until the annotated planned version is available. The implementation of ReadWait simply reads both the data object and its version number. Then, it compares the version number to the annotated read version number. If they match, the read data object is returned; otherwise, the read is retried until the planned version is read.

After reading the planned version, the number of version readers is incremented (line 5). Then, the transaction execution proceeds by performing the machine learning computation using the read model parameters and the data sample's information (line 6). Writes to the model parameters computed by the machine learning computation, δ, are buffered before they are applied to the model parameters (lines 7-11). First, each write, w, is tagged with a version number equal to the transaction's id (line 8). Thus, future transactions that read the state can infer that T_i is the transaction that wrote these updates. Then, the algorithm waits until the previous version has been read by all planned readers by making sure that the number of version readers is equal to the planned number of readers of that version and by making sure that the current version is identical to $w.p\_writer$ (lines 9-10). Since we are writing a new version, the corresponding entry in *num_reads* is reset to 0. The writes are then incorporated in the shared state (line 12).

3.4 Performance and Overheads

In Section 2.3 we discussed two overheads of consistency schemes: conflict detection overhead and backoff overhead. The backoff overhead incurred in COP is similar to Locking and OCC, *i.e.*, transactions wait until conflicting transactions complete. COP's goal is to minimize the other source of overhead: conflict detection overhead that is incurred whether a conflict is detected or not. In COP, the conflict detection overhead is due to: (1) The validation using the ReadWait operation, and (2) Validation that each write operation's previous readers have already read the previous version. These two tasks are performed via arithmetic operations and comparisons only, without the need for expensive synchronization operations like acquiring and releasing locks. This is the main contributor to COP's performance advantage.

4. CORRECTNESS PROOFS

In this section, we present two proofs. The first proves that COP is serializable and the second proves that deadlocks do not occur in COP.

4.1 COP Serializability

We prove the correctness of COP and that it ensures a serializable execution that is equivalent to a serial execution. We use a serializability graph (SG) to prove COP's serializability [3]. A protocol is proven serializable if the SGs that represent its possible executions do not have cycles.

A SG consists of nodes and edges. Each node represents a committed transaction. A directed edge from one node to another represents a conflict relation. There are three types of conflict relations (edges) in SGs:

- *Write-read (wr) relation*: This relation is denoted as $T_i \rightarrow_{wr} T_j$, which means that there is a *wr* edge from T_i to T_j. This relation exists if T_i writes a version of a data object x and T_j reads that version.
- *Write-write (ww) relation*: This relation is denoted as $T_i \rightarrow_{ww} T_j$, which means that there is a *ww* edge from T_i to T_j. This relation exists if T_i writes a version of a data object x and T_j overwrites that version with a new one.
- *Read-write (rw) relation*: This relation is denoted as $T_i \rightarrow_{rw} T_j$, which means that there is a *rw* edge from T_i to T_j. This relation exists if T_i reads a version of a data object x and T_j overwrites that version with a new one. If this edge exists between two transactions ($T_i \rightarrow_{rw} T_j$) then it must be the case that there exists a transaction T_k that writes x with the following conflict relations: (1) A write-write conflict relation from T_k to T_j ($T_k \rightarrow_{ww} T_j$), and (2) a write-read conflict relation from T_k to T_i ($T_k \rightarrow_{wr} T_i$).

LEMMA 1. *For any conflict relation $T_i \rightarrow T_j$ in SG of a COP execution, the following is true: $T_i <_o T_j$, where $<_o$ is the ordering relation of the initial planned order.*

PROOF. Assume that the data object that causes the conflict relation is data object x. We prove this lemma for the three conflict relations:

- *Write-read (wr) conflict relations ($T_i \rightarrow_{wr} T_j$):* according to Definition 1 a transaction T_j is planned to read from a transaction T_i if there is an ordering dependency $T_i \rightsquigarrow_x T_j$. One of the conditions of this ordering dependency is that T_i is ordered before T_j ($T_i <_o T_j$). In the implementation algorithm, this is enforced by the ReadWait operation (see Algorithm 4 lines 3-4).
- *Write-write (ww) conflict relations ($T_i \rightarrow_{ww} T_j$)):* according to Definition 1 a transaction T_j is planned to overwrite a value written by transaction T_i if there is an ordering dependency $T_i \rightsquigarrow_x T_j$. One of the conditions of this ordering dependency is that T_i is ordered before T_j ($T_i <_o T_j$). In the implementation algorithm, this is enforced by the check of *w.p_writer* (see Algorithm 4 lines 9-10).
- *Read-write (rw) conflict relations ($T_i \rightarrow_{rw} T_j$):* this relation implies the existence of a transaction T_k with the relations $T_k \rightarrow_{wr} T_i$ and $T_k \rightarrow_{ww} T_j$. According to our analysis in the previous two points, the following is true:

$$T_k <_o T_i \quad and \quad T_k <_o T_j \tag{1}$$

Thus, the following ordering dependencies exist:

$$T_k \rightsquigarrow_x T_i \quad and \quad T_k \rightsquigarrow_x T_j \tag{2}$$

We now show by contradiction that the following is true: $T_i <_o T_j$. Assume to the contrary that $T_j <_o T_i$ is true. If $T_j <_o T_i$ then according to Equation 1 the following is true:

$$T_k <_o T_j <_o T_i \tag{3}$$

However, this equation violates one of the definitions in Definition 1 that states that the ordering relation $T_k \rightsquigarrow_x T_i$ that exists according to Equation 2 implies that there exists no transaction that is ordered between them and writes x. However, according to Equation 3, T_j is ordered between T_k and T_i and it writes x. This violation leads to a contradiction to $T_j <_o T_i$ thus proving that $T_i <_o T_j$. In the implementation algorithm, this is enforced by the check of *w.p_readers* (see Algorithm 4 lines 9-10).

The condition of the lemma is proven for all three conflict relations. □

THEOREM 1. *Conflict Order Planning (COP) algorithms guarantee serializability.*

PROOF. According to Lemma 1, a conflict relation $T_i \rightarrow T_j$ in SG means that $T_i <_o T_j$. We need to show that a cycle $T_i \rightarrow \ldots \rightarrow T_i$ cannot exist. Assume to the contrary that such a cycle exists. This means according to Lemma 1 that $T_i <_o \ldots <_o T_i$. Since the ordering relation $<_o$ is transitive this leads to $T_i <_o T_i$, which is a contradiction, thus proving that no cycles exist in the SG of COP executions. The absence of cycles in SG is a sufficient condition to prove serializability [3]. □

4.2 COP Deadlock Freedom

The COP execution algorithm 4 can block in two locations: (1) a read waits for its planned version to be available, and (2) a write waits until all reads of the previous versions and the write of the previous version are complete. In this section we prove that these waits do not cause a deadlock scenario where a group of transactions are waiting for each other. We prove this by constructing a deadlock graph (DG). Nodes in DG are transactions. A directed edge from one transaction to another, $T_i \rightarrow_d T_j$, denotes that T_j may block waiting for a read or a write of T_i.

LEMMA 2. *For any edge in DG, $T_i \rightarrow_d T_j$, the following is true: $T_i <_o T_j$, where $<_o$ is the ordering relation of the planned order.*

PROOF. An edge $T_i \rightarrow_d T_j$ exists in three cases: (1) T_j reads a version written by T_i, (2) T_j overwrites a version written by T_i, or (3) T_j overwrites a version to be read by T_i. All cases are true in the COP algorithms only if the ordering dependency $T_i \rightsquigarrow T_j$ exists. According to Definition 1, an ordering dependency $T_i \rightsquigarrow T_j$ is only true if $T_i <_o T_j$. □

THEOREM 2. *Conflict Order Planning (COP) algorithms guarantee deadlock freedom.*

PROOF. According to Lemma 2, a dependency relation $T_i \rightarrow_d T_j$ in DG means that $T_i <_o T_j$. We need to show that a cycle $T_i \rightarrow_d \ldots \rightarrow_d T_i$ cannot exist. Assume to the contrary that such a cycle exists. This means according to Lemma 2 that $T_i <_o \ldots <_o T_i$. Since the ordering relation $<_o$ is transitive this leads to $T_i <_o T_i$, which is a contradiction, thus proving that no cycles exist in the DG of COP executions. The absence of cycles in DG is a sufficient condition to prove deadlock freedom. □

	Properties				Performance (M transactions/s)			
Dataset	# features	training set size	test set size	avg. transaction size	Ideal	COP	Locking	OCC
KDDA [26]	20,216,830	8,407,752	510,302	36.3	7.2	4.1	0.75	0.82
KDDB [26]	29,890,095	19,264,097	748,401	29.4	8.0	5.8	0.95	1.0
IMDB	685,569	167,773		14.6	15.2	11.0	6.7	4.9

Table 1: Performance comparison across of COP, Locking, OCC, and Ideal (without conflict detection) for three datasets

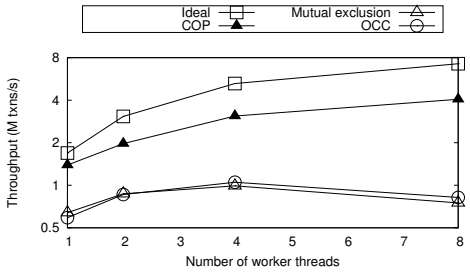
5. EVALUATION

In this section, we evaluate COP in comparison to Locking and OCC. We also compare with an upper-bound of performance, which is the performance without any conflict detection. We will call this upper-bound the *ideal* baseline, or Ideal for short. Ideal does not guarantee a serializable execution, unlike COP, Locking, and OCC. Thus, Ideal does not guarantee preserving the theoretical properties of the machine learning algorithm.

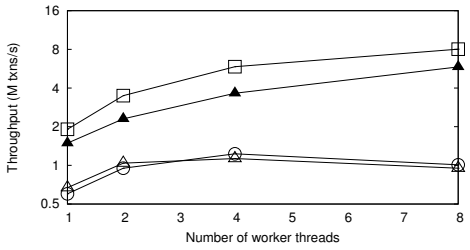
The transactional framework of machine learning can be applied to a wide-range of machine learning algorithms. For this evaluation, we run our experiments with a Stochastic Gradient Descent (SGD) algorithm to learn a Support Vector Machine (SVM) model. The goal of the machine learning algorithm is to minimize a cost function f. We use a separable cost function for SVM [25]. Each iteration in SGD processes a single sample from the dataset. Gradients are computed according to the cost function. The gradients are then used to compute the new values of the model that are relevant to the sample. We apply this machine learning algorithm to the transactional template we presented in Algorithm 1. Each transaction corresponds to an iteration of SGD. The iteration computation (*i.e.*, ML_computation() in the algorithm) represents the gradient computation using the cost function. For this machine learning algorithm, the read and write-sets of a transaction are the features in the corresponding sample, *i.e.*, the features with a non-zero value. In all experiments, we initialize the SGD step size value to 0.1. The step size value diminishes by a factor 0.9 at the end of each epoch over the training dataset. All experiments are run for 20 epochs, where an epoch is a complete pass on the whole dataset.

We implemented COP, Locking, and OCC as a layer on top of the parallel machine learning framework of Hogwild! [25] that is available publicly[1]. The source code is written in C++. We use an Amazon AWS EC2 virtual machine to run our experiments. The virtual machine type is c4.4xlarge with 30 GB memory and 16 vCPUs that are equivalent to 8 physical cores (Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz) and 16 hyper threads. Unless mentioned otherwise, the number of worker threads used in the experiments is 8. Our experiments with more than 8 threads show no significant performance difference.

We use three datasets to conduct our experiments, summarized in Table 1. The first two datasets are KDDA and KDDB datasets, which were part of the 2010 KDD Cup [26]. KDDA (labeled algebra_2008_2009) has 20,216,830 features and contains 8,407,752 samples in the training set and 510,302 samples in the test set. KDDB (labeled bridge_to_algebra_2008_2009) has 29,890,095 features and contains 19,264,097 samples in the training set and 748,401 samples in the test set. The third dataset is the IMDB dataset[2] that has 685,569 features and contains 167,773 samples. The IMDB dataset is not divided into training and test sets. The average sample (transaction) size of each dataset, which is the number of model parameters represented in each sample, is 36.3 for KDDA, 29.4 for KDDB, and 14.6 for IMDB. In addition to these three datasets, we use synthetic datasets for experiments that require controlling the dataset properties, such as contention.



(a) Throughput with the KDDA dataset



(b) Throughput with the KDDB dataset

(c) Throughput with the IMDB dataset

Figure 4: The throughput of Ideal, COP, Locking, and OCC while varying the number of threads for three datasets (log scale is used)

5.1 Throughput

The metric that we are interested in the most is throughput. We measure throughput as the number of processed samples

[1]http://i.stanford.edu/hazy/victor/Hogwild/

[2]http://komarix.org/ac/ds/

(*i.e.*, transactions) per second. Table 1 shows a summary of throughput numbers for the different evaluated methods on the three datasets. COP outperforms Locking and OCC by a factor of 5-6x for KDDA and KDDB. For IMDB, COP's throughput is 64% higher than Locking and 124% higher than OCC. The magnitude of performance improvement of COP compared to Locking and OCC is influenced by the level of contention in the dataset, *i.e.*, the likelihood of conflict between transactions. Our inspection of the datasets revealed that there is more opportunity for conflict in the KDDA and KDDB datasets than the IMDB dataset. We do not present the statistical properties of the datasets to show this due to the lack of space. However, we perform more experiments in Section 5.2 to study the effect of contention on performance. The comparison with Ideal shows that COP's throughput is 27-44% lower than Ideal. This percentage represents COP's overhead to preserve consistency. Although conflicts are planned in COP, there is still an overhead for conflict detection and backoff.

The throughputs of Locking and OCC are relatively close to each other. For KDDA and KDDB, the throughputs of Locking and OCC are within 10% of each other. For IMDB, Locking outperforms OCC by 36.7%. In the case of KDDA and KDDB, the locking contention for both Locking and OCC (to implement atomic validation) dominates performance. In general, OCC benefits in cases where the read-set is larger than the write-set. Because our machine learning workload has a read and write-sets of equal sizes, the advantage of OCC is not manifested (see Section 2.3). In IMDB, which is the workload with less contention, Locking outperforms OCC. This is due to the additional work needed to validate transactions by OCC. For the conflict detection overhead, OCC experiences both the overheads of locking and validation, while Locking only experiences the overhead of locking. The overhead of validation is exposed with workloads with less contention because in these cases, locking contention does not dominate performance, *i.e.*, in the case of the KDDA and KDDB datasets, the overhead due to locking contention dominates the validation overhead. We revisit the effect of contention in Section 5.2.

In Figure 4, we show the performance of the different schemes while varying the number of threads. Increasing the number of threads increases contention. Also, using more cores in the experiment exposes the effect of the underlying cache and cache coherence on the performance of the different schemes. Figure 4(a) shows the performance for the KDDA dataset. Consider the throughput of all schemes with a single worker thread. In this case, there is no conflict or cache coherence overhead. What is observed is the *conflict detection* overhead in isolation (Section 2.3). Ideal is only 21% higher than COP in the case of a single worker thread. This shows that the overhead of conflict detection is small compared to Locking and OCC; the throughput of Ideal is 163% higher than Locking and 186% higher than OCC.

For scenarios with more than one worker thread in Figure 4(a), the backoff and cache coherence overheads are experienced in addition to the conflict detection overhead. Ideal does not suffer from the backoff overhead because conflicts are not prevented. Also, Ideal has an advantage with the cache coherence overhead compared to the consistent schemes. Unlike COP, Locking, and OCC, Ideal does not maintain additional locking or versioning data that may be invalidated by cache coherence protocols. These factors cause

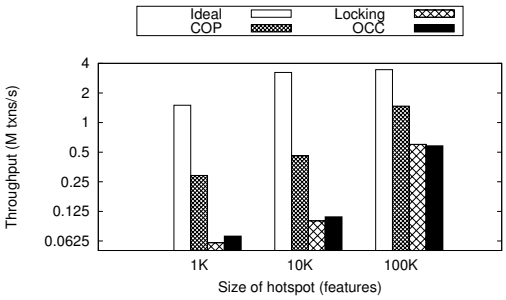


Figure 5: Quantifying the effect of contention on performance by experiments on synthetic datasets with varying contention levels

the performance gap between Ideal and the other schemes to grow as the number of threads is increased. COP's throughput, for example, is 17% lower than Ideal with one worker thread, but it is lower by 43% in the case of 8 threads. The contention between cores due to cache coherence limits scalability. Ideal with 8 threads achieves 4 times the performance of the case with a single thread—rather than 8 times the performance in the case of linear scalability. COP with 8 threads achieves 3 times the performance of the case with a single thread. For Locking and OCC, the contention is so severe that performance slightly decreases beyond 4 threads.

We show the same set of experiments for KDDB and IMDB in Figures 4(b) and 4(c). The experiments with the KDDB dataset show similar behavior to the experiments with the KDDA dataset. One difference is that COP scales better, as the KDDB dataset is sparser than KDDA; for KDDB, COP's throughput with 8 threads is 4 times the throughput with a single thread, rather than a 3x factor with the KDDA dataset. For the IMDB dataset, there is less contention compared to KDDA and KDDB. All schemes—including Locking and OCC—scale with a factor around 4x when increasing the number of threads from 1 to 8. Also, the smaller transaction sizes with the IMDB dataset makes the absolute throughput numbers higher than those with the KDDA and KDDB datasets.

5.2 Contention Effect

Contention affects performance because it increases the rate of conflict. A conflict between two transactions causes at least one of them to either wait or restart, thus wasting resources. Here, we quantify the effect of contention on the performance of our consistency schemes. We generate synthetic datasets to give us more flexibility in controlling the contention. The synthetic datasets we generate contain one million samples each. We fix the size of each sample to 100 features, which means that each transaction contains 100 data objects in the read and write-sets. To control the contention, we restrict transactions to a hot spot in the parameter space. Each data object is sampled uniformly from the hot spot. We control contention by varying the size of the hot spot.

Figure 5 shows the performance with hot spot sizes of 1K, 10K, and 100K features. Contention leads to a higher conflict overhead and lower performance. This is why consistency schemes perform lower in the highest contention case (1K features) when compared to cases with less contention. The performance improvement factor of the case with 100K fea-

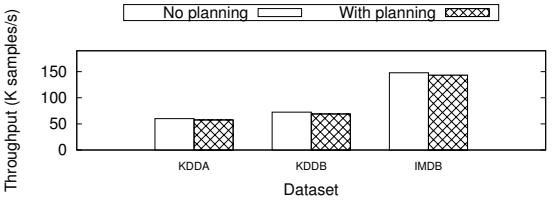


Figure 6: A comparison of the loading time of the dataset to main memory with and without order planning.

tures compared to the case with 1K features is 4x for COP, 8.8x for Locking, and 7.3x for OCC. Ideal also performs lower as contention increases, although it does not face an overhead due to conflicts. The performance of Ideal with 100K features is 131% higher than the performance with 1K features. The reason is that more contention also means more contention on cache lines, leading to a larger overhead for cache coherence.

As contention decreases, the performance gap between the consistency schemes and Ideal decreases. Part of the performance benefit of Ideal compared to the consistency schemes is that Ideal does not block or restart transactions due to conflicts. As contention decreases, this performance benefit diminishes, and the performance of the consistency schemes becomes closer to Ideal. For example, in the high contention case (1K features) Ideal's throughput is 4x the throughput of COP. For the low contention case (100K features), this gap decreases with Ideal's throughput only 34% higher than COP. This is also true for Locking and OCC, where Ideal's throughput is higher than them by a factor of 20-23x in the high contention case, but this factor decreases to around 5x for the low contention case.

Like the performance difference between Ideal and the other consistency schemes, the performance gap between COP and the other consistency schemes (Locking and OCC) also decreases as contention decreases. COP's light-weight conflict detection makes it less prone to conflicts than Locking and OCC because the latency of the transaction is lower. Thus, Locking and OCC suffer from contention more than COP. In the low contention case, COP's throughput is 3.7x higher than Locking and 3.1x higher than OCC. This performance gap decreases in the low contention case where COP outperforms Locking by 46% and OCC by 51%.

5.3 Planning Overhead

COP's performance advantage is due to having conflicts planned ahead of time. We have outlined in Section 2.1 examples of machine learning environments. In these environments, planning can be done in advance, and thus the planning overhead is not observed when the machine learning algorithm is processed using COP. This includes the case of the machine learning framework where a dataset is reused in different experiments and is possibly stored with the annotated plan for future sessions. However, there are cases where machine learning algorithms are used for fresh and raw datasets. In these cases, the planning overhead becomes important.

We performed several experiments to quantify the overhead of planning. We propose two alternatives to plan for a dataset. The first planning strategy is to plan while loading the dataset. A dataset is typically stored in a persistent storage such as a disk. Planning can be done in conjunction with reading the raw dataset from persistent storage and loading it into the appropriate data structures in main memory. Figure 6 shows the loading throughput with and without planning for three datasets. Planning only adds a small overhead to loading that we measure to be between 3% and 5%.

The second planning strategy is to plan during the first epoch and then use the plan for later epochs (Section 3.2.2). In the first epoch, a consistency scheme must be used. We run the first epoch using Locking and the rest of the epochs using COP. The throughput of the first epoch is within 1% of the throughput of Locking for all our datasets. The throughput of the remaining epoch is also within 1% of the performance of COP with offline planning.

6. RELATED WORK

The use of transactional and consistency concepts have been explored recently for parallel and distribute machine learning by Pan et.al [20–24]. Some of these works build consistent algorithms that follow the OCC pattern for distributed unsupervised learning [21], correlation clustering [20, 24], and submodular maximization [22]. These proposals show that domain-specific implementations of OCC—rather than general OCC that we presented in this work—achieve performance close to their coordination-free counterparts while guaranteeing serializability [22, 24].

The study of consistent machine learning algorithms has been motivated by the complexity of developing mathematical guarantees and coordination-free algorithms that are parallel [20–24]. However, many coordination-free machine learning algorithms were developed [1, 6, 18, 25]. Hogwild! [25], for example, is an asynchronous parallel SGD algorithm with proven convergence guarantees for several classes of machine learning algorithms.

Bounded staleness has been proposed as an alternative to both serializability and coordination-free execution for parallel machine learning. Bounded staleness is a correctness guarantee of the freshness of read data objects. It has been demonstrated for distributed machine learning tasks [13, 17]. Bounded staleness, however, may still lead to data corruption which requires a careful design of machine learning algorithms that leverage bounded staleness.

The concept of planning execution to improve the performance of distributed and parallel transaction processing has been explored in different contexts. Calvin [28] is a deterministic transaction execution protocol. Sequencing workers intercept transactions and put them in a global order that is enforced by scheduling workers. Calvin is built for typical database transactional workload and thus does not leverage the dataset knowledge property of machine learning workloads. This makes its design incur unnecessary overheads compared to COP for machine learning workloads, such as always having the sequencing and scheduling workers in the path of execution. Schism [5] is a workload-driven replication and partitioning approach. The access patterns are learned from the coming workload to create partitioning strategies that minimize conflict between partitions and thus improve performance. Cyclades [23] adopts a similar approach to Schism for parallel machine learning workloads. Cyclades improves the performance of both conflict-free and consistent machine learning algorithms by partitioning access for batches of the dataset to minimize conflict between parti-

tions. Each partition is then processed by a dedicated thread, leading to better performance. Partitioning for performance complements COP's objective. Whereas partitioning aims to minimize conflict between workers, COP ensures that conflicts are handled more efficiently.

7. CONCLUSION

In this paper, we propose Conflict Order Planning (COP) for consistent parallel machine learning. COP leverages dataset knowledge to plan a partial order of concurrent execution. Planning enables COP to execute with light-weight synchronization operations and outperform existing consistency schemes such as Locking and OCC while maintaining serializability for machine learning workloads. Our evaluations validate the efficiency of COP on a SGD algorithm for SVMs.

8. ACKNOWLEDGMENT

This work is partially funded by a gift grant from Oracle and a gift grant from NEC Labs America. We would also like to thank Amazon for access to Amazon EC2.

References

[1] H. Avron et al. Revisiting asynchronous linear solvers: Provable convergence rate through randomization. *Journal of the ACM (JACM)*, 62(6):51, 2015.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[3] P. A. Bernstein and E. Newcomer. *Principles of transaction processing.* Morgan Kaufmann, 2009.

[4] I. Cano et al. Towards geo-distributed machine learning. In *Workshop on ML Systems at NIPS*, 2015.

[5] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *VLDB*, 3(1-2):48–57, 2010.

[6] C. M. De Sa, C. Zhang, K. Olukotun, and C. Ré. Taming the wild: A unified analysis of hogwild-style algorithms. In *NIPS*, pages 2674–2682, 2015.

[7] J. Dean et al. Large scale distributed deep networks. In *NIPS*, pages 1223–1231. 2012.

[8] K. P. Eswaran et al. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[9] J. E. Gonzalez et al. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

[10] J. Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[11] J. Gray and A. Reuter. *Transaction processing: concepts and techniques.* Elsevier, 1992.

[12] M. Hall et al. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[13] Q. Ho et al. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231. 2013.

[14] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

[15] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, 1981.

[16] H. Li, A. Kadav, E. Kruus, and C. Ungureanu. Malt: distributed data-parallelism for existing ml applications. In *EuroSys*, 2015.

[17] M. Li et al. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[18] J. Liu, S. J. Wright, C. Ré, V. Bittorf, and S. Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. *Journal of Machine Learning Research*, 16(285-322):1–5, 2015.

[19] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[20] X. Pan et al. Scaling up correlation clustering through parallelism and concurrency control. *In DISCML workshop at NIPS*, 2014.

[21] X. Pan, J. E. Gonzalez, S. Jegelka, T. Broderick, and M. I. Jordan. Optimistic concurrency control for distributed unsupervised learning. In *NIPS*, pages 1403–1411. 2013.

[22] X. Pan, S. Jegelka, J. E. Gonzalez, J. K. Bradley, and M. I. Jordan. Parallel double greedy submodular maximization. In *NIPS*, pages 118–126, 2014.

[23] X. Pan, M. Lam, S. Tu, D. Papailiopoulos, C. Zhang, M. I. Jordan, K. Ramchandran, C. Re, and B. Recht. Cyclades: Conflict-free asynchronous machine learning. In *NIPS*. 2016.

[24] X. Pan, D. Papailiopoulos, S. Oymak, B. Recht, K. Ramchandran, and M. I. Jordan. Parallel correlation clustering on big graphs. In *NIPS*, pages 82–90. 2015.

[25] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701. 2011.

[26] J. Stamper, A. Niculescu-Mizil, S. Ritter, G. Gordon, and K. Koedinger. Challenge data set from kdd cup 2010 educational data mining challenge, 2010. [https://pslcdatashop.web.cmu.edu/KDDCup/].

[27] A. Talwalkar, T. Kraska, R. Griffith, J. Duchi, J. Gonzalez, D. Britz, X. Pan, V. Smith, E. Sparks, A. Wibisono, et al. Mlbase: A distributed machine learning wrapper. *Big Learning Workshop at NIPS*, 2012.

[28] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[29] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.