Automating Dependence-Aware Parallelization of Machine Learning Training on Distributed Shared Memory

Jinliang Wei
Carnegie Mellon University
jinlianw@cs.cmu.edu

Phillip B. Gibbons
Carnegie Mellon University
gibbons@cs.cmu.edu

Garth A. Gibson
Vector Institute, CMU and University of Toronto
garth.gibson@acm.org

Eric P. Xing
Petuum Inc., Carnegie Mellon University
eric.xing@petuum.com

Abstract

Machine learning (ML) training is commonly parallelized using data parallelism. A fundamental limitation of data parallelism is that conflicting (concurrent) parameter accesses during ML training usually diminishes or even negates the benefits provided by additional parallel compute resources. Although it is possible to avoid conflicting parameter accesses by carefully scheduling the computation, existing systems rely on programmer manual parallelization and it remains a question when such parallelization is possible.

We present Orion, a system that automatically parallelizes serial imperative ML programs on distributed shared memory. The core of Orion is a static dependence analysis mechanism that determines when dependence-preserving parallelization is effective and maps a loop computation to an optimized distributed computation schedule. Our evaluation shows that for a number of ML applications, Orion can parallelize a serial program while preserving critical dependences and thus achieve a significantly faster convergence rate than data-parallel programs and a matching convergence rate and comparable computation throughput to state-of-the-art manual parallelizations including model-parallel programs.

CCS Concepts • Computing methodologies → Machine learning; • Software and its engineering → Distributed memory; Just-in-time compilers; Source code generation;

Copyright © 2019 ACM, Inc. ACM Reference Format:

1 Introduction

Machine learning (ML) techniques have been successfully applied to a wide range of application domains including recommender systems [27], image classification [21, 28], topic modeling [8], just to name a few. The core of ML is a statistical model which is a set of parametric functions that serve inference queries, such as mapping an image to a label. ML training finds model parameter values that minimize (or maximize) a given objective function so the model best fits the observed data samples (i.e., the training dataset) based on certain criteria. Commonly used training algorithms repeatedly process the observed data samples, until the objective function stops improving, i.e., the algorithm has converged. Given a training dataset $D = \{D_i\mid 1 \leq i \leq N\}$ where $D_i$ denotes a mini-batch of one or multiple data samples, a serial training algorithm computes an update function $\Delta(A_i, D_i)$ for each mini-batch $D_i$ using the current parameter values $A_i$ and updates the parameters before processing the next mini-batch. Training algorithms typically take many passes (i.e. iterations) over the training dataset before they converge.

1.1 Data Parallelism

ML training is commonly parallelized using data parallelism. Under data parallelism, $K$ random data mini-batches $\{D_{i+K+k-1}\mid 1 \leq k \leq K\}$ are distributed to $K$ workers at the $i$-th time step. Each worker computes the update function $u_k = \Delta(A_i, D_{i+K+k-1})$ in parallel using the current parameter values $A_i$ and its assigned mini-batch $D_{i+K+k-1}$. The master copy of parameters is updated by aggregating refinements $\{u_k\mid 1 \leq k \leq K\}$ from all workers. The updated parameter values are then distributed to workers before they enter the next time step.
1.2 Dependence-aware Parallelization

In many ML applications, $\Delta$ reads only a subset of the model parameters and generates refinements to a (possibly different) subset of parameters. If each worker is assigned with a mini-batch $D_k^\ell$ such that the read-write sets of all $\{A_\ell, D_k^\ell\}$ computations are disjoint, then the parallel execution is serializable. That is, the parallel execution produces the same result as a serial execution following some sequential ordering of the mini-batches. We refer to this style of parallelization that preserves data dependence among mini-batches as dependence-aware parallelization. Fig. 1 compares data parallelism with dependence-aware parallelism. Note that under the dependence-aware parallelization shown, the parallel execution is equivalent to sequentially processing mini-batches $D_1, D_4, D_2,$ and $D_3$ (serializable), while under the shown data-parallelism, execution is not serializable.

STRADS [26] has exploited this property and demonstrates that training algorithms converge considerably faster compared to data parallelism when computation is scheduled to avoid conflicting parameter accesses (also referred to as model parallelism). However, STRADS requires programmers to manually parallelize the training algorithm, which demands significant programmer effort and is error-prone. In contrast, our system Orion automates dependence-aware parallelization of serial imperative ML programs for efficient distributed execution. Orion’s parallelization strategies are similar to STRADS but our focus is on automating dependence analysis and dependence-aware parallelization for serial imperative ML programs.

While imperative programming with a shared memory abstraction is highly expressive and natural for programmers, parallelization is more difficult compared to functional programming as dependency has to be inferred from memory accesses. Orion employs static dependence analysis and parallelization techniques from automatic parallelizing compilers and takes advantage of ML-specific properties to relax program semantics and thus improve parallelism. Semantic relaxations include programmer-controlled dependence violations, which enables data parallelism with few code changes. Moreover, Orion minimizes remote random access overhead via automated data partitioning and bulk prefetching based on the memory access pattern discovered in static analysis to achieve efficient distributed execution.

Experiments on a number of ML applications confirm that preserving data dependence can significantly improve ML training’s convergence progress and our proposed techniques are effective. We also compare Orion with various offline ML training systems [5, 26, 45] and show that Orion achieves much better or matching convergence progress and at least comparable computation throughput, even when compared with state-of-the-art manual parallelization, while substantially reducing programmer effort.

This paper makes three major contributions. First, we present a mechanism to automatically parallelize serial imperative ML programs for distributed training, with respect to data dependence. Our mechanism employs static dependence analysis and parallelization techniques from automatic parallelizing compilers, enhanced with ML-specific semantic relaxations. Second, we describe Orion, an end-to-end distributed system that efficiently implements the parallelization mechanism. Orion also features a new programming
abstraction that unifies dependence-aware parallelization and data parallelism and supports a wide range of ML applications. Third, we present an extensive experimental evaluation that demonstrates Orion parallelization’s effectiveness and competitive performance against other state-of-the-art offline ML training systems.

2 Motivation

In this section, we motivate the need for automation by discussing what dependence-aware parallelization of a real ML application entails, for example, when implementing it on STRADS [26], a state-of-the-art scheduler framework.

2.1 Matrix Factorization using SGD

Matrix factorization (MF) is a popular model used in recommender systems [27]. Given a large (and sparse) $m \times n$ matrix $V$ and a small rank $r$, the goal of MF is to find an $m \times r$ matrix $W$ and an $r \times n$ matrix $H$ such that $V \approx WH$, where the quality of approximation is defined by an application-dependent loss function $L$. A commonly used loss function in recommender systems is nonzero squared loss $L_{NZSL} = \sum_{i,j} \mathbb{1}(V_{ij} \neq 0) (V_{ij} - [WH]_{ij})^2$.

MF is often solved as an optimization problem using Stochastic Gradient Descent (SGD) that minimizes the loss function (i.e., the objective function). Note that $L_{NZSL}$ can be decomposed into the sum of local losses, i.e., $L_{NZSL} = \sum_{i,j} \mathbb{1}(V_{ij} \neq 0) l(V_{ij}, W_{ij}, H_{ij})$, where $l(V_{ij}, W_{ij}, H_{ij}) = (V_{ij} - W_{ij} H_{ij})^2$.

We denote a subset of the nonzero entries in $V$ as training set $Z$. With a step size $\epsilon$, an SGD algorithm for MF can be described in Alg. 1 ([19, 27]). Alg. 1 describes an abstract serial algorithm that is not bound to a particular system. Convergence of the algorithm is measured by a training loss defined over the training set $Z \subseteq V$, i.e., $L_{tr} = \sum_{i,j} \mathbb{1}(Z_{ij} \neq 0) l(Z_{ij}, W_{ij}, H_{ij})^2$.

Algorithm 1: SGD For Matrix Factorization

```
Input: the training set $Z$ and rank $r$
Output: factor matrices $W$ and $H$
Randomly Initialize $W$ and $H$

while not converged do
    for $Z_{ij} \in Z$ do
        $W'_{ij} \leftarrow W_{ij} - \epsilon \frac{\partial}{\partial W_{ij}} l(Z_{ij}, W_{ij}, H_{ij})$
        $H'_{ij} \leftarrow H_{ij} - \epsilon \frac{\partial}{\partial H_{ij}} l(Z_{ij}, W_{ij}, H_{ij})$
        $W_{ij} \leftarrow W'_{ij}$

    $L_{tr} \leftarrow \sum_{i,j} \mathbb{1}(Z_{ij} \neq 0) l(Z_{ij}, W_{ij}, H_{ij})^2$
end while
```

2.2 Parallelizing SGD Matrix Factorization

Similar to other iterative convergent ML algorithms, the heavy computation in SGD MF resides in the for-loop that iterates over the training set $Z$, which is desired to be parallelized. Implementations of SGD MF on parameter server

1Practical applications may employ regularization. Here we omit regularization for simplicity since it does not affect parallelization.

2STRADS is open-sourced here: https://github.com/sailing-pmls/strads (last visited: 1/10/2019). SGD MF is not part of the open-sourced repository and was obtained from STRADS authors.
neither static or dynamic analysis, nor code generation. Application programmers thus manually analyze data dependence and derive a computation schedule. While deriving an efficient computation schedule is most challenging, implementation is also highly non-trivial. SGD MF on STRADS is implemented as a coordinator and a worker program, totally consisting of 1788 lines of C++ code. The application program is responsible for coordination among workers, data partitioning, parameter communication and synchronization, etc. Due to STRADS’ low-level abstraction, there is little code reuse across STRADS applications.

In contrast, Orion performs static dependence analysis to find an efficient dependence-preserving parallelization and reuse the corresponding computation schedule. Orion abstracts away worker coordination by providing high-level primitives such as @parallel_for, map and groupBy. Moreover, Orion generates code for data loading, partitioning, and prefetching, tailored to specific data types, so that a computation schedule can be resued for different computation and data types without losing efficiency. Thus application programmers can focus on the core ML algorithm.

3 Orion Programming Model

Orion consists of a distributed runtime and an application library (Fig 3). Orion application programmers implement an imperative driver program that executes instructions locally and in Orion’s distributed runtime using the application library. Distributed programming in Orion seamlessly integrates with the rest of the program thanks to Orion’s distributed shared memory (DSM) abstraction and parallel for-loops. Our prototype implementation supports application programs written in Julia [7]. Julia is a scripting language that offers high programmer productivity like Python with great execution speed [3] using just-in-time compilation.

3.1 Distributed Arrays

Orion’s main abstraction for DSM is a set of multi-dimensional matrices, which we refer to as Distributed Arrays (or DistArrays). A DistArray can contain elements of any serializable type and may be either dense or sparse. A DistArray is partitioned and stored in the memory of a set of distributed machines in Orion’s runtime and Orion automatically repartitions DistArrays to minimize remote access overhead when executing distributed parallel for-loops.

Figure 4. Distributed parallel for-loop example

Elements of an $N$-dimensional DistArray are indexed with an $N$-tuple ($p_1, p_2, ..., p_n$). A DistArray supports random access via both point queries (e.g. $A[1, 3, 2]$) to access a single element and set queries (e.g. $A[1:3, 3, 2]$) where a range is specified for one or multiple DistArray dimensions. Here $[1, 3, 2]$ and $[1:3, 3, 2]$ are DistArray subscripts, analogous to DSM addresses. Statements that access DistArray elements can either execute locally or in Orion’s distributed workers by using the parallel for-loop primitive.

Similar to Resilient Distributed Datasets (RDD) [52], DistArrays can be created by loading from text files using a user-defined parser or by transforming an existing DistArray using operations like map and groupBy. Text file loading and map operations are recorded by Orion and not evaluated until the driver program calls materialize. This allows Orion to fuse the user-defined functions across operations and avoids memory allocation for intermediate results. Unlike RDDs, set operations that may cause shuffling, such as groupBy, are evaluated eagerly for simplicity.

Compared to RDD, DistArray supports indexed random accesses (i.e., point and set queries) and in-place updates, which makes DistArray better suited for holding trainable model parameters which are iteratively updated, especially when each mini-batch updates only a subset of the parameters. A DistArray is automatically distributed among a set of worker machines and can be sparse. At the lowest level, TensorFlow tensors are dense matrices that reside on a single device and TensorFlow applications may manually represent sparse and distributed matrices using dense tensors (e.g., [42]). While DistArray does not provide a rich set of linear algebra operations like TensorFlow tensors, a DistArray set query returns a Julia Array, which can leverage the rich set of linear algebra operations natively provided by Julia.

3.2 Distributed Parallel For-Loop

The driver program may iterate over the elements of an $N$-dimensional DistArray using a vanilla Julia for-loop. For example, the loop in Fig. 4 iterates over each element of DistArray $A$ where $e_{idx}$ is the element’s index and $e_{val}$ is the element’s value. As $A$ is an $N$-dimensional matrix, the for-loop is naturally an $N$-level perfectly nested loop and the DistArray represents the loop nest’s iteration space. Each DistArray element corresponds to a loop iteration and the element’s index $e_{idx}$ is the loop iteration’s index vector, of which each element is referred to as a loop index variable.

For-loops iterating over a DistArray can be parallelized across a set of distributed workers using a @parallel_for macro. Depending on the loop body’s access pattern to other
DistArrays, parallelization assigns iterations to workers and adds synchronization when it is needed for preserving data dependence among loop iterations (i.e., loop-carried dependence). Iterations that have dependences between them because of shared accesses on DistArrays are executed one after another in the correct order. Thus the parallel execution is equivalent to a serial execution of the loop (serializable).

Tools like OpenMP [13] and MATLAB parfor [4] also provide parallel fork–loop primitives, provided that the programmer asserts the for-loops have no dependency among its iterations. But Orion’s @parallel_for macro can be applied to loops that have dependences among iterations, and preserves loop-carried dependences. Moreover, Orion’s parallel for-loop executes in a distributed cluster while existing tools only apply to single machines.

Let $P = \{(p_1, p_2, \ldots, p_n) | \forall i \in [1, n] : 0 \leq p_i < s_i\}$ represent the iteration space of a $n$-dimensional DistArray, where $(p_1, p_2, \ldots, p_n)$ represents the index vector of an iteration, and the size of the iteration space’s $i$-th dimension is $s_i$. For any two iterations $\vec{p} = (p_1, p_2, \ldots, p_n)$ and $\vec{p}' = (p'_1, p'_2, \ldots, p'_n)$, Orion can parallelize the for-loop while preserving all loop-carried dependences if one of the following is true:

1. **1D Parallelization:** There exists a dimension $i$ such that when $p_i \neq p'_i$, there doesn’t exist any loop-carried dependence between iteration $\vec{p}$ and iteration $\vec{p}'$. Note that this also includes the case when there’s no dependence between any iterations.

2. **2D Parallelization:** There exist two dimensions $i$ and $j$ such that when $p_i \neq p'_i$ and $p_j \neq p'_j$, there doesn’t exist any loop-carried dependence between iteration $\vec{p}$ and iteration $\vec{p}'$.

3. **2D Parallelization w/ Unimodular Transformation:** When neither 1D nor 2D parallelization is applicable, in some cases (see Sec. 4.3), unimodular transformations [46] may be applied to transform the iteration space to enable 2D parallelization.

**Applicability.** Static parallelization requires the size of the iteration space to be constant and known at compile time. ML training applications usually iterate over a fixed input dataset or model parameters and Orion JIT compiles a for-loop after the iteration space DistArray is loaded or created. Orion’s dependence-aware parallelization strategies apply to for-loops when the loop body accesses only a subset of the shared memory addresses and the addresses can be fully determined given the loop index variables, i.e., the iteration-space DistArray index. More specifically, our current implementation accurately captures dependence when DistArray subscripts contain at most one loop index variable plus or minus a constant at each position. A more complex subscript is conservatively regarded as that it may take any value within the DistArray’s bounds. The loop body may inherit any driver program variable. The inherited variables are assumed to be read-only during a single loop execution but their values could change between different executions of the same loop.

ML applications commonly represent data records as a mapping from a $n$-tuple key to a value, i.e., $(k_1, k_2, \ldots, k_n) \rightarrow \text{value}$, where the key uniquely identifies the data record. Thus data records may be organized in a $n$-dimensional tensor, indexed by the key tuple. When parameter accesses are also indexed by the key tuple, parallelization via static dependence analysis is possible. For example, the popular bag-of-words model represents text as a set of mappings from a word to its number of occurrences. ML applications on text data often have parameters associated with each word, such as the word topic count vector in topic modeling with Latent Dirichlet Allocation or the word embedding vector, which are accessed based on word ID.

Deep neural network (DNN) training is an increasingly important class of ML workloads. The core computation of a typical DNN training program is a loop that iterates over data mini-batches where each iteration performs a forward pass, a backward pass and updates the neural network weights. DNNs commonly read and update all weights in each iteration, therefore serializable parallelization over mini-batches is not applicable. DNN training is most commonly parallelized with data parallelism, which can be achieved in Orion by permitting dependence violation as discussed in Sec. 3.3.

### 3.3 Distributed Array Buffers

Static dependence analysis avoids materializing a huge dependence graph whose size is proportional to the training dataset. Such a graph would be expensive to store and analyze. However, static dependence analysis requires the DistArray subscripts to be determined (as an expression of loop index variables and constants) statically to accurately capture the dependence among loop iterations.

First, some ML models, such as DNNs, perform dense parameter accesses. Second, while parameter accesses might be sparse in some models, the DistArray subscripts may depend on runtime values (e.g., `e_val` in Fig. 4). For example, in sparse logistic regression, processing a data sample reads and updates the weights corresponding to the sample’s nonzero features. In this case, traditional dependence analysis conservatively marks all DistArray positions as accessed, leading to false dependences among iterations and impeding parallelization. For these models, serializable parallelization can be severely limited in computation throughput or simply inapplicable, therefore such ML training applications are often parallelized with dependence violations. The algorithm converges better (closer to serial execution) when there are fewer collisions and when writes make small changes. In
order to support these applications, Orion application programmers may selectively exempt certain (or all) writes from dependence analysis using Distributed Array Buffers (or DistArray Buffers). By applying all writes to DistArray Buffers instead of DistArrays, an Orion application effectively resorts to data parallelism.

A DistArray Buffer is a write-back buffer of a DistArray, and provides the same API for point and set queries. A DistArray Buffer maintains a buffer instance on each worker, which is usually initialized empty. The application program may apply a subset of DistArray writes to a corresponding DistArray Buffer and exempt those writes from dependence analysis, making it possible to parallelize a for-loop that can’t be parallelized otherwise.

Typically the buffered writes are applied to the corresponding DistArray after the worker executes multiple for-loop iterations. The application program may optionally bound how long the writes can be buffered. Orion supports an element-wise user-defined function (UDF) for applying each DistArray Buffer’s buffered writes. This UDF is executed atomically on each DistArray element and thus supports atomic read-modify-writes. The UDF for applying buffered writes allows applications to define sophisticated custom logic for applying updates, and makes it easy to implement various adaptive gradient algorithms [15, 34, 44].

3.4 Putting Everything Together

Fig. 5 shows a Julia SGD MF program parallelized by Orion. The serial program has less than 90 lines of Julia code and can be parallelized by changing only a few lines. The parallel program creates DistArrays instead of local matrices for training data (ratings) and parameters (W and H) by loading from text files (text_file) or random initialization (randn). DistArrays can be manipulated with set operations, like map (e.g., line #9). The for-loops that iterate over the ratings matrix entries (e.g., line #14) are parallelized by applying the @parallel_for macro.

The parallel for-loop’s loop body may read any driver program variable that is visible to the loop (e.g., step_size) and the driver program may access the result of a parallel for-loop execution by reading from DistArrays or by using an accumulator (e.g., err). When an accumulator variable is created (e.g., line #12), an instance of this variable is created on each Orion worker, and the state of each worker’s accumulators are retained across for-loop executions. The driver program may aggregate the value of all workers’ accumulators using a user-defined commutative and associative operator (e.g., line #25).

4 Static Parallelization

Given Orion’s expressive programming model, in this section, we discuss how for-loops are parallelized and scheduled, along with various novel techniques to improve distributed execution throughput without programmer effort.

```plaintext
1 step_size = 0.01
2 # Omitted variable and function definitions
3 Orion.@dist_array ratings =
4   Orion.text_file(data_path, parse_line)
5 Orion.materialize(ratings)
6 dim_x, dim_y = size(ratings)
7 Orion.@dist_array W = Orion.randn(K, dim_x)
8 Orion.@dist_array W =
9   Orion.map(W, init_param, map_values=true)
10 Orion.materialize(W)
11 # Omitted: create DistArray H
12 Orion.accumulator err = Float32(0.0)
13 for iter = 1:num_iterations
14   Orion.@parallel_for for (key, rv) in ratings
15     W_row = @view W[:, key[1]]
16     H_row = @view H[:, key[2]]
17     # Omitted computing W_grad and H_grad
18     W[:, key[1]] -= W_row - W_grad * step_size
19     H[:, key[2]] -= H_row - H_grad * step_size
20   end
21   Orion.@parallel_for for (key, rv) in ratings
22     # Omitted: compute the predicted rating
23     err += abs2(rv - pred)
24   end
25   err = Orion.get_aggregated_value(:err, :) +
26   Orion.reset_accumulator(:err)
27
Figure 5. SGD Matrix Factorization Parallelized using Orion
4.1 Parallelization Overview

Orion’s @parallel_for primitive is implemented as a Julia macro, which is expanded when the for-loop is compiled. A Julia macro is a function that is invoked during compilation (as opposed to at runtime), which takes in an abstract syntax tree (AST) and produces a new AST to be compiled by the Julia compiler. Orion’s @parallel_for macro statically analyzes the for-loop’s AST to compute dependences among loop iterations based on the loop body’s access pattern to DistArrays. These dependences are represented as dependence vectors.

Based on the dependence pattern, Orion decides whether the for-loop is 1D or 2D parallelized and whether a unimodular transformation is needed (see Sec. 4.3). During macro expansion, Orion generates functions that perform the loop body’s computation and defines those functions in the distributed workers. According to the parallelization strategy, the generated new AST, that executes on driver, invokes a static computation schedule with the corresponding loop body functions. The generated AST also contains code that 1) repartitions relevant DistArrays to minimize remote access overhead; and 2) captures and broadcasts driver program variables that are inherited in the loop body’s scope. Note that even though the parallel for-loop may itself be inside of another for-loop and executed multiple times, the macro expansion and compilation is executed only once. A global
A lexicographically positive vector \( \mathbf{d} \) denotes a dependence vector of an \( n \)-loop nest if and only if there exist two dependent iterations \( p_1 \) and \( p_2 \) such that \( \mathbf{d} = \mathbf{p}_1 - \mathbf{p}_2 \). Infinity \( \infty \) (or positive/negative infinity, \( +\infty/-\infty \)) in dependence vectors means that the dependence vector may take any (positive or negative) integer value at that position. In Fig. 6, dependence vector \( (0, \infty) \) means that any iteration \( (p'_1, p'_2) \) depends on iteration \( (p_1, p_2) \) as long as \( p'_1 - p_1 = 0 \). A dependence vector implies a dependence pattern shared by all iterations, yielding a concise dependence representation. However, dependence vectors may conservatively represent a dependence that exists for only certain iterations as a dependence for all iterations, unnecessarily limiting parallelism.

Previous work discussed how to compute dependence vectors [24, 33]. An iteration depends on another (earlier) iteration if and only if they both access the same memory location and at least one of the accesses is a write. In general, computing dependence vectors requires performing a dependence test on the subscripts of each pair of DistArray references from two different iterations, and either prove independence or produce a dependence vector for the loop indices occurring in the scripts [24]. Since Orion currently supports accurate dependence capturing only for subscripts that contain at most one loop index variable plus or minus a constant at each position, we can simplify the algorithm. We represent each subscript as a 3-tuple \( (\text{dim}_\text{idx}, \text{const}, \text{stype}) \), representing the loop index variable’s dimension index in the iteration space, the constant and the type of the subscript, i.e., whether it is a single value or a range and whether the subscript is supported for dependence analysis. Alg. 2 presents Orion’s core procedure for computing dependence vectors. Our algorithm produces at most one dependence vector from each pair of static DistArray references. Two DistArray references are independent when they are both read, and write-write dependence may be omitted when the loop iterations can be executed in any order (unordered_loop). After skipping such reference pairs, we initialize a dependence vector whose elements are infinity, meaning that any two iterations may be dependent due to these two DistArray references. We then refine this conservative dependence by checking each subscript position. We declare the two references are independent if their subtitles will never match. In the end, we add the dependence vector to the set of dependence vectors after making sure it is lexicographically positive. The algorithm has a time complexity of \( O(N^2 \times D) \) for each referenced DistArray where \( N \) is the number of static DistArray references and \( D \) is the number of dimensions of the referenced DistArray.

### 4.3 Parallelization and Scheduling

Orion partitions the iteration space based on dependence vectors so that different partitions can be executed in parallel. Each worker is assigned a number of iteration space partitions and synchronizes at most once per partition.
Parallelization of a 4 x 4 iteration space. Ellipses denote loop iterations and edges denote dependence between iterations. Note that representing the dependence in (a) requires only 1 dependence vector, namely (0, 1), and representing the dependence in (b) and (c) requires only 2 dependence vectors, namely (1, 0) and (0, 1). Iterations of the same color are executed in parallel. Rectangles denote iteration space partitions. Workers are denoted as w0, w1, etc. M and N denote the number of unique time-dimension (vertical) and space-dimension (horizontal) indices. Although it’s not shown here, typically each worker is assigned with multiple space-dimension indices for better load balancing and multiple time-dimension indices for pipelined parallelism (Sec. 4.4).

Given the set of dependence vectors D, if there exists a dimension i such that \( \forall \vec{d}' = (d_1, d_2, ..., d_n) \in D, d_i = 0 \), then any two iterations \( \vec{p} = (p_1, p_2, ..., p_n) \) and \( \vec{p}' = (p'_1, p'_2, ..., p'_n) \) are independent as long as \( p_i \neq p'_i \). Partitioning the iteration space by dimension \( i \) ensures that any two iterations \( \vec{p} \) and \( \vec{p}' \) from two different partitions are independent. Thus the loop can be scheduled by assigning different iteration space partitions to different workers as there’s no data dependence across partitions. This is referred to as 1-dimensional (i.e. 1D) parallelization. Note that all such dimensions \( i \) that satisfy the above condition are candidate partitioning dimensions. Fig. 7a shows an example that applies 1D parallelization to a 2-level loop nest and partitions the 2D iteration space by dimension \( j \). The corresponding compute schedule is shown in Fig. 7d. The workers synchronize with each other after executing all iterations in its assigned partition.

If there exist two dimensions \( i \) and \( j \) such that \( \forall \vec{d}' = (d_1, d_2, ..., d_n) \in D, d_i = 0, d_j = 0 \), then any two iterations \( \vec{p} = (p_1, p_2, ..., p_n) \) and \( \vec{p}' = (p'_1, p'_2, ..., p'_n) \) are independent as long as \( p_i \neq p'_i \) and \( p_j \neq p'_j \). In this case, the loop can be parallelized by partitioning the iteration space by dimensions \( i \) and \( j \), which we refer to as 2-dimensional (i.e. 2D) parallelization (see Fig. 7b). The partitions are assigned to workers based on one of the dimensions, e.g. \( j \) in this case, which we refer to as the space dimension and the other dimension is referred to as the time dimension. The computation is executed in a sequence of global time steps. Within each time step, multiple workers may execute a local partition in parallel, where the partition’s time dimension index is derived from the time step number to ensure that all parallel partitions’ indices differ in both space and time dimensions. We observe that a partition depends on only two other iteration space partitions from the previous time step and one of them belongs to the same worker. Thus a worker waits for a signal from a single predecessor worker to begin the next time step instead of a global synchronization barrier.

Relaxing the ordering constraints. Traditional automatic parallelizing compilers preserve the lexicographical ordering of loop iterations and thus dependences indicate the execution ordering of dependent loop iterations, such as shown in Fig. 7b. With the ordering constraints, simultaneous execution of two iterations might not be possible even when they do not access the same memory location. For example, in Fig. 7b, even though they do not access the same memory location, iteration (3, 1) cannot be executed in parallel with (0, 0) due to the ordering constraints enforced by (3, 0).

Many ML algorithms such as Gibbs sampling do not require a particular ordering in which data samples or mini-batches are processed. Other algorithms such as stochastic
Algorithm 2: Computing dependence vectors

```
input : refs - the list of references on DistArray D
output: dvecs - the set of dependence vectors due to references to D

dvecs = EmptySet();
for each unique pair ref_a and ref_b in refs do
  if (ref_a.is_read and ref_b.is_read) or (unordered_loop and ref_a.is_write and ref_b.is_write) then
    continue;
dvec = Vec(iter_space.num_dims, inf);
independent = false;
for dim ∈ D.dims do
  sub_a = ref_a.subs[dim];
  sub_b = ref_b.subs[dim];
  if sub_a and sub_b contains a single loop index variable then
    if sub_a.dim_idx == sub_b.dim_idx then
      dist = sub_a.const - sub_b.const;
    if dvec[sub_a.dim_idx] != inf and dvec[sub_a.dim_idx] != dist then
      independent = true;
      break;
    dvec[sub_a.dim_idx] = dist;
  else
    continue;
  else
    Test dependence for other subscript types;
if not independent then
  correct dvec for lexicographical positiveness;
  dvecs = union(dvecs, {dvec});
```

gradient descent usually randomly shuffle the dataset before or during training. For such ML algorithms, even though different iteration ordering may result in different numerical values and thus affect the convergence process, enforcing a particular ordering, such as the lexicographical ordering, is not necessarily beneficial but sacrifices parallelism. Therefore, Orion’s parallelization by default ensures only serializability but not the lexicographical ordering. Applications may enforce ordering by using the ordered argument in @parallel_for. Relaxing the ordering constraints allows Orion to reorder iterations to maximize parallelism: Orion schedules workers to start from different indices along the time dimension to fully utilize all workers (Fig. 7c and Fig. 7f).

Unimodular transformation. When neither 1D or 2D parallelization can be directly applied, Orion may apply unimodular transformations on the iteration space, when the dependence vectors contain only numbers or positive infinity, to enable 2D parallelization. Parallelizing for-loops using unimodular transformations was introduced by Wolf et. al [46]. The

set of dependence vectors after unimodular transformation denoted as $\mathcal{D}'$ satisfy that $\forall \vec{d} = (d_1, d_2, ..., d_n) \in \mathcal{D}' : d_i > 0$ (all dependences are carried by the outermost loop). With the transformed loop nest denoted as $L_1, L_2, ..., L_n$, there’s no dependence between iterations of the innermost loop nest $L_2, L_3, ..., L_n$ in the same outermost loop $L_1$. Thus the for-loop can be parallelized by partitioning the transformed iteration space by the outermost dimension and any combination of the inner loop dimensions. By reversing the transformation, we can derive a 2D partitioning of the original iteration space.

As multiple candidate partitioning dimensions may exist, Orion uses a simple heuristic to choose the partitioning dimension(s) among candidates that minimizes the number of DistArray elements needed to be communicated among Orion workers during loop execution. This heuristic can be overridden by the application program.

Dealing with Skewed Data Distribution. As the parallel for-loop’s iteration space is often sparse and the data distribution is often skewed (for example, when iterating over a skewed dataset), partitioning the iteration space into equal-sized partitions results in imbalanced workload among workers. Orion DistArrays support a `randomize` operation that randomizes a DistArray along one or multiple dimensions to achieve a more uniform data distribution. Further more, Orion computes a histogram along each partitioning dimension to approximate the data distribution, which is used to generate a more balanced partitioning.

Fault tolerance. An Orion driver program can checkpoint a DistArray by writing it to disk, which is eagerly evaluated. For ML training, a common approach is to checkpoint the parameter DistArrays every $N$ data passes.
4.4 Reducing Remote Random Access Overhead

Generally, DistArray random access can be served by a parameter server. However, in this case, each random access potentially result in a remote access over the inter-machine network. The overhead of network communication is significant even when Orion workers cache DistArray values and buffer DistArray writes.

Locality and pipelining. Usually different workers read and write to disjoint subsets of elements of a DistArray. If the workers’ read/write sets are disjoint range partitions of a DistArray, the DistArray may be range partitioned among workers so random access to it can be served locally.

Under 2D parallelization, the DistArray range partition accessed by a worker may be different at different time steps and a worker has to wait to receive a DistArray partition from its predecessor before starting a new time step. When the ordering constraints can be relaxed (Fig. 7f), Orion avoids the workers’ idle waiting time by creating multiple time-dimension partition indices per worker and letting the worker proceed to a locally available time-dimension partition index while waiting for data from its predecessor, as illustrated in Fig. 8.

Bulk prefetching. If the same elements of a DistArray are simultaneously accessed by different workers, for example, when it is updated by a DistArray Buffer, or the disjoint sets of elements cannot be obtained from efficiently partitioning the DistArray, the DistArray is served by a number of server processes, similar to a Parameter Server. In this case, in order to minimize the random remote access overhead, Orion prefetches DistArray reads in bulk.

In order to accurately determine which values to prefetch, existing Parameter Server systems rely on programmers to implement a “virtual iteration” besides the actual computation to provide the parameter access pattern [12] or to manually implement prefetching and cache management [29]. Orion automates bulk prefetching by synthesizing a function that generates the list of DistArray element indices that are read during the loop body computation. The generated function executes loop body statements that read from non-locally allocated DistArrays, but instead of reading DistArray elements and performing computation, those statements are transformed to only record the DistArray subscript value. Since the DistArray subscripts may depend on runtime values, such as loop index variable and driver program variables (which are captured and broadcasted to workers as read-only variables), the function also executes statements that the DistArray subscripts have a data or control dependence on with proper control flow and ordering. If a DistArray subscript depends on values read from DistArrays, computing it may incur an expensive remote access. Therefore, DistArray subscripts that depend on other DistArray values are not recorded for bulk prefetching. The code generation algorithm is in spirit similar to dead code elimination.

5 Offline ML Training Systems

In this section, we review and compare existing offline ML training systems (Table 1) with Orion, with an emphasis on their programming model and parallelization strategy. We focus on dataflow systems and graph processing systems, which present two distinct programming models.

5.1 Batch Dataflow Systems and TensorFlow

Many systems [5, 36, 51, 52] adopt a dataflow execution model, where the application program constructs a directed acyclic graph (DAG) that describes the computation and the computation DAG is lazily evaluated only when certain output is requested. A popular system among them is Spark [52], in which each node of the DAG represents a set of data records called a Resilient Distributed Dataset (RDD) and the edges represent transformation operations that transform one RDD to another. A fundamental limitation of traditional dataflow systems is that their computation DAG does not allow mutable states in order to ensure deterministic execution, which makes updating model parameters an expensive operation. For example, mutable states in Spark such as driver local variables or accumulators, are not represented in the computation graph and are stored and updated by a single driver process. SparkNet [35] represents model weights as driver program local variables, which are broadcasted to workers to compute new weights. The new weights produced by workers are collected and averaged by the driver. Each broadcast and collection takes about 20 seconds [35].

TensorFlow [5] is a deep learning system which also adopts the dataflow programming model, where nodes of the computation DAG represent operations whose inputs and outputs are tensors flowing along the edges. TensorFlow introduces mutable states such as variable and queue into the computation graph to efficiently handle model parameter updates. A typical TensorFlow program constructs a DAG that implements the update operation processing a single mini-batch of data, where trainable model parameters are represented as variables. One approach to represent different mini-batch’s or data sample’s access pattern on invidiudal model parameters is to represent each mini-batch (or data sample) and model parameter as separate nodes in the DAG (i.e., statically unroll the whole loop), resulting in a huge DAG that’s expensive to store and analyze.

Alternatively, the computation can be described as a while-loop [50] iterating over mini-batches or data samples. While TensorFlow while-loop allows different iterations to be executed in parallel, each operation is assigned with, and bound to, a single computing device (different operations can be placed on different devices). In other words, a TensorFlow while-loop does not partition its iteration space among distributed devices and may fail to exploit the full parallelism enabled by the loop. On the other hand, TensorFlow while-loop enables additional parallelism for loops with a large and
complex loop body (e.g., a multi-layer RNN), since the loop body can be distributed among multiple computing devices. Moreover, TensorFlow while-loop dynamically computes the loop termination condition and supports data-dependent control flow inside the loop body including nested loops.

5.2 Graph Processing Systems

Graph processing systems [10, 20, 31, 32, 49, 53, 54] take a user-provided data graph as input and execute a vertex program on each graph vertex. Since a vertex program is restricted to access only data stored on that vertex itself, its edges or its neighboring vertices, the graph naturally describes the vertex program’s data dependence on mutable states. This property allows some systems to schedule independent vertex computation and ensure serializability by using graph coloring or pessimistic concurrency control [20, 31, 32]. However, graph coloring is an NP-complete problem and is expensive to perform; and with pessimistic concurrency control, lock contention may heavily limit the system’s scalability as demonstrated by a weak scaling experiment on PowerGraph [20]. As a result, recent graph processing systems have given up serializability: their vertex program either executes asynchronously or synchronizes with Bulk Synchronous Parallel synchronization [10, 49, 53, 54], both violating dependence among vertices.

6 Experimental Evaluation

Orion is implemented in \(~17,000\) lines of C++ and \(~6,300\) lines of Julia (v0.6.2), and has been open sourced. In this section, we evaluate Orion, focusing on parallelization effectiveness and execution efficiency. Our experiments were conducted on a 42-node cluster where each machine contains an Intel E5-2698V3 Xeon CPU and 64GiB of memory. Each CPU contains 16 cores with hyper-threading. These machines are connected with 40Gbps Ethernet.

6.1 Evaluation Setup and Methodology

We are interested in answering the following questions through experimental evaluation:

1. Is the algorithms’ convergence rate sensitive to data dependence? Can dependence violation (such as data parallelism) significantly slow down algorithm convergence? Previous work (e.g., STRADS [26]) demonstrated that data dependence may have critical impact on algorithmic convergence and our results confirm their observations.
2. Can proper semantic relaxations such as relaxing the loop ordering constraints and violating non-critical dependences indeed improve computation throughput without jeopardizing convergence?
3. While preserving critical dependences, can Orion parallelization effectively speed up the computation throughput and thus overall convergence rate of serial Julia ML programs?
4. Do Orion applications achieve higher or competitive computation throughput and convergence rates compared to applications on other state-of-the-art offline ML training systems, including both manually parallelized data- and model-parallel programs?

ML applications. We’ve implemented a number of ML applications on Orion, exercising different parallelization strategies, as summarized in Table 2. In this section, we focus on evaluating performance for SGD MF (w/o and w/ AdaRev) and LDA, which are commonly used benchmark applications and allow us to compare Orion with other systems.

Datasets. We evaluated SGD MF (w/o and w/ AdaRev) on the Netflix dataset [1] for movie recommendation, which contains \(~100\) million movie ratings (rank is set to 1000). We evaluated LDA on a smaller NYT Times dataset that contains \(~300\) thousand documents and a subset of the large ClueWeb dataset [2] that contains \(~25\) million documents (32GB) (number of topics is set to 1000 and 400 respectively).

Metrics. Ultimately ML training applications desire to reach a high model quality in the least amount of time, which we refer to as overall convergence rate. A high overall convergence rate requires the training system to both process a large number of data samples per second, i.e., achieve a high computation throughput, and improve the model quality by a large margin per data pass, i.e., achieve a high per-iteration convergence rate. A serial execution typically achieves the best per-iteration convergence rate and thus serves as a gold standard. Different parallelizations may have different per-iteration convergence rates depending on whether and which data dependences are violated. Our evaluation metrics include both overall and per-iteration convergence rate to properly attribute the performance differences.

ML systems in comparison. We compared Orion with a number of state-of-the-art ML offline training systems on SGD MF (w/ and w/o AdaRev) and LDA in terms of both

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataflow</td>
<td>Spark [52], DryadLINQ [51]</td>
<td>No</td>
<td>dataflow</td>
<td>Scala, Java, Python</td>
</tr>
<tr>
<td>Parameter Server</td>
<td>parameter server [29], Bösen [45]</td>
<td>Yes</td>
<td>imperative</td>
<td>C++</td>
</tr>
<tr>
<td>PS w/ scheduling</td>
<td>STRADS [26]</td>
<td>Yes</td>
<td>imperative</td>
<td>C++</td>
</tr>
<tr>
<td>Graph Processing</td>
<td>PowerGraph [20], PowerLyra [10]</td>
<td>Limited</td>
<td>vertex programming</td>
<td>C++</td>
</tr>
<tr>
<td>Orion</td>
<td></td>
<td>Yes</td>
<td>imperative</td>
<td>Julia</td>
</tr>
</tbody>
</table>

Table 1. Comparing different systems for offline machine learning training.

URL: https://github.com/jinliangwei/orion
computation throughput and overall convergence rate. The systems that we experimentally compared to include Bösen parameter server [45], STRADS and TensorFlow.

TuX\(^2\) [49] is a recently proposed graph processing system, particularly optimized for ML training workloads. TuX\(^2\) was reported to have over an order of magnitude faster per-iteration time on SGD MF compared to PowerGraph [20] and PowerLyra [10]. With a rank of 50, TuX\(^2\) SGD MF\(^7\) takes \(-0.7\) seconds to perform one data pass on the Netflix dataset [1] using 8 machines, each with two Intel Xeon E5-2650 CPUs (16 physical cores), 256GiB of memory, and a Mellanox ConnectX-3 InfiniBand NIC with 54Gbps bandwidth (all higher than ours except for slight slower CPUs). In contrast, Orion SGD MF achieves a per-iteration time of \(-1.4\) seconds on 8 machines with the same number of CPU cores. On the other hand, with a carefully tuned mini-batch size, TuX\(^2\) SGD MF reaches a nonzero squared loss (lower is better) of \(-7 \times 10^{10}\) in \(-600\) seconds using 32 machines in its best case, while Orion SGD MF reaches \(-8.3 \times 10^{7}\) in \(-68\) seconds using only 8 machines. Even though TuX\(^2\) SGD MF achieves a higher computation throughput, its overall convergence rate is much lower than Orion’s due to violating data dependence.

### 6.2 Summary of Evaluation Results

1. Preserving data dependence is critical for SGD MF (w/o and w/ AdaRev) and LDA. Dependence-violating parallelization (i.e., data parallelism) takes many more data passes than serial execution to reach the same

\(^7\)TuX\(^2\) is not open sourced

### 6.3 Parallelization Effectiveness

We compare Orion-parallelized Julia programs with serial Julia programs in terms of both computation throughput
(i.e., time per iteration) and per-iteration convergence rate (Fig. 9). As shown in Fig. 9a, although the Orion abstraction incurs some overhead, Orion parallelization outperforms the serial Julia programs using only two workers and enables consistent speedup up to 384 workers. Although Orion’s parallelization relaxes the loop ordering constraints for both SGD MF and LDA, and violates some non-critical dependences in LDA, preserving (critical) dependences enable Orion parallelization to achieve a matching convergence rate to serial execution (Fig. 9b and Fig. 9c). On the other hand, data parallelism (using Bösen) converges substantially slower than serial execution due to violating dependences freely.

Table 3 compares ordered and unordered 2D parallelization in terms of computation throughput. Theoretically, relaxing the loop ordering constraints at most doubles parallelism. But it also enables a more efficient communication scheme (see section 4.4) that hides the communication latency, achieving an over 2× speedup. Fig. 9b and Fig. 9c show that loop ordering makes negligible differences in convergence rate. While we observe a bigger difference when adaptive revision [34] is used, relaxing the loop constraints is still beneficial for the improved computation throughput.

**Bulk Prefetching.** When training SLR using SGD, each data sample reads and updates a number of weight values corresponding to the nonzero features of the data record, which is unknown until the data sample is processed. The sequence of DistArray reads causes a sequence of inter-process communication, possibly over inter-machine networks. In a single-machine experiment using the KDD2010 (Algebra) [18] dataset, each data pass takes 7682 seconds, wasting most of the time on communication. Orion automatically synthesizes a function to prefetch the needed DistArray values in bulk (see Section 4.4) and thus reduces the per-iteration time to 9.2 seconds. It can be further reduced to 6.3 seconds by caching the prefetch indices.

### 6.4 Comparison with Other Systems

**Manual data parallelism.** Under data parallelism, Bösen workers synchronize after processing the entire local data partition. While achieving a high computation throughput, data-parallel applications on Bösen converge considerably slower than Orion-parallelized programs.

**Data parallelism w/ communication management.** Bösen features a communication management (CM) mechanism that improves the convergence rate of data-parallel training. Given a bandwidth budget, CM proactively communicates parameter updates and fresh parameter values before the synchronization barrier, when spare network bandwidth is available, to reduce the error due to violating data dependence. Moreover, CM prioritizes large updates to more effectively utilize the limited bandwidth budget. We assign each Bösen machine a bandwidth budget of 1600Mbps and
We present Orion, a system that parallelizes ML programs based on static data dependence and unifies various parallelization strategies under a clean programming abstraction. Orion achieves better or competitive performance compared to state-of-the-art offline ML training systems while sacrificing accuracy. Sampson et al. [41] rely on profiling to determine the best heuristics to make reasonable decisions while auto-tuning can be expensive.
Acknowledgments
We thank Andy Pavlo, Aaron Harlap, Abutalib Aghayev and Qing Zheng for their comments on earlier drafts of the paper. We thank Jin Kyu Kim for providing us the STRADS SGD MF source code. We thank our shepherd Yuan Yu and the anonymous EuroSys’19 reviewers for their valuable and constructive feedback. We also thank the member companies of the PDL Consortium (Alibaba, Broadcom, Dell EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, Micron, Microsoft, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma, Toshiba, Veritas, and Western Digital) for their interest, insights, feedback, and support. This research is supported in part by National Science Foundation under awards CCF-1629559 and CCF-1725663. Part of the work was done when Jinliang was visiting the Vector Institute.

References


