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

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

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

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

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*;

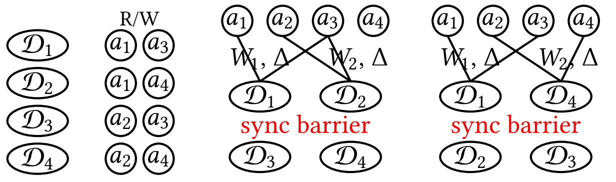
ACM Reference Format:
Jinliang Wei, Garth A. Gibson, Phillip B. Gibbons, and Eric P. Xing. 2019. Automating Dependence-Aware Parallelization of Machine Learning Training on Distributed Shared Memory. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3302424.3303954

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 $\mathcal{D} = \{\mathcal{D}_i | 1 \leq i \leq N\}$ where $\mathcal{D}_i$ denotes a mini-batch of one or multiple data samples, a serial training algorithm computes an update function $\Delta(A_t, \mathcal{D}_i)$ for each mini-batch $\mathcal{D}_i$ using the current parameter values A_t 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 $\{\mathcal{D}_{i*K+k-1} | 1 \leq k \leq K\}$ are distributed to K workers at the i-th time step. Each worker computes the update function $u_k = \Delta(A_t, \mathcal{D}_{i*K+k-1})$ in parallel using the current parameter values A_t and its assigned mini-batch $\mathcal{D}_{i*K+k-1}$. The master copy of parameters is updated by aggregating refinements $\{u_k | 1 \leq k \leq K\}$ from all workers. The updated parameter values are then distributed to workers before they enter the next time step.


R/W
$\mathcal{D}_1$ a_1 a_3
$\mathcal{D}_2$ a_1 a_4
$\mathcal{D}_3$ a_2 a_3
$\mathcal{D}_4$ a_2 a_4
a_1 a_2 a_3 a_4
W_1, Δ
W_2, Δ
sync barrier
sync barrier

(a) Read-write sets **(b)** Data parallelism **(c)** Dep-aware

Figure 1. Data parallelism vs. dependence-aware parallelism: (a) the read-write (R/W) sets of data mini-batches $\mathcal{D}_1$ to $\mathcal{D}_4$; (b) in data parallelism, mini-batches are randomly assigned to workers, leading to conflicting parameter accesses; (c) in dependence-aware parallelization (note that D_4 instead of D_2 is scheduled to run in parallel with D_1), mini-batches are carefully scheduled to avoid conflicting parameter accesses.

A key problem of data parallelism is that it is not equivalent to serial execution because a worker does not observe the other concurrent workers' parameter updates produced at the same time step. Compared to a serial execution, under data parallelism, a worker computes Δ using a stale version of model parameter values, violating data dependence.

Non-serializable execution often leads to slower algorithm convergence and lower model quality, therefore data parallelism is not always the best parallelization method. We can understand the effect of such non-serializable parallelization from two perspectives. First, for stochastic gradient descent (SGD), synchronous data parallelism over K workers is equivalent to sequential SGD using a mini-batch size of K times larger. Mini-batch size is a SGD hyperparameter and a mini-batch size that is too large often requires more data passes to reach the same model quality and may also lead to lower model performance on unseen data. Previous work reported this effect for both traditional ML models [26] and neural networks [23, 25]. Second, generally speaking, non-serializable parallelization is an erroneous execution of the sequential algorithm, where parameter values contain error due to conflicting accesses. Intuitively, the error's magnitude increases when more workers are used and decreases when workers synchronize more frequently. Thanks to ML algorithms' tolerance to bounded error [22, 39], the erroneous execution may still produce an acceptable model but the algorithm's convergence rate and model quality degrades as the the error increases [22, 26, 45]. Large mini-batch size or synchronization once per mulitple mini-batches is common in distributed training in order to ammortize synchronization overhead. This is especially common for traditional ML models where per-data-sample computation is light.

1.2 Dependence-aware Parallelization

In many ML applications, Δ 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 $\mathcal{D}'_k$ such that the read-write sets of all $\Delta\left(A_t, \mathcal{D}'_k\right)$ 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 $\mathcal{D}_1$, $\mathcal{D}_4$, $\mathcal{D}_2$, and $\mathcal{D}_3$ (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. Sematic relaxations include programmer-controlled dependence violation, 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* $L_{NZSL} = \sum_{i,j:V_{ij} \neq 0}(V_{ij} - [WH]_{ij})^2$.

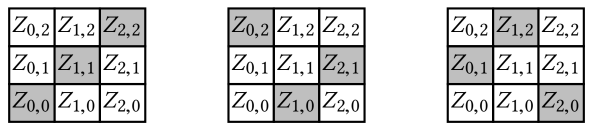
MF is often solved as an optimization problem using Stochastic Gradient Descent (SGD) that minimizes the loss function (i.e., the objective function). Note that L_{NZSL} can be decomposed into the sum of local losses, i.e., $L_{NZSL} = \sum_{i,j:V_{ij} \neq 0} l(V_{ij}, W_{i*}, H_{*j})$, where $l(V_{ij}, W_{i*}, H_{*j}) = (V_{ij} - W_{i*}H_{*j})^2$. We denote a subset of the nonzero entries in V as training set Z. With a step size ϵ, an SGD algorithm for MF can be described in Alg. 1 [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., $L_{tr} = \sum_{i,j:Z_{i,j} \in Z} l(Z_{ij}, W_{i*}, H_{*j})^2$.

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
        $W'_{i*} \leftarrow W_{i*} - W_{i*}\epsilon \frac{\partial}{\partial W_{i*}} l(Z_{ij}, W_{i*}, H_{*j})$
        $H_{*j} \leftarrow H_{*j} - H_{*j}\epsilon \frac{\partial}{\partial H_{*j}} l(Z_{ij}, W_{i*}, H_{*j})$
        $W_{i*} \leftarrow W'_{i*}$
```

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 systems [11, 45] and graph processing systems [10, 20, 49] are often parallelized using data parallelism, where the training set Z is randomly partitioned and assigned to workers. Random partitioning leads to conflicting accesses on W or H and violating data dependence, e.g., if data samples Z_{ip} and Z_{iq}, both reading and writing W_{i*}, are assigned to different workers and processed in the same time step.



$Z_{0,2}$ $Z_{1,2}$ $Z_{2,2}$
$Z_{0,1}$ $Z_{1,1}$ $Z_{2,1}$
$Z_{0,0}$ $Z_{1,0}$ $Z_{2,0}$

(a) Time step 1 **(b)** Time step 2 **(c)** Time step 3

Figure 2. Stratified SGD taking a full pass over Z: Z is partitioned into 3 strata, which each corresponds to a unique time step. Each stratum consists of 3 range-partitioned blocks. While strata are processed sequentially, blocks within a stratum can be processed in parallel without violating any dependence.

We may observe two data samples Z_{ij} and $Z_{i'j'}$, $\forall i,j,i',j'$: $i \neq i', j \neq j'$, are independent. That is, processing Z_{ij} and $Z_{i'j'}$ does not read or write to the same entries in W or H. We can devise a serializable parallelization by processing only independent data samples in parallel. Although different orderings of data samples may indeed lead to different numerical values of W and H, serializability is often sufficient for matching sequential execution's convergence rate and model quality. Based on this observations, Gemulla et al. [19] proposed a serializable, parallel SGD algorithm called stratified SGD, which partitions the training dataset Z (i.e., the iteration space) into a sequence of strata. The strata are processed sequentially but blocks within a stratum are processed in parallel. A 3×3 partitioned matrix Z and the corresponding stratified SGD execution is depicted in Fig 2.

Generally, with manual parallelization, programmers identify the data dependences among loop iterations based on how they access shared memory and devise a computation schedule. A computation schedule breaks down the iteration space (e.g., Z) into partitions, which conceptually form a dependency graph. An ideal partitioning provides sufficient parallelism (i.e., many partitions can be executed in parallel) while amortizing synchronization overhead (i.e., partitions are large enough). The computation schedule also assigns partitions to workers. Dependencies among iteration space partitions incur synchronization among workers and network communication. Partition assignment affects synchronization frequency and communication volume.

STRADS [2] [26] is a scheduler framework for traditional model-parallel ML programs, whose applications, such as SGD MF and topic modeling (LDA) achieve state-of-the-art convergence rate. Compared to Orion, STRADS performs

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

[2]STRADS 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.

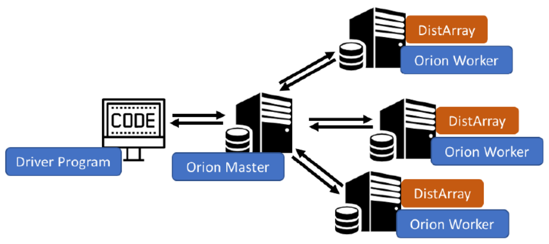


Figure 3. Orion System Overview

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 constrast, 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.

```
Orion.@parallel_for for (e_idx, e_val) in A
    ...loop body...
end
```

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 a 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 [3] 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 for-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 $\mathcal{P} = \{(p_1, p_2, ..., p_n) | \forall i \in [1, n] : 0 \leq p_i < s_i\}$ represent the iteration space of a n-dimensional DistArray, where $(p_1, p_2, ..., p_n)$ 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 $\vec{p} = (p_1, p_2, ..., p_n)$ and $\vec{p'} = (p'_1, p'_2, ..., p'_n)$, 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 data set 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 [4] 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, ..., k_n)` → `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 propotional 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

[3]They both access the same DistArray element and at least one of the accesses is a write.

[4]The loop body may still write to those variables but the new value is visible only to the worker that performs the write.

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 (`randomn`). 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' accmulators using a user-defined commutative and associative operator (e.g. line #25).

```
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 end
```

Figure 5. SGD Matrix Factorization Parallelized using Orion

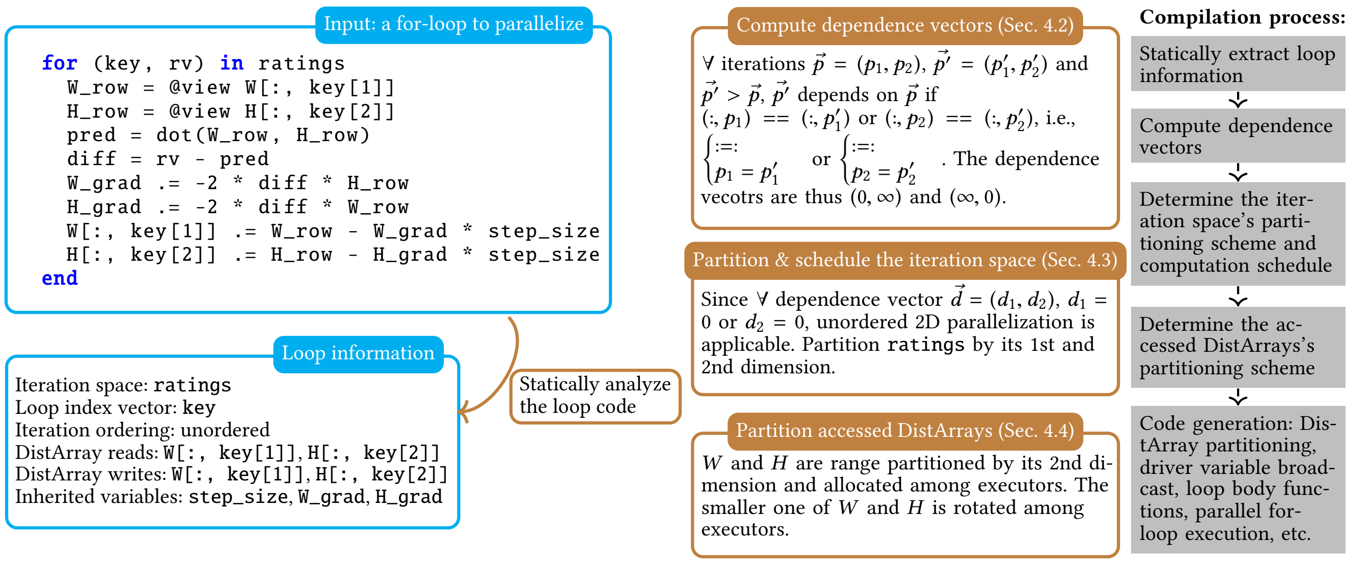
4 Static Parallelization

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.

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



Input: a for-loop to parallelize

```
for (key, rv) in ratings
  W_row = @view W[:, key[1]]
  H_row = @view H[:, key[2]]
  pred = dot(W_row, H_row)
  diff = rv - pred
  W_grad .= -2 * diff * H_row
  H_grad .= -2 * diff * W_row
  W[:, key[1]] .= W_row - W_grad * step_size
  H[:, key[2]] .= H_row - H_grad * step_size
end
```

Statically analyze the loop code

Loop information

Iteration space: `ratings`
Loop index vector: `key`
Iteration ordering: unordered
DistArray reads: `W[:, key[1]]`, `H[:, key[2]]`
DistArray writes: `W[:, key[1]]`, `H[:, key[2]]`
Inherited variables: `step_size`, `W_grad`, `H_grad`

Compute dependence vectors (Sec. 4.2)

$\forall$ iterations $\vec{p} = (p_1, p_2)$, $\vec{p}' = (p'_1, p'_2)$ and $\vec{p}' > \vec{p}$, $\vec{p}'$ depends on $\vec{p}$ if $(:, p_1)$ == $(:, p'_1)$ or $(:, p_2)$ == $(:, p'_2)$, i.e., $\begin{cases} := \\ p_1 = p'_1 \end{cases}$ or $\begin{cases} := \\ p_2 = p'_2 \end{cases}$. The dependence vecotrs are thus $(0, \infty)$ and $(\infty, 0)$.

Partition & schedule the iteration space (Sec. 4.3)

Since $\forall$ dependence vector $\vec{d} = (d_1, d_2)$, $d_1 = 0$ or $d_2 = 0$, unordered 2D parallelization is applicable. Partition `ratings` by its 1st and 2nd dimension.

Partition accessed DistArrays (Sec. 4.4)

W and H are range partitioned by its 2nd dimension and allocated among executors. The smaller one of W and H is rotated among executors.

Compilation process:

Statically extract loop information

Compute dependence vectors

Determine the iteration space's partitioning scheme and computation schedule

Determine the accessed DistArrays's partitioning scheme

Code generation: DistArray partitioning, driver variable broadcast, loop body functions, parallel for-loop execution, etc.

Figure 6. Overview of Orion's static parallelization process using SGD MF as an example.

statement in a Julia program is just-in-time compiled and executed before the following global statements are compiled, thus the compilation of a statement may make use of previous statements' runtime execution results, such as DistArray sizes. Fig. 6 presents an overview of the JIT compilation process using SGD MF as an example.

4.2 Computing Dependence Vectors

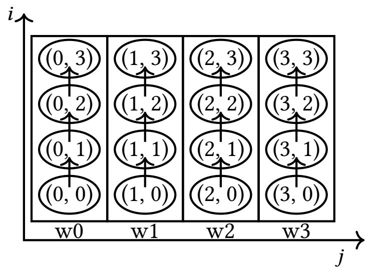
A lexicographically positive vector[5] $\vec{d}$ denotes a dependence vector of an n-loop nest if and only if there exist two dependent iterations $\vec{p_1}$ and $\vec{p_2}$ such that $\vec{p_1} = \vec{p_2} + \vec{d}$. Infinity ∞ (or positive/negative infinity, $+\infty/-\infty$) in dependence vectors 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 occuring 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 `(dim_idx, const , 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 subscripts 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.

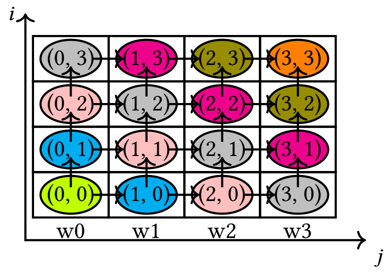
4.3 Parallelization and Scheduling

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.

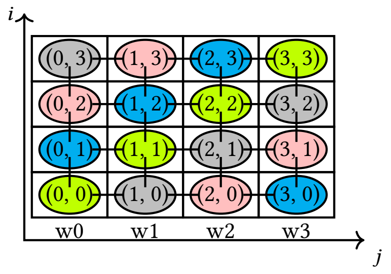
[5]A vector $\vec{d} = (d_1, d_2, \ldots, d_n)$ is lexicographically positive if $\exists i : d_i > 0$ and $\forall j < i : d_j \geq 0$



(a) 1D parallelization.



(b) 2D parallelization.



(c) Unordered 2D parallelization.

```
in_parallel for j = 0:(N-1)
  for iter in partition[j]
    execute_iteration(iter)
  synchronize()
```

(d) 1D computation schedule.

```
for time_step = 0:(M + N - 2)
  in_parallel for j = 0:(N-1)
    i = time_step - j
    if i >= 0 && i < N
      for iter in partition[j, i]
        execute_iteration(iter)
    synchronize()
```

(e) 2D computation schedule.

```
for time_step = 0:(M-1)
  in_parallel for j = 0:(N-1)
    i = (j + time_step) % N
    for iter in partition[j, i]
      execute_iteration(iter)
    synchronize()
```

(f) Unordered 2D computation schedule.

Figure 7. Parallelization of a 4×4 iteration space. Ellipses denote loop iterations and edges denote dependence between iterations. Note that representing the dependence in (a) requires only 1 depenence 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 $\mathcal{D}$, if there exists a dimension i such that $\forall \vec{d} = (d_1, d_2, ..., d_n) \in \mathcal{D}, d_i = 0$, 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 $\forall \vec{d} = (d_1, d_2, ..., d_n) \in \mathcal{D}, d_i = 0, d_j = 0$, 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 $(3, 0)$.

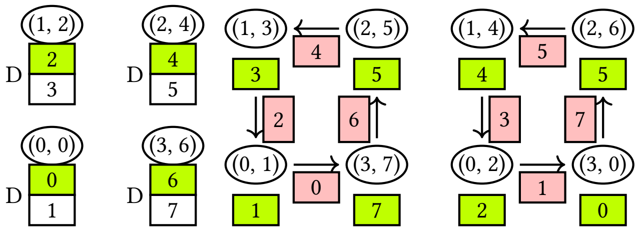
Many ML algorithms such as Gibbs sampling do not require a particular ordering in which data samples or mini-batches are processed. Other algorithms such as stochastic

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
    ▷ Skip checking dependence if both references are read or
    if the loop is unordered and both references are write.
    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 lexicogrpahical 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



(a) time step 0 **(b)** time step 1 **(c)** time step 2

Figure 8. Pipelined computation of a 2D parallelized unordered loop on 4 workers. An ellipse represents a worker executing a partition (`space_partition_id`, `time_partition_id`). The workers access different partitions of DistArray `D` at different time steps. Partitions of `D` that are being used by workers are lime-colored and the partitions that are being communicated are pink-colored. At the beginning of the loop execution, each worker is assigned with 2 time partition indices and thus 2 partitions of DistArray `D`. Upon finishing the first time step, a worker sends out the updated `D` partition and immediately begins the next time step using its locally available `D` partition.

set of dependence vectors after unimodular transformation denoted as $\mathcal{D}'$ satisfy that $\forall \vec{d} = (d_1, d_2, ..., d_n) \in \mathcal{D}' : d_1 > 0$ (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 balanaced 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, a TensorFlow while-loop enables additional parallelism for loops with a large and

Category	Examples	DSM	Programming Paradigm	App. Program. Lang.
Dataflow	Spark [52], DryadLINQ [51]	No	dataflow	Scala, Java, Python
Dataflow w/ mutable states	TensorFlow [5]	Yes	dataflow	Python, C, C++
Parameter Server	parameter server [29], Bösen [45]	Yes	imperative	C++
PS w/ scheduling	STRADS [26]	Yes	imperative	C++
Graph Processing	PowerGraph [20], PowerLyra [10]	Limited	vertex programming	C++
Orion		Yes	imperative	Julia

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

complex loop body (e.g., a multi-layer RNN), since the loop body can be distributed among multiple computing devices. Moreover, a TensorFlow while-loop dynamically computes a 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.[6] 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 E5-2698Bv3 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:

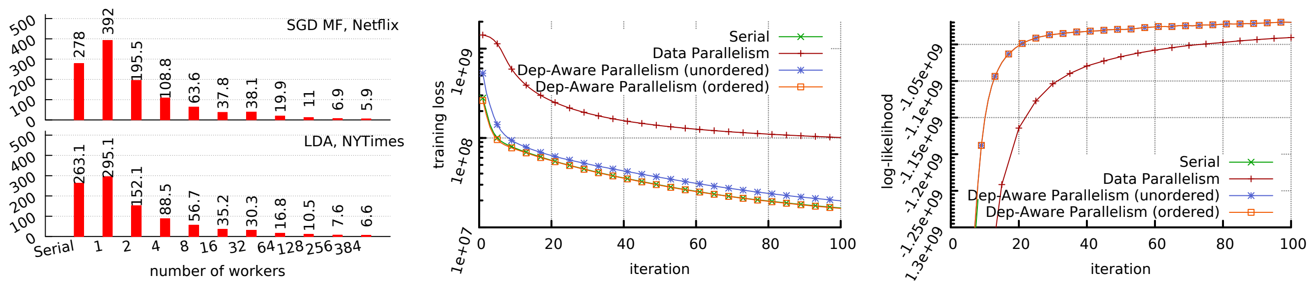
1. Is the algorithms' convergence rate sensitive to data dependence? Can dependence violation (such as data parallelism) significantly slow down algorithm convergence? Previous work (e.g., STRADS [26]) demonstrated that data dependence may have critical impact on algorithmic convergence and our results 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 competative 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 NYTimes 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

[6]URL: https://github.com/jinliangwei/orion

Acronym	Model	Learning Algorithm	LoC	Parallelizations
SGD MF	Matrix Factorization	SGD	87	2D Unordered
SGD MF AdaRev	Matrix Factorization	SGD w/ Adaptive Revision	108	2D Unordered
SLR	Sparse Logistic Regression	SGD	118	1D (data parallelism)
SLR AdaRev	Sparse Logistic Regression	SGD w/ Adaptive Revision	143	1D (data parallelism)
LDA	Latent Dirichlet Allocation	Collaposed Gibbs Sampling	398	2D Unordered, 1D
GBT	Gradient Boosted Tree	Gradient Boosting	695	1D

Table 2. ML applications parallelized by Orion.



(a) Time (seconds) per iteration **(b)** SGD MF, Netflix **(c)** LDA, NYTimes

Figure 9. Orion parallelization effectiveness. (a) compares the time per iteration (averaged over iteration 2 to 8) of serial Julia programs with Orion-parallelized programs. The Orion-parallelized programs are executed using different number of workers (virtual cores) on up to 12 machines, with up to 32 workers per machine. (b) and (c) compare the per-iteration convergence rate of different parallelization schemes and serial execution; the parallel programs are executed on 12 machines (384 workers).

computation throughput and overall convergence rate. The systems that we experimentally compare 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 an 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 $\sim 7 \times 10^{10}$ in ~600 seconds using 32 machines in its best case, while Orion SGD MF reaches $\sim 8.3 \times 10^7$ 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 model quality, while dependence-aware parallelization (even with proper semantic relaxations) retains a comparable per-iteratoin convergence rate to serial execution.
2. Orion-parallelized SGD MF (w/ and w/o AdaRev) and LDA converge significantly faster than manual data-parallel implementations on Bösen, in terms of both number of iterations and wall clock time.
3. Data-parallel SGD MF AdaRev and LDA on Bösen converges faster with more frequent communication of parameter values and updates, approaching Orion parallelization at the cost of higher network bandwith.
4. Orion-parallelized SGD MF AdaRev and LDA achieve a matching per-iteration convergence rate to manual model-parallel programs on STRADS, but may have a slower time per iteration mainly due to Julia's language overhead compared to C++.
5. Orion-parallelized SGD MF converges considerably faster than a data-parallel implementation on TensorFlow while achieving a 2.2× faster per-iteration time.

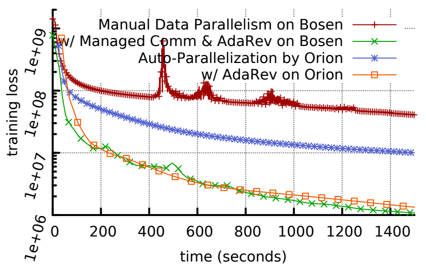
	Ordered	Unordered	Speedup
SGD MF (Netflix)	13.1	5.9	2.2×
SGD MF AdaRev (Netflix)	43.6	16.7	2.6×
LDA (NYTimes)	29.9	5.0	6.0×

Table 3. Time per iteration (seconds) with ordered and unordered 2D parallelization (12 machines), averaged over iteration 2 to 100.

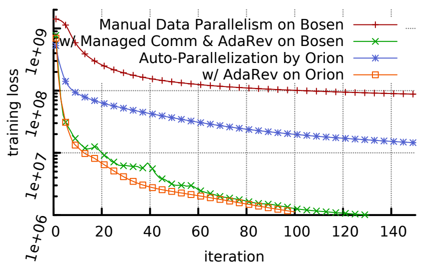
6.3 Parallelization Effectiveness

We compare Orion-parallelized Julia programs with serial Julia programs in terms of both computation throughput

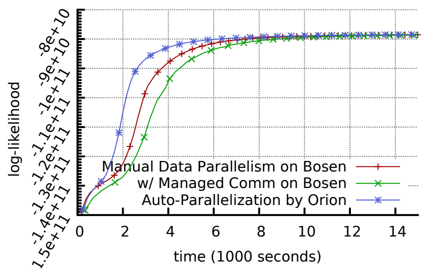
[7]TuX^2 is not open sourced



(a) Over time - SGD MF (AdaRev), Netflix

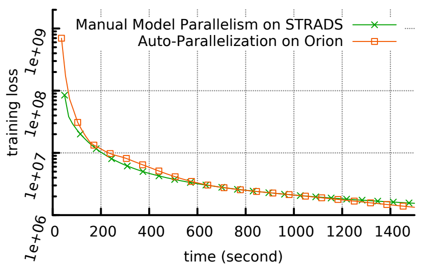


(b) Over iterations - SGD MF (AdaRev), Netflix

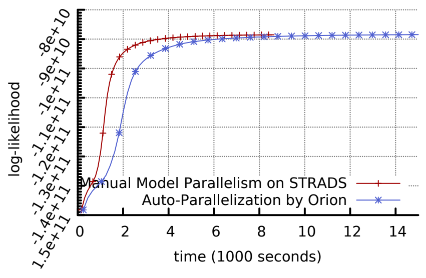


(c) Over time - LDA, ClueWeb

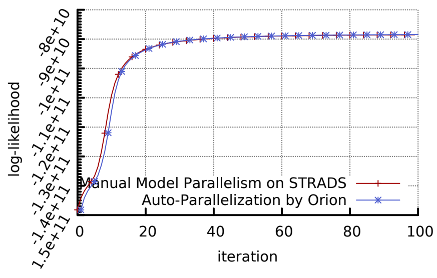
Figure 10. Orion vs. Bösen, convergence on 12 machines (384 workers)



(a) Over time - SGD MF AdaRev, Netflix



(b) Over time - LDA, ClueWeb



(c) Over iterations - LDA, ClueWeb

Figure 11. Orion vs. STRADS, convergene on 12 machiens (384 workers)

(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 constrants 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 interprocess communication, possibly over inter-machine networks. In a single-machine experiment using the KDD2010 (Algebra) [18] dataset, each data pass takes 7682 seconds,

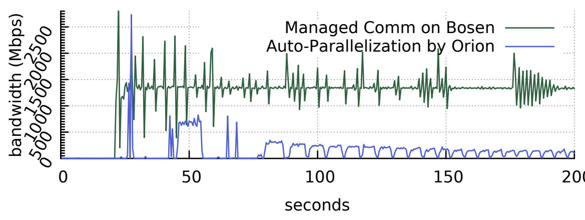


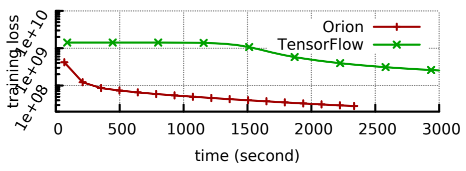
Figure 12. Bandwidth usage, LDA on NYTimes

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.

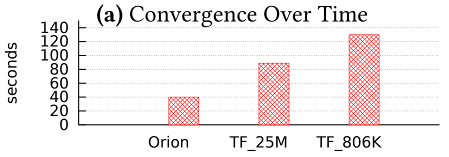
6.4 Comparison with Other Systems

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



(a) Convergence Over Time



(b) Time (seconds) per iteration; TF_x denotes a mini-batch size of x

Figure 13. Orion vs. TensorFlow, SGD MF on Netflix

2560Mbps respectively for SGD MF and LDA for maximal overall convergence rate. For SGD MF on Netflix and LDA on ClueWeb25M, CM achieves similar per-iteration convergence rate compared to dependence-aware parallelization by Orion but is still ~40% slower for LDA on NYTimes. For both SGD MF and LDA, CM uses substantially higher network bandwidth than Orion due to the aggressive communication (Fig. 12). Excessive communication incurs CPU overhead due to marshalling and lock contention, reducing Bösen's computation throughput and leading to a slower overall convergence rate than Orion when training LDA on ClueWeb25M.

Manual model parallelism. Compared to manually optimized model-parallel programs on STRADS, Orion-parallelized SGD MF AdaRev and LDA achieve a matching per-iteration convergence rate (Fig. 11). While achieving a similar computation throughput on SGD MF AdaRev, Orion takes ~1.8× (ClueWeb25M) and ~4.0× (NYTimes) longer than STRADS to execute an iteration for LDA. STRADS's better performance is largely due to a communication optimization: communicating data between workers on the same machine requires only pointer swapping. Since Julia (v0.6.2) doesn't yet support shared-memory multi-threading, inter-process communication in Orion incurs marshalling and memory copies. This overhead is negligible for SGD MF where the communication is mostly `float` arrays with trivial serialization.

TensorFlow. We compare Orion-parallelized SGD MF with an implementation on TensorFlow (v1.8), both executed on a single machine using only CPU (Fig. 13). Following TensorFlow (TF) common practices, our SGD MF program constructs a DAG which processes of a mini-batch of data matrix entries to exploit TF's highly parallelized operators. Since TF does not update model parameters until a full mini-batch is processed, TF SGD MF converges considerally slower than Orion's iteration-wise. With a mini-batch size of 25 million, TF is ~2.2× slower in terms of per-iteration time, partly due to redundant computation with respect to sparse data matrix (TF runs out of memory with larger mini-batch sizes). Each iteration takes longer with a smaller mini-batch size (Fig. 13b) because of not fully utilizing all CPU cores. Overall TF SGD MF converges much slower than Orion's, indicating that TF might not be the best option for sparse ML applications.

7 Related Work

Automatic parallelizing compilers. There have been decades of work on automatically parallelizing programs based on static data dependence analysis. This includes both vectorization [6, 37] and parallelization for multiple processors with a shared global memory, like Orion. Many loop transformation techniques have been developed for the latter, including loop interchange [47], loop skewing [48] and loop reversal. These transformations can be unified under unimodular transformations [46], which can only be applied to perfectly nested loops, e.g., traversing a multi-dimensional tensor. Affine scheduling [14, 16, 17] applies to arbitrary nestings of loops and unifies unimodular transformation with loop distribution, fusion, reindexing and scaling. Affine scheduling maps dynamic instances of instructions to a time space and instructions assgned the same time can be executed in parallel. Lim et al. [30] additionally partitions the instructions among processors to minimize synchronization.

Dynamic analysis. Pingali et al. [38] addresses parallelization by representing algorithms as operators and a topology, which describes the dependence between operators. The topology graph may be obtained from static analysis or dynamic tracing, or given as an input. Compared to static dependence analysis, this approach may be effective in parallelizing algorithms that deal with irregular data structures, e.g., graphs, but may suffer a larger overhead due to dynamic tracing and analyzing a large dependence graph.

Approximate computing. Previous work has proposed taking advantage of the approximate nature of application programs and introduced techniques, such as loop perforation [43] and task skipping [40] to reduce computation while sacrificing accuracy. Sampson et al. [41] rely on programmers to declare data that tolerates approximation so it can be mapped to lower-power hardware to save energy. HELIX-UP [9] also proposes to relax program semantics to increase parallelism and uses programmer-provided training inputs to tune the degree of approximation. Although auto-tuning could be incorporated in Orion, we believe that ML practitioners have domain-specific heuristics to make reasonable decisions while auto-tuning can be expensive.

8 Conclusion

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 substatially reducing programmer effort. We believe that Orion is an effective first step towards applying static dependence analysis to parallelize imperative ML programs for distributed training.

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

[1] 2009. Netflix Prize Data. https://www.kaggle.com/netflix-inc/netflix-prize-data/.

[2] 2013. ClueWeb. https://lemurproject.org/clueweb12/.

[3] Last visited Dec 2018. Julia Micro-Benchmark. https://julialang.org/benchmarks/.

[4] Last visited Dec 2018. MATLAB Parallel For Loop. https://www.mathworks.com/help/matlab/ref/parfor.html.

[5] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[6] Randy Allen and Ken Kennedy. 1987. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Transactions on Programming Languages and Systems* 9 (1987), 491–542.

[7] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[8] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3 (March 2003), 993–1022. http://dl.acm.org/citation.cfm?id=944919.944937

[9] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2015. HELIX-UP: Relaxing Program Semantics to Unleash Parallelization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 235–245. http://dl.acm.org/citation.cfm?id=2738600.2738630

[10] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*.

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

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

[13] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313

[14] Alain Darte and Yves Robert. 1995. Affine-by-Statement Scheduling of Uniform and Affine Loop Nests over Parametric Domains. *J. Parallel Distrib. Comput.* 29 (08 1995), 43–59. https://doi.org/10.1006/jpdc.1995.1105

[15] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.* 12 (July 2011), 2121–2159. http://dl.acm.org/citation.cfm?id=1953048.2021068

[16] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming* 21, 5 (01 Oct 1992), 313–347. https://doi.org/10.1007/BF01407835

[17] Paul Feautrier. 1992. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming* 21, 6 (01 Dec 1992), 389–420. https://doi.org/10.1007/BF01379404

[18] Hsiang fu Yu, Hung yi Lo, Hsun ping Hsieh, Jing kai Lou, Todd G. Mckenzie, Jung wei Chou, Po han Chung, Chia hua Ho, Chun fu Chang, Jui yu Weng, En syu Yan, Che wei Chang, Tsung ting Kuo, Po Tzu Chang, Chieh Po, Chien yuan Wang, Yi hung Huang, Yu xun Ruan, Yu shi Lin, Shou de Lin, Hsuan tien Lin, and Chih jen Lin. 2011. Feature engineering and classifier ensemble for KDD Cup 2010. In *In JMLR Workshop and Conference Proceedings*.

[19] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. 2011. Large-scale Matrix Factorization with Distributed Stochastic Gradient Descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '11)*. ACM, New York, NY, USA, 69–77. https://doi.org/10.1145/2020408.2020426

[20] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 17–30.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

[22] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B. Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In *Advances in Neural Information Processing Systems 26*, C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger (Eds.). Curran Associates, Inc., 1223–1231.

[23] Elad Hoffer, Itay Hubara, and Daniel Soudry. 2017. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. 1729–1739.

[24] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[25] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *CoRR* abs/1609.04836 (2016). arXiv:1609.04836 http://arxiv.org/abs/1609.04836

[26] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. 2016. STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. ACM, New York, NY, USA, Article 5, 16 pages. https://doi.org/10.1145/2901318.2901331

[27] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (Aug. 2009), 30–37.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., USA, 1097–1105. http://dl.acm.org/citation.cfm?id=2999134.2999257

[29] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 583–598.

[30] Amy W. Lim and Monica S. Lam. 1998. Maximizing Parallelism and Minimizing Synchronization with Affine Partitions. In *Parallel Computing*. ACM Press, 201–214.

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

[32] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Framework For Parallel Machine Learning. In *UAI*.

[33] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. 1991. Efficient and Exact Data Dependence Analysis. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI '91)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/113445.113447

[34] H. Brendan McMahan and Matthew Streeter. 2014. Delay-Tolerant Algorithms for Asynchronous Distributed Online Learning. *Advances in Neural Information Processing Systems (NIPS)* (2014).

[35] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I. Jordan. 2015. SparkNet: Training Deep Networks in Spark. *CoRR* abs/1511.06051 (2015). arXiv:1511.06051 http://arxiv.org/abs/1511.06051

[36] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: A Universal Execution Engine for Distributed Data-flow Computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 113–126. http://dl.acm.org/citation.cfm?id=1972457.1972470

[37] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop Vectorization: Revisited for Short SIMD Architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 2–11. https://doi.org/10.1145/1454115.1454119

[38] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 12–25. https://doi.org/10.1145/1993498.1993501

[39] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems 24*, J. Shawe-Taylor, R.S. Zemel, P.L. Bartlett, F. Pereira, and K.Q. Weinberger (Eds.). Curran Associates, Inc., 693–701.

[40] Martin Rinard. 2006. Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*. ACM, New York, NY, USA, 324–334. https://doi.org/10.1145/1183401.1183447

[41] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. *SIGPLAN Not.* 46, 6 (June 2011), 164–174. https://doi.org/10.1145/1993316.1993518

[42] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., 10435–10444.

[43] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing Performance vs. Accuracy Tradeoffs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 124–134. https://doi.org/10.1145/2025113.2025133

[44] Suvrit Sra, Adams Wei Yu, Mu Li, and Alexander J. Smola. 2016. AdaDelay: Delay Adaptive Distributed Stochastic Optimization. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*. 957–965. http://jmlr.org/proceedings/papers/v51/sra16.html

[45] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2015. Managed Communication and Consistency for Fast Data-parallel Iterative Analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 381–394. https://doi.org/10.1145/2806777.2806778

[46] Michael E. Wolf and Monica S. Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems* 2, 2 (Oct. 1991), 452–472.

[47] Michael Wolfe. 1986. Advanced Loop Interchanging. In *ICPP*.

[48] Michael Wolfe. 1986. Loops skewing: The wavefront method revisited. *International Journal of Parallel Programming* 15, 4 (01 Aug 1986), 279–293. https://doi.org/10.1007/BF01407876

[49] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. 2017. Tux^2: Distributed Graph Computation for Machine Learning. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 669–682. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/xiao

[50] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Gordon Murray, and Xiaoqiang Zheng. 2018. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. 18:1–18:15. https://doi.org/10.1145/3190508.3190551

[51] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 1–14. http://dl.acm.org/citation.cfm?id=1855741.1855742

[52] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of*

the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). USENIX, San Jose, CA, 15–28.

[53] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the Hidden Dimension in Graph Processing. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 285–300. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhang-mingxing

[54] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu