STRADS-AP: Simplifying Distributed Machine Learning Programming without Introducing a New Programming Model

Jin Kyu Kim and Abutalib Aghayev, ***Carnegie Mellon University;***
Garth A. Gibson, ***Carnegie Mellon University, Vector Institute, University of Toronto;***
Eric P. Xing, ***Petuum Inc, Carnegie Mellon University***

https://www.usenix.org/conference/atc19/presentation/kim-jin

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

STRADS-AP: Simplifying Distributed Machine Learning Programming without Introducing a New Programming Model

Jin Kyu Kim[1] Abutalib Aghayev[1] Garth A. Gibson[1,2,3] Eric P. Xing[1,4]
[1]*Carnegie Mellon University* [2]*Vector Institute* [3]*University of Toronto* [4]*Petuum, Inc.*

Abstract

It is a daunting task for a data scientist to convert sequential code for a Machine Learning (ML) model, published by an ML researcher, to a distributed framework that runs on a cluster and operates on massive datasets. The process of fitting the sequential code to an appropriate programming model and data abstractions determined by the framework of choice requires significant engineering and cognitive effort. Furthermore, inherent constraints of frameworks sometimes lead to inefficient implementations, delivering suboptimal performance.

We show that it is possible to achieve *automatic* and *efficient* distributed parallelization of *familiar* sequential ML code by making a few mechanical changes to it while hiding the details of concurrency control, data partitioning, task parallelization, and fault-tolerance. To this end, we design and implement a new distributed ML framework, STRADS-Automatic Parallelization (AP), and demonstrate that it simplifies distributed ML programming significantly, while outperforming a popular data-parallel framework with a *non-familiar* programming model, and achieving performance comparable to an ML-specialized framework.

1 Introduction

The systems community has made significant progress on simplifying distributed parallel programming, producing many high-level frameworks such as MapReduce [13], Spark [54], Pregel [34], PowerGraph [19], GraphX [20], PyTorch [40], and TensorFlow [2]. To automatically parallelize computation while achieving essential requirements such as fault tolerance and load balancing, these frameworks offer constrained programming models and limited data abstractions. For example, Spark offers Resilient Distributed Datasets (RDDs) without fine-grained write access; Spark and MapReduce ask programmers to specify a program using a handful of operators such as *map* and *reduce* while GraphLab requires adopting a rarely used *vertex-centric* programming model.

The programming models of these frameworks are different from a sequential programming model that is widely taught and easily understood [27]. Therefore, it is not surprising that rewriting sequential ML code using the data abstractions and programming models provided by the frameworks incurs significant effort. Furthermore, the simplicity of the mechanisms provided can often result in suboptimal use of cluster resources. These frameworks abstract away data placement, task mapping, and communication, which comes at the cost of limited access to hardware resources, and challenge in implementing ML algorithms efficiently. Studies show that a single threaded [35] or an MPI implementation [50] of popular ML algorithms is up to two orders of magnitude faster than the corresponding implementations on popular frameworks. In summary, a high-level framework often requires data scientists to switch to a **different mental programming model** with its own peculiarities, and it can end up delivering **suboptimal performance**. We believe that the complexity surrounding distributed ML programming as well as the inefficiency in execution are *incidental* and not *inherent*. That is, many sequential ML code can be automatically parallelized to make near optimal use of cluster resources. To prove our point, we present STRADS-AP, a novel distributed ML framework that provides an API requiring minimally-invasive, mechanical changes to sequential ML program code, and a runtime that automatically parallelizes the code on an arbitrary-sized cluster while delivering the performance of hand-tuned distributed ML programs.

STRADS-AP is an evolution of STRADS [26] that provides a framework for parallelizing the execution of ML programs according to user-specified scheduling plan. The plan usually avoids data conflicts, thereby improving statistical progress per iteration. The challenge with STRADS is that the user needs to understand the code and manually come up with a scheduling plan. STRADS-AP addresses this challenge by automatically generating data conflict-free scheduling plan.

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

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

To fill the gap of debugging tools for distributed ML pro-

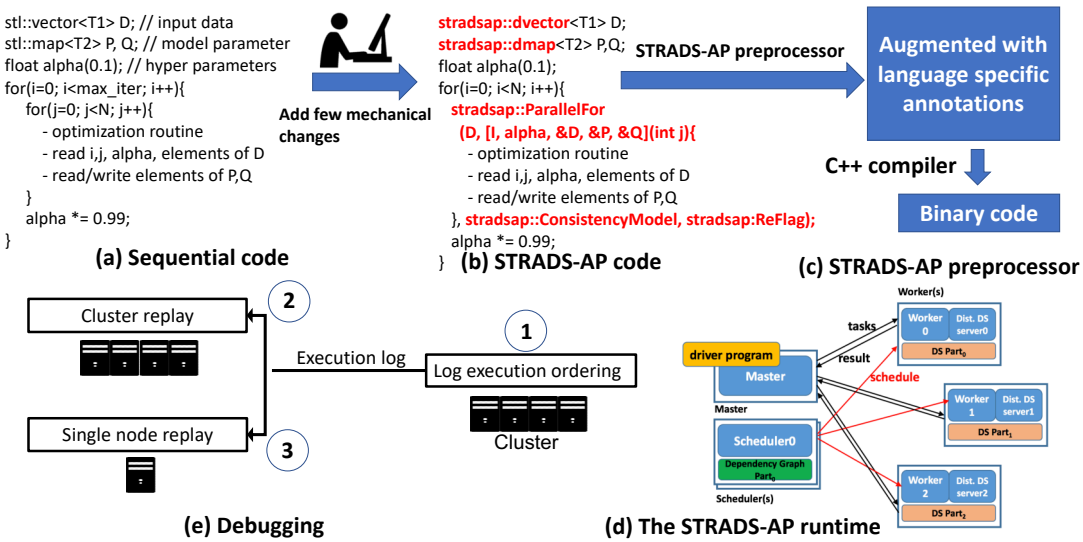


Figure 1: STRADS-AP workflow: (a) Data scientist implements an ML algorithm in sequential code; (b) Derives STRADS-AP parallel code with mechanical changes; (c) STRADS-AP preprocessor adds more annotation to address language-specific constraints, and the source code is compiled by a native compiler; (d) The STRADS-AP runtime runs the binary in parallel on a cluster; (e) Debugging features of STRADS-AP: Logging parallel execution order, and replaying it on a cluster ② for deterministic re-execution, or on a single node ③ for easy debugging.

grams, STRADS-AP offers two debugging modes—*cluster replay* and *single-node replay*—as shown in Figure 1(e). In *cluster replay* mode, the parallel execution log from the previous parallel run is replayed by obeying the same lock grant ordering and message ordering (② in Figure 1(e)), allowing deterministic re-execution. In *single-node replay* mode, the parallel execution log is replayed on a single node (③ in Figure 1(e)) allowing easier inspection of program state with a debugger.

TensorFlow and PyTorch simplify programming Deep Neural Networks (DNN) models, which is just one of the plethora of ML models. Implementing or researching non-DNN models and algorithms in these frameworks, however, often requires adding new kernel operators for parallelization, taking significant effort (Section 6.1). STRADS-AP, on the other hand, provides automatic parallelization of a wide range of non-DNN algorithms by requiring few mechanical changes to a serial implementation.

We implement STRADS-AP as a C++ library in about 16,000 lines of code.[1] STRADS-AP is largely rewritten from scratch, reusing some components of STRADS. We evaluate its performance on a moderate-sized cluster with four widely-used ML applications, using real data sets. To evaluate the increase in user productivity, we ask a group of students to convert a serial ML application to a distributed one using STRADS-AP, and we report our findings. The key contributions of our work are:

- The STRADS-AP API, a familiar C++ STL-like data structures and loop operators, requiring minimal changes when converting sequential ML code to STRADS-AP parallel code.
- The STRADS-AP runtime that achieves low latency DDS access, fault-tolerance, and concurrency control.
- Two debugging modes that simplify debugging and verification of distributed ML programs.
- Performance and productivity evaluation with four well-established ML applications implemented on STRADS-AP.

In the rest of this paper, we first make the case for STRADS-AP by presenting the complications imposed by high-level frameworks on users (Section 2), as well as the performance bottlenecks caused by their simple mechanisms, giving specific examples. We then present the STRADS-AP API (Section 3), and runtime implementation details (Section 4). Next, we give an overview of STRADS-AP's debugging features (Figure 5), followed by an extensive performance and productivity evaluation (Section 6). Finally, we cover the related work (Section 7) and conclude (Section 8).

2 The Cost of Using a Framework

In this section, we demonstrate that converting sequential ML code into high-level framework code requires substantial programming effort and leads to suboptimal performance

Algorithm 1 Pseudocode for SGDMF

```
1: A: a set of ratings. Each rating contains (i:user id, j:item id, r: rating)
2: W:M × K matrix; initialize W randomly
3: H:N × K matrix; initialize H randomly
4: for each rating r in A
5:     err = r.r - W[r.i]*H[r.j]
6:     ΔW = γ*(err*H[r.j] -λ*W[r.i] )
7:     ΔH = γ*(err*W[r.i] -λ*H[r.j] )
8:     W[r.i] += ΔW
9:     H[r.j] += ΔH
```

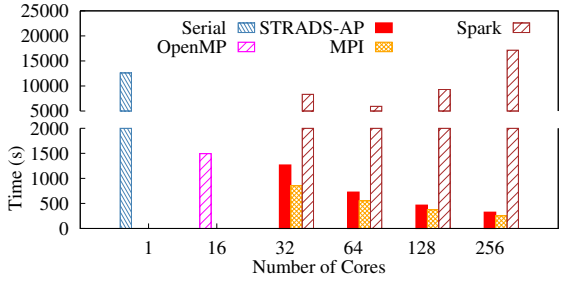
[1] Reported by CLOC [1] tool, skipping blanks and comments.

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

(a) Sequential SGDMF code

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

(b) Code in (a) parallelized with OpenMP



(e) Time for 60 iterations with Netflix dataset [24], rank = 1000.

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

(c) Code in (a) parallellized with STRADS-AP API

```
1  val P = K  // number of executors
2  val ratings = sc.textFile(rfile, P).map(parser)
3  val blks=sc.parallelize(0 until P, P).persist()
4  val W = blks.map(a->Create_WpSubmatrix(a))
5  var H = blks.map(a->Create_HpSubmatrix(a))
6  var AW = ratings.join(W,P)
7  var AWH = AW.join(H,P).mapPartitions(a->ComputeFunc(a,0))
8  float gamma(.01f), lambda(.1f);
9  for(auto i(0);i<maxiter;i++){
10   for(auto sub(0);sub<P;sub++){ // subiteration
11     val idx = i*P + sub;
12     if(idx > 0){
13       AWH = AW(idx).join(H,P).
14         mapPartitions(a->ComputeFunc(a,subepoch))
15     }
16     AW = AWH(idx).mapPartitions(x->separateAW_Func(x))
17     H = AWH.map(x->separateH_and_Shift_Func(x))
18   }
19 }
20 def ComputeFunc(it:Iterator to AWH){
21   val tmp = ArrayBuffer[type of AWH]
22   for(e <- it){
23     val Ap = e.Ap
24     var Wp = e.Wp
25     var Hp = e.Hp
26     for(auto r: Ap){
27       if(r.2 not belong to Hp)
28         continue //skip if not in Hp item indices
29       val err = r.3 - Wp[r.1]*Hp[r.2]
30       val Wd = gamma*(err*Wp[r.1]-lambda*Hp[r.2])
31       val Hd = gamma*(err*Hp[r.2]-lambda*Wp[r.1])
32       Wp[r.i] += Wd
33       Hp[r.j] += Hd
34     } // end of for(auto r ..
35     tmp += Tuple2(e.key, ((Ap, Wp), Hp));
36   } // end of for(e ..
37   val ret = tmp.toArray
38   ret.iterator
39 } // end of def update_Func
```

(d) Algorithm 1 reimplemented with Spark

Figure 2: SGDMF (Algorithm 1) implemented as sequential code (a), and parallelized using OpenMP (b), STRADS-AP (c) and Spark (d). The code snippets show only the core training routine, and do not include data loading and parsing code. STRADS-AP code requires fewer changes to the sequential code compared to OpenMP code, while achieving efficient distributed parallelism in addition to shared-memory parallelism. Spark, on the other hand, requires a complete reimplementation using its abstractions. STRADS-AP outperforms Spark by more than an order of magnitude (e) and continues to scale up to 256 cores, while Spark stops scaling at 64 cores. Hand-tuned MPI code is faster than STRADS-AP by 22% on 256 cores at the cost of a significantly longer programming and debugging effort.

that is orders of magnitude slower than a STRADS-AP implementation, or a hand-tuned implementation.

As a concrete example, we choose Spark as the framework, and Matrix Factorization (MF) as the algorithm—a popular recommender systems algorithm. First, we write sequential code that solves MF using Stochastic Gradient Descent (SGD), denoted as SGDMF in Algorithm 1. Then, we convert the sequential code into three different parallel implementations—shared-memory, STRADS-AP, and Spark—and compare their performance.

2.1 Programming Effort

MF learns user's preferences over all items from an incomplete rating dataset represented as a sparse matrix $A \in \mathbb{R}^{M \times N}$ where M and N are the number of users and items, respectively. MF factorizes the incomplete matrix A into two low-rank matrices, $W \in \mathbb{R}^{M \times K}$ and $H \in \mathbb{R}^{N \times K}$, such that $W \cdot H^T$ approximates A.

Algorithm 1 iterates through the ratings in the matrix A and for each rating $r_{i,j}$, it calculates gradients $\Delta W[i]$, $\Delta H[j]$ and adds the gradients to $W[i]$, $H[j]$, respectively. The computed parameter values for the rating $r_{i,j}$ are immediately visible when computing the next rating, which is an example of asynchronous computation.

2.1.1 Sequential SGDMF code

Sequential implementation of Algorithm 1 is a direct translation of the pseudocode as shown in Figure 2(a).

2.1.2 OpenMP Parallel SGDMF

We parallelize the sequential code in Figure 2(a) using OpenMP [12] to form a single-node baseline. We make two modifications to the sequential code as shown in Figure 2 (b): annotate the loop with parallel-for pragma to let the OpenMP runtime know what to parallelize, and add mutexes to avoid data race on shared matrices W and H. OpenMP parallelizes the inner loop over loop indices using fork-join model where threads run the loop body with different loop indices and join at the completion.

2.1.3 STRADS-AP Distributed SGDMF

Parallelizing the same sequential code using STRADS-AP is done almost mechanically by (1) replacing serial data structures with STRADS-AP's distributed data structures, and (2) replacing the inner loop with STRADS-AP's AsyncFor loop operator, as shown in Figure 2(c).

Unlike OpenMP code, STRADS-AP code has no explicit locking. The runtime is responsible for addressing data conflicts on matrices W and H while executing the loop body in a distributed setting, relieving users from writing error-prone locking code. With little effort, STRADS-AP achieves efficient distributed parallelism, in addition to shared-memory parallelism.

2.1.4 Spark Distributed SGDMF

Unlike with STRADS-AP, parallelizing the sequential code in Figure 2 (a) with Spark requires significant programming effort as detailed below.

Concurrency Control: Spark lacks concurrency control primitives. Since the inner loop of SGDMF leads to data dependencies when parallelized, we need to implement a scheduling plan for correct execution. Reasoning about concurrency control is application-specific and often requires a significant design effort. For SGDMF, we use the Strata scheduling algorithm by Gemulla [18]. The scheduling code shown in Figure 2(d) (lines 9-19) and the training code (lines 20-40) were abridged to fit the page.

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

First, Spark operators, such as *map*, operate on a single RDD object, while the inner loop body in Figure 2(a) accesses multiple objects: the input data A, and the parameter matrices W and H. To parallelize the inner loop with *map* we need to merge A, W, and H into a single RDD, requiring multiple *join* operations involving costly data shuffling.

Second, merging via *join* operator requires changes to data structures. Since *join* operator works only on RDD[Key,Value] type, we have to replace the vectors A, W, H, in Figure 2(a), with RDD[K,V] where V might be also key-value pair type.

Finally, data movement for concurrency control requires extra *join* and *map* operations. At the end of every subiteration, the Strata scheduling algorithm moves H partitions among nodes, which requires two extra operations for every subiteration: (1) a *map* operation that separates H from the merged RDD and modifies the key field of H, (2) a *join* operation that remerges H and AW into AWH for the next subiteration, as shown in Figure 2(d) (lines 9-19.)

In summary, engineering the Spark implementation of SGDMF algorithm involves a large amount of *incidental* complexity that stems from the limitations of Spark API and its data abstractions. As we show next, in addition to the loss in productivity, there is also a loss in efficiency.

2.2 Performance Cost

As a baseline for the distributed implementations, we implement SGDMF using MPI [16] and OpenMP, which are efficient at the cost of larger programming effort. MPI SGDMF uses the same Strata scheduling algorithm and point-to-point communication to circulate H partitions among nodes. All SGDMF implementations achieve proper concurrency control, making similar statistical progress per iteration. Therefore, our performance comparison focuses only on elapsed time for running 60 iterations, after which all implementations converge. We run experiments with Netflix dataset using up to 256 cores on 16 machines that are connected via 40Gbps Ethernet.

As Figure 2(e) shows, Spark is about 68× slower than MPI on 256 cores. In the same setting, STRADS-AP is slower than

	Requires Changing Programming Model	Programming Language	Application-Level Concurrency Control	Hides Details of Distributed Programming	Fault Tolerance	Debuggability for Distributed ML
STRADS-AP	No	C++	Yes	Yes	Yes (Fast re-execution)	Yes
STRADS [26]	Yes (model-schedule)	C++	Yes(user defined schedule)	Partly (communication)	Yes (Checkpoint)	No
Orion [49]	No	Julia	Yes	Yes	Yes (Checkpoint)	No
GraphLab [19]	Yes (vertex-centric)	C++	Yes	Yes	Yes (Checkpoint)	No
Spark [54]	Yes (map/reduce/...)	Multi	No	Yes	Yes (RDD)	Partly
TensorFlow [2]	Yes (data-flow)	Multi	No	Yes	Yes (Checkpoint)	Yes
Parameter Server [31]	Yes (key-value)	C++	No	Partly (parameter comm)	Yes (Replication)	No
MPI [16]	Yes (message-passing)	C	No	Partly (communication)	No	No
UPC [14]	Yes (PGAS)	C	Partly (lock APIs)	Yes	No	No

Table 1: Summary of features of frameworks used in distributed ML programming. For detailed comparison, refer to Section 2.3 and Section 7. For efficiency comparison, see Section 6

MPI by only 22%, whereas it is over 50× faster than Spark. The suboptimal performance of Spark implementation is due to the aforementioned factors (Section 2.1.4). STRADS-AP is 38.8 and 4.6 times faster than sequential and OpenMP, respectively.

2.3 Other High-Level Frameworks

Our findings of incidental complexity and suboptimal performance are not limited to the example of Spark and SGDMF. For example, PowerGraph provides concurrency control mechanisms, but its *vertex-centric* programming model requires users to redesign data structures to fit to a graph representation and express computations using GAS (Gather, Apply, Scatter) routines. TensorFlow provides a very high-level programming model taking a loss function and automates the gradient update process but does not support serializable asynchronous computation well. Parameter Servers (PS) [3,11,31,48] abstract away the details of parameter communication through the key-value store interface but many other details of distributed parallel programming, such as data partitioning, parallelization of tasks, and application-level concurrency control, are left to the user; that is, PS does not provide an illusion of sequential programming. UPC [14] extends the C programming language with Partitioned Global Address Space (PGAS) programming model that burdens the programmer with the job of doing careful performance tuning (i.e. affinity between threads and shared memory partitions, low-level data layout, use of collective functions such as gather, scatter, reduce.) As Table 1 shows, STRADS-AP and Orion [49] are the only frameworks that allow users to take their sequential code and automatically parallelize it to run on a cluster without sacrificing productivity or efficiency. STRADS-AP owes this flexibility to its familiar API and data structures that we describe next. The differences between STRADS-AP and Orion are described in detail in Section 7.

3 STRADS-AP Programming Interface

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

To implement a STRADS-AP application, a user writes a simple *driver program* that declares hyper-parameters, invokes

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

(a) Pretraining part

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

(b) Training part

Figure 3: ML applications targeted by STRADS-AP are divided into two parts: (a) pretraining part and (b) training part.

operators to create and transform DDSs (Figure 3 (a)), and then invokes STRADS-AP loop operators for optimization (Figure 3(b).) We describe each of these in the following subsections.

3.1 Distributed Data Structures (DDSs)

Table 2 shows a subset of STRADS-AP programming interface. DDS[T] is a mutable in-memory container that partitions a collection of elements of type T over a cluster. DDSs provide a global address space abstraction with fine-grained read/write access and uniform access model independent of whether the accessed element is stored in a local memory or in the memory of a remote node. STRADS-AP offers three types of containers: *dvector*, *dmap*, and *dmultimap*, with interfaces similar to their C++ STL counterparts. These DDSs allow all threads running on all nodes to read and write arbitrary elements while unaware of details such as data partitioning and placement.

Support for distributed and fine-grained read/write accesses gives STRADS-AP an important advantage over other frameworks. It allows reuse of data structures and routines from a sequential program by changing just the declaration of the data type. We describe the inner workings of DDSs in Section 4.3.

3.2 STRADS-AP Operators

The two parts of ML applications, pretraining and training (Figure 3), have different workload characteristics. Pretraining is data-intensive, non-iterative, and embarrassingly-parallel,

	Type	Description
Distributed Data Structures (DDSs)	dvector[T]	A distributed vector of type T elements with per-element read/write access
	dmap[K,V]	A distributed map of [K,V] element pairs of type K and V with per-element read/write access
	dmultimap[K,V]	A distributed multimap of [K,V] element pairs of type K and V with per-element read/write access
Loop Operators	AsyncFor(int64 **S**, int64 **E**, UDF **F**)	Parallelizes closure **F** over indices [**S**, **E**] in isolated manner—avoid ing data conflicts
	SyncFor(DDS[T] **&D**, int **M**, UDF **F**, SyncOpt **S**, bool **RE**)	Parallelizes closure **F** over minibatches of **D** each of size **M** using synchronization option **S** in data-parallel manner. **RE** indicates whether to perform Reconnaissance Execution(§ 4.2)

Table 2: A subset of STRADS-AP API—DDSs, and loop operators for ML training.

whereas training is compute-intensive and iterative, and the inner loop(s) may have data dependencies. STRADS-AP provides two sets of operators that allow natural expression of both types of computation.

3.2.1 Pretraining Operators

STRADS-AP provides *Map*, *Reduce*, *Join*, *Load*, and *Create* operators for loading, storing, and creating DDSs during pretraining. However, STRADS-AP puts no constraints on their usage for expressing training computations.

3.2.2 Loop Operators

STRADS-AP provides loop operators shown in Table 2 to replace the inner loop(s) in the training part of ML programs (Figure 3 (b)). The loop operators take a user-defined closure as the loop body. The closure is a C++ lambda expression that captures the specified DDSs and variables in the scope, and implements the loop body by reading from and writing to arbitrary elements of the captured DDSs. This allows users to mechanically change the loop body of a sequential ML program to STRADS-AP code that is automatically parallelized.

STRADS-AP supports four models of parallelizing ML computations: (1) serializable asynchronous [33], (2) synchronous (BSP [46]), (3) stale-synchronous (SSP [23]), and (4) lock-free asynchronous (Hogwild! [38]) within a node and synchronous across nodes, which we call Hybrid.

STRADS-AP offers two loop operators to support these models. A user can choose AsyncFor loop operator for serializable asynchronous model. For the remaining models a user can choose SyncFor operator and specify the desired model as an argument to the loop operator, as shown in Table 2. Other than choosing the appropriate loop operator, a user does not have to write any code for concurrency-control—the STRADS-AP runtime will enforce the chosen model as described next.

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

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

If the lambda expression modifies a DDS, then data conflicts will happen. Although ML algorithms are error-tolerant [23], some algorithms, like Coordinate Descent Lasso [29, 45], LDA [5, 53], and SGDMF [18, 28] converge slowly in the presence of numerical errors due to data conflicts. Following previous work [26], STRADS-AP runtime improves statistical progress by avoiding data conflicts using data conflict-free scheduling for lambda executions. Figure 2(c) shows an example use of AsyncFor implementing SGDMF.

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

SyncFor takes five arguments: the input data D of type DDS[T], the size of a mini-batch M, a C++ lambda expression F, a synchronization option (BSP, SSP, or Hybrid), and a flag indicating whether it should perform Reconnaissance Execution (Section 4.2). The runtime partitions the input data chunk of a node into L mini-batches of size M (typically L is much larger than the number of threads per node), and then schedules multiple threads to process mini-batches concurrently. A thread executes the lambda expression with a local copy of captured variables, and allows reads and writes only to the local copy while running F. At the end of processing a mini-batch, a separate per-node thread synchronizes the local copy of only those DDSs captured by reference across the nodes, and synchronizes local threads according to the sync option. Figure 4 shows an example use of SyncFor that reimplements Google's Word2vec model [22].

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

4 Implementation

This section covers important details of STRADS-AP implementation: the driver program execution (Section 4.1), Reconnaissance Execution (Section 4.2), DDSs (Section 4.3), Concurrency Control (Section 4.4), and STRADS-AP preprocessor (Section 4.5).

4.1 Execution of Driver Program

In the STRADS-AP driver program, the statements are classified into three categories: sequential statements, STRADS-AP

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

Figure 4: Reimplementing Google's Word2vec model using STRADS-AP API.

data processing statements, and STRADS-AP loop statements. The runtime maintains a state machine with one state per category to keep track of the type of code to execute. A driver node starts the driver program in sequential state, and performs sequential execution locally until the first invocation of a STRADS-AP operator. On a STRADS-AP operator invocation, the state machine switches to the corresponding state and the runtime parallelizes the operator over multiple nodes. At the completion of the STRADS-AP operator, the runtime switches back to sequential state and continues running the driver program locally.

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

4.2 Reconnaissance Execution

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

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

For the SyncFor operator, when the parameter access is sparse (that is, a small portion of parameters are accessed when processing a mini-batch), the runtime reduces the amount of data transferred by referring to *access records* of RE. However, in applications with dense parameter access, (that is, most parameters are accessed when processing a mini-batch), RE does not help to improve the performance. Therefore, the SyncFor operator's boolean RE parameter (Table 2) allows users to skip RE and prefetch/cache all elements of DDSs captured by the corresponding lambda expression.

To reduce RE overhead, STRADS-AP runs it once per parallel loop operator in the driver program, and reuses read/write set for subsequent iterations. This optimization is based on two assumptions about ML workloads: (1) *iterativeness*—a loop operator is repeated many times until convergence, and (2) *static control flow*—read/write sets of loop bodies do not change over different iterations. That is, the control flow of the inner loop does not depend on model parameter values. Both assumptions are routinely accepted in ML algorithms [3–5, 8, 17, 24, 28, 30, 41, 45, 51–53, 55].

4.3 Distributed Data Structures

On the surface, a DDS is a C++ class template that provides index- or key-based uniform access operator. Under the hood, the elements of a DDS are stored in a distributed in-memory key-value store as key-value pairs. The key is uniquely composed of the table id plus the element index for *dvector*, and the table id plus the element key for *dmap/dmultimap*. Each node in a cluster runs a server of the distributed key-value store containing the elements of a DDS partitioned by the key hash. The implementation of DDS class template reduces the element access latency by prefetching and caching remote elements based on the access records generated by RE (Section 4.2).

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

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

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

4.4 Concurrency Control

STRADS-AP implements two concurrency control engines: (1) *serializable engine* for the AsyncFor operator, and (2) *data-parallel engine* for the SyncFor operator. Both engines use the read/write set from Reconnaissance Execution (Section 4.2) for prefetching remote DDS elements, while the serializable engine also uses it for making data conflict-free execution plans.

4.4.1 Serializable Engine for AsyncFor

In the serializable engine, a task is defined as the loop body with a unique loop index value i, which ranges from S to E, where S and E are AsyncFor arguments (Table 2). The serializable engine implements a scheduler module that takes the read/write set from RE, analyzes data dependencies, generates a dependency graph, and generates a parallel execution plan that avoids data conflicts. To increase parallelism, the serializable engine may change the execution order of tasks assuming that any serial reordering of the loop body executions is acceptable. This assumption is also routinely accepted in ML computations [26, 33, 39].

The scheduler divides the loop bodies into N *task groups*, where N is much larger than the number of nodes in a cluster, using an algorithm that combines the ideas of static scheduling from STRADS [26] and connected component-based scheduling from Cyclades [39]. The algorithm allows dependencies within a task group but ensures no dependency across task groups. At runtime, the scheduler places task groups on nodes, where each node keeps a pool of task groups.

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

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

To reduce scheduling overhead, the serializable engine caches the scheduling plan and reuses it over multiple iterations based on the aforementioned assumptions (Section 4.2). Hence, the overhead of RE and computing a scheduling plan is amortized over multiple iterations.

4.4.2 Data-Parallel Engine for SyncFor

In the data-parallel engine, a task is defined as the loop body with a mini-batch of D with size M, where D and M are SyncFor arguments (Table 2). Hence, a single SyncFor call generates multiple tasks with different mini-batches. The engine places the tasks on nodes that hold the associated mini-batches, where nodes form a pool of assigned tasks.

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

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

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

The SyncFor operator allows users to choose among BSP, SSP, and Hybrid (Section 3.2.2) models of parallelizing ML computations. The BSP [46] and SSP [23] models are well-known, and our implementation follows previous work. Hybrid, on the other hand, is a lesser-known model [25]. It allows lock-free asynchronous update of parameters among threads within a node (Hogwild! [38]), but synchronizes across machines at fixed intervals. In the Hybrid model, a node creates a single DDS cache that is accessed by all threads without taking locks. When all of the threads complete a single task, which denotes a subiteration, the node synchronizes the DDS cache with the key-value store.

4.5 STRADS-AP Preprocessor

While there exist mature serialization libraries for C++, none of them support serializing lambda function objects. The lack of reflection capability in C++, and the fact that lambda functions are implemented as anonymous function objects [36], make serializing lambda challenging. We overcome this challenge by implementing a preprocessor that analyzes the source code using Clang AST Matcher library [10], identifies the types of STRADS-AP operator arguments, and generates RPC stub code and a uniquely-named function object for each lambda expression that is passed to STRADS-AP operators.

The preprocessor also analyzes the source code to see if it declares DDSs of user-defined types. While DDSs of built-in types are automatically serialized using Boost Serialization library [6], for user-defined types the library requires adding boilerplate code, which is automatically added by the preprocessor.

5 Debugging STRADS-AP Applications

STRADS-AP supports two debugging modes: (1) *cluster replay* mode, and (2) *single-node replay* mode. Currently, STRADS-AP debugging modes support only serializable parallel execution generated by AsyncFor operator whose

Dataset	Workload	Feature	Size	Application	Purpose
Netflix [24]	100M ratings	489K users, 17K movies, rank=1000	2.2 GB	SGDMF	Recommendations
1Billion [9]	1 billion words	Vocabulary size 308K, vector size=100	4.5 GB	Word2Vec	Word Embeddings
ImageNet [43]	285K images	1K classes, 21K features, preprocessed by LCC feature extraction [47]	21 GB	MLR	Multi-Class Classification
FreeBase-15K [7]	483K facts	14,951 entities, 1,345 relations, vector size=100	36 MB	TransE	Graph Embeddings

Table 3: Datasets used in benchmarks.

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

Table 4: ML programs used for benchmarking. Serial and OpenMP are single core and multi-core applications on a shared-memory machine, respectively, while the rest are distributed parallel applications. MPI applications use OpenMP for shared-memory parallelism within the nodes.

execution can be non-deterministic (Section 4.4.1). The support for SyncFor operator is in progress.

Cluster Replay: The AsyncFor operator allows non-deterministic execution for achieving high performance. Instead of predefined deterministic execution [32], STRADS-AP logs the execution order including lock grant ordering and message ordering, and allows users to replay the log. For this purpose, STRADS-AP implements record/replay modules. The record module records the ordering of lock grantings in every node, and the ordering of message arrivals in the DDS key-value store into persistent storage. To avoid coordination overhead and bottlenecking a single node, each node records partial ordering locally, without making a total ordering. When replaying the log, each node enforces the same partial order.

Single-node Replay: Debugging an ML program can be classified into two categories: (1) search for a traditional software bug, and (2) the inspection of the optimization path. Unfortunately, these debugging tasks are not easy to do in a distributed environment since step-by-step tracing of a distributed application is hard. To address this problem, STRADS-AP offers *single-node replay* mode for parallel ML applications. The *single-node replay* mode takes a parallel execution log, and replays the ordering in a single node setting, where users can trace the program execution using a debugger like GDB.

6 Evaluation

We evaluate STRADS-AP on (1) application performance, and (2) programmer productivity, using the following real world ML applications: SGDMF, Multinomial Logistic Regression (MLR), Word2vec, and TransE [7], summarized in Table 4.

For performance evaluation, as a baseline we implement these applications as (1) a sequential C++ application, (2) a single-node shared-memory parallel C++ application using OpenMP, and (3) a distributed- and shared-memory parallel C++ application using MPI and OpenMP. We then compare the iteration throughput (time per epoch) and the statistical accuracy of Spark, TensorFlow (TF), and STRADS-AP implementations of these applications to those of the baselines, while running them on real datasets shown in Table 3. For brevity, in the rest of the paper, when we mention that an application is implemented using MPI, we mean that it uses OpenMP on a single node and MPI among the nodes.

For productivity evaluation, we conduct two user studies on a group of students with Word2vec and TransE applications. As a measure of productivity, we count the lines of code produced, and measure the time it took students to convert a serial implementation of the algorithm into a STRADS-AP implementation.

All experiments were run on a cluster with 16 machines each with 64 GB of memory and 16-core Intel Xeon E5520 CPUs, running Ubuntu 16.04 Linux distribution, connected with 40 Gbps Ethernet network. The reported numbers are the averages of at least three runs. Error bars are not included due to low variance among the runs.

6.1 Word2Vec

Word2vec is a Natural Language Processing (NLP) model developed by Google that computes vector representations of words, called "word embeddings". These representations can be used for various NLP tasks, such as discovering semantic similarity. Word2vec can be implemented using two architectures: continuous bag-of-words (CBOW) or continuous skip-gram, the latter of which produces more accurate results for large datasets [22].

We implement the skip-gram architecture in STRADS-AP based on Google's open source multithreaded implementation [22] written in C. We make two changes to Google's implementation: (1) modify it to keep all the input data in memory to avoid I/O during training, and (2) replace POSIX threads with OpenMP. After our changes, we observe 6% increase in performance on 16 cores. We then run our improved implementation using a single thread for the serial baseline, and using 16 threads on 16 cores for the shared-memory parallel baseline.

Google recently released a highly-optimized multithreaded Word2vec [21] implementation on TensorFlow with two custom kernel operators written in C++. As of now, Google has not yet released a distributed version of Word2vec on TensorFlow. Therefore, we extend Google's implementation to run in a data-parallel distributed setting. To this end, we modify the kernel operators to work on partitions of input data, and synchronize parameters among nodes using MPI.

Performance Evaluation: Figure 5 shows the execution time

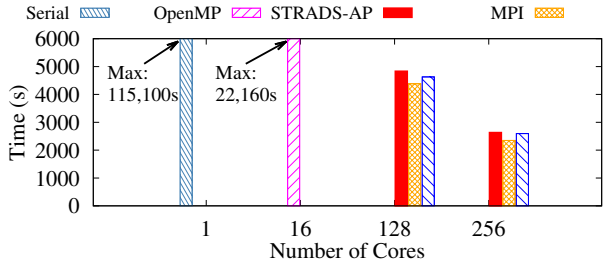


Figure 5: Time for 10 iterations of Word2Vec on 1 Billion word dataset [9] with vector size = 100, window = 5, negative sample count = 10.

Cores	Similarity			Analogy		
	STRADS-AP	MPI	TF	STRADS-AP	MPI	TF
128	0.601	0.601	0.602	0.566	0.564	0.568
256	0.603	0.597	0.601	0.562	0.557	0.561

Serial	0.610	0.570
OpenMP	0.608	0.571

Table 5: The top table reports similarity test accuracy [15], and analogy test accuracy [37] for distributed Word2Vec implementations on 1 Billion word dataset, after 10 iterations. The bottom table shows respective values for the serial and OpenMP implementations.

of Word2vec for 10 iterations with a 1 billion word data set. On 256 cores (16 machines), MPI performs better than TensorFlow and STRADS-AP by 9.4% and 10.1%, respectively; that is, a STRADS-AP program obtained by mechanical changes to serial code matches the performance of a highly optimized TensorFlow program. The higher performance of an MPI program stems from the serialization overhead in TensorFlow and STRADS-AP. The MPI implementation stores parameters in arrays of built-in types, and uses in-place MPI_Allreduce call to operate on values directly. STRADS-AP outperforms serial and OpenMP implementations by 45× and 8.7×, respectively.

Table 5 shows the similarity test accuracy [15] and the analogy test [37] accuracy, after running 10 iterations. Using the accuracy of the serial algorithm as the baseline, we see that parallel implementations report slightly lower accuracy (within 1.1%) than the baseline due to the use of stale parameter values.

Productivity Evaluation: Table 6 shows the line counts of Word2vec implementations in the first column. The STRADS-AP implementation has 15% fewer lines than the serial implementation, which stems mainly from the coding style and the use of STRADS-AP's built-in text-parsing library. If we focus the comparison on the core of the program—the training routine—both implementations have around 100 lines, since STRADS-AP implementation takes the serial code and makes a few simple changes to it.

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

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

Table 6: Line counts of model implementations using different frameworks. Unless specified next to the lines counts, the language is C++. *TensorFlow implementation of Word2vec has 282 lines in Python and 364 lines in C++.

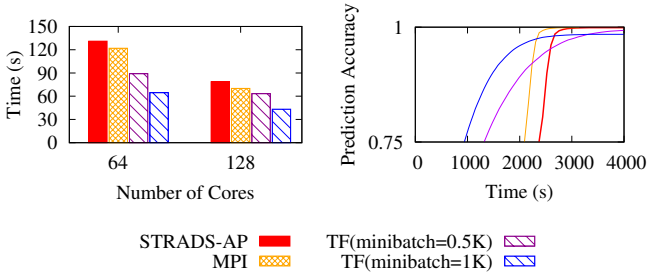


Figure 6: The left figure shows time for a single iteration. We run the TensorFlow implementation with a minibatch sizes of 500 and 1,000. STRADS-AP and MPI implementations do not use vector instructions, therefore, we run them with a minibatch size of 1. Serial and OpenMP implementations (omitted from the graph) also run with a minibatch size of 1, and take 3,380 and 467 seconds to complete, respectively. The right figure shows prediction accuracy as the training progresses. While each implementation runs for 60 iterations, the graph shows only the time until all of them converge to a stable value.

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

6.2 Multinomial Logistic Regression

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

Performance Evaluation: Figure 6 shows single iteration time on 25% of the ImageNet dataset [43] on the left, and accuracy after 60 iterations on the right. TensorFlow makes heavy use of vector instructions, which explains the 30% decrease in runtime when increasing the minibatch size from 500 to 1,000, and almost two times shorter runtime than MPI and STRADS-

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

Table 7: The breakdown of times (in hours) of five students that converted the serial implementation of the TransE [7] graph embedding algorithm to a distributed STRADS-AP implementation. We split the conversion task into five subtasks: [T1] understand the algorithm, [T2] understand the reference serial code, [T3] review STRADS-AP API guide, [T4] review the provided serial MLR code and the corresponding STRADS-AP code, [T5] convert the serial implementation to STRADS-AP implementation.

AP implementations, which do not use vector instructions. On the other hand, as the right graph in Figure 6 shows, TensorFlow sacrifices accuracy for higher throughput. Unlike the MPI and STRADS-AP implementations that achieve 99.5% accuracy after about 2,800 seconds, the accuracy of TensorFlow remains under 98.4 even after 4,000 seconds. The difference in accuracy is due to STRADS-AP and MPI implementations running with a minibatch size of 1, given that they do not use vector instructions. A single iteration of TensorFlow with a minibatch size of 1 (for which it was not optimized) took about 6 hours, which we omitted from the graph.

Productivity Evaluation: Table 6 shows the line counts of MLR implementations in the second column. The TensorFlow implementation has 38% and 50% fewer lines than the STRADS-AP and MPI implementations, respectively, because while both of the latter implement large chunks of code to compute gradients and apply them to parameters, TensorFlow hides all of these under library function calls. On the other hand, most of the TensorFlow implementation consists of code for partitioning data and setting up the cluster and parameter server variables. This is counter-productive for users who do not want to deal with cluster setup and data partitioning, but want to change the algorithms.

6.3 Matrix Factorization

We already covered (Section 2.1) the implementation details and performance evaluation of solving Matrix Factorization algorithm using SGD optimization (SGDMF). Therefore, we continue with the productivity evaluation.

Productivity evaluation: Table 6 shows line counts of SGDMF implementations in the third column. SGDMF implementation in Scala, even after including the line count for Gemulla's Strata scheduling algorithm (Section 2.1.4), has about 15% fewer lines than STRADS-AP implementation. This is not surprising, given that functional languages like Scala tend to have more expressive power than imperative languages like C++. However, the difficulty of implementing the Strata scheduling algorithm is not captured well in the line count. Figuring out how to implement this algorithm using the limited Spark primitives, and tuning the performance so that the lineage graph would not consume all the memory on the cluster took us about a week, whereas deriving STRADS-AP implementation from the pseudocode took us about an hour.

The line count of MPI is higher than serial code due to Strata scheduling implementation and manual data partitioning.

6.4 User Study

To further evaluate the productivity gains of using STRADS-AP, we conducted two more user studies. In the first study, as a capstone project we assigned a graduate student to implement a distributed version of Word2vec using STRADS-AP and MPI, after studying Google's C implementation [22]. The student had C and C++ programming experience, and had just finished an introductory ML course. After studying the reference source code, the student spent about an hour studying the STRADS-AP API and experimenting with it. It then took him about two hours to deliver a working distributed Word2vec implemented with STRADS-AP API. On the other hand, it took the student two days to deliver a distributed Word2vec implemented with MPI. The MPI implementation was not able to match the STRADS-AP implementation in terms of accuracy and performance until the student had invested two weeks of performance optimizations.

In the second study, we conducted an experiment similar to a programming exam, with five graduate students. We provided the students with a two-page STRADS-AP API documentation, example serial MLR code, and the corresponding STRADS-AP code. We then gave the students a serial C++ program written by an external NLP research group that implemented the TransE [7] graph embedding algorithm, and asked them to produce a distributed version of the same program using STRADS-AP.

Table 7 shows the breakdown of times each student spent at different phases of the experiment, including the students' backgrounds, and the primary challenges they faced. While most students lacked proficiency in C++, they still managed to complete the conversion in a reasonable amount of time. Student 5, who was the most proficient in C++, finished the experiment in 1.5 hours, while Student 1 took 2.4 hours, most of which he spent in the last subtask debugging syntax errors, after breezing through the previous subtasks. Feedback from the participants indicated that (1) converting serial code into STRADS-AP code was straightforward because data structures, the control flow, and optimization functions in the serial program were reusable with a few changes, and (2) the lack of C++ familiarity was the main challenge. The list of reported

mistakes included C++ syntax errors, forgetting to resize local C++ STL vectors before populating them, and an attempt to create a nested DDS, which STRADS-AP does not currently support. We evaluated the students' implementations by running them on FreeBase-15K [7] dataset for 1000 iterations with vector size of 50. The students' implementations were about 22× faster than the serial implementation on 128 cores, averaging at 45.3% accuracy, compared to 46.1% accuracy achieved by the serial implementation.

6.5 Scope and Limitations of STRADS-AP

STRADS-AP facilitates converting serial ML programs into distributed ML programs with minimal changes. Our evaluation shows that the converted ML programs achieve performance comparable to hand-tuned distributed implementations, and to implementations written using ML-specialized frameworks. To achieve higher performance, STRADS-AP relies on two optimizations: reordering of loop indices to find more opportunities for parallelism, and reuse of RE output to amortize the overhead of running RE and making scheduling decision over multiple iterations. These optimizations require ML programs to meet three assumptions: serializability (4.4.1), iterativeness (4.2), and static control flow (4.2). Fortunately, these three assumptions are commonly found in a broad range of ML applications. However, STRADS-AP has some limitations on its expressiveness. For example, it does not support nested parallel loops and user defined data structures having nested DDSs.

7 Related Work

STRADS-AP's design elements rely on a body of previous work. The virtual iteration of IterStore parameter server [11] inspired Reconnaissance Execution (RE). IterStore uses the read/write set only for prefetching parameters into nodes from the parameter server. STRADS-AP, however, uses the read/write set for generating a data conflict-free execution schedule as well as prefetching.

Calvin [44] introduces reconnaissance queries for efficient distributed transactions with low locking overhead. However, Calvin cannot reuse the results of the query because DBMS workloads do not generally have the *iterativeness* and *static control flow* properties of ML workloads. STRADS-AP runtime runs RE on just the first invocation, and the output of RE is reused for every iteration until convergence, amortizing the cost of RE.

OpenMP [12] and Distributed R [42] are popular among ML programmers and provide parallel loop operators. However they lack the support of high-level abstractions to parallelize ML programs in which the ML training routine has data dependencies. For example, when loop bodies of a parallel loop have data dependencies on shared ML model parameters, OpenMP and Distributed R delegate concurrency control to the ML programmer. This requires ML programmers to write routines for aggregating shared parameters in the case of synchronous parallel execution, and handling data dependencies in the case of asynchronous execution. On the contrary, the STRADS-AP runtime hides parameter aggregation and concurrency control from ML programmers through Sync/Async loop operators.

GraphLab [19, 33], Cyclades [39], and STRADS [26, 29] present ML scheduling ideas that avoid executing updates with data conflicts to improve statistical progress per iteration. However, GraphLab expects users to express a serial ML algorithm using GAS (Gather, Apply, Scatter) primitives, Cyclades targets a single shared-memory machine, limiting scalability, and STRADS requires users to design and implement data conflict-free scheduling strategy. STRADS-AP addresses all of these limitations by (1) allowing users to convert a serial program into parallel program through mechanical changes, (2) scaling out to an arbitrary-sized cluster, and (3) automatically generating data conflict-free execution schedules.

More recently, Orion [49] proposed automatic parallelization using static analysis of matrix index access patterns. STRADS-AP and Orion [49] share the same goal of automating scheduling decision. However, while Orion targets ML programs written in Julia scripting language, STRADS-AP targets C++ ML programs because there are a large number of serial ML programs in written in C++. This difference in the choice of programming language leads us to explore substantially different design options, such as STL-like DDSs and dynamic analysis, instead of distributed matrix and static analysis. Specifically, Orion's static analysis requires that data dependencies are determined statically in the form of a linear combination of loop variables. This assumption does not hold frequently in real-world ML applications. On the contrary, STRADS-AP performs dynamic analysis that captures data accesses and dependencies at runtime without relying on such assumptions.

8 Conclusion

Despite the availability of a plethora of frameworks for distributed Machine Learning (ML) programming, we believe distributed ML programming is still unnecessarily complicated. Each framework comes with its own restricted programming model and abstractions, its inefficiencies, and peculiarities that add to the growing list of things that data scientists should master. We take a step back and ask: how can we take a serial imperative implementation of an ML model, and parallelize it over a cluster with minimal effort from the user.

Our answer is STRADS-AP—a distributed ML framework, which is a combination of a runtime and an API comprised of Distributed Data Structures (DDSs) and parallel loop operators. Using four real-world applications, we show that STRADS-AP allows data scientists to easily convert a serial implementation of an ML model to a distributed implementation that achieves performance comparable to hand-tuned MPI and TensorFlow implementations, while outperforming a Spark implementation by more than an order of magnitude.

Acknowledgements

We thank our shepherd Steven Hand and the anonymous reviewers. This research is supported in part by National Science Foundation under awards CCF-1629559, IIS-1563887, and IIS-1617583. We 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.

References

[1] CLOC: Count Lines of Code. http://cloc.sourceforge.net/.

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

[3] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. Scalable inference in latent variable models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 123–132, New York, NY, USA, 2012. ACM.

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

[5] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, March 2003.

[6] Boost. Boost C++ Library - Serialization. http://www.boost.org/doc/libs/1_66_0/libs/serialization/doc/index.html.

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

[8] Joseph K. Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 321–328, USA, 2011. Omnipress.

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

[10] Clang. AST Matcher Reference. http://clang.llvm.org/docs/LibASTMatchersReference.html.

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

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

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[14] Tarek El-Ghazawi and Lauren Smith. Upc: Unified parallel c. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

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

[16] Message P Forum. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.

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

[18] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. 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, pages 69–77, New York, NY, USA, 2011. ACM.

[19] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[20] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

[21] Google. TensorFLow Optimized Word2vec. https://github.com/tensorflow/models/blob/master/tutorials/embedding/word2vec_optimized.py.

[22] Google. word2vec. https://code.google.com/archive/p/word2vec/.

[23] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, pages 1223–1231, USA, 2013. Curran Associates Inc.

[24] James Bennett and Stan Lanning and Netflix Netflix. The Netflix Prize. In *In KDD Cup and Workshop in conjunction with KDD*, 2007.

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

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

[27] Keith Kirkpatrick. Parallel Computational Thinking. *Commun. ACM*, 60(12):17–19, November 2017.

[28] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009.

[29] Seunghak Lee, Jin Kyu Kim, Xun Zheng, Qirong Ho, Garth A. Gibson, and Eric P. Xing. On model parallelization and scheduling strategies for distributed machine learning. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2834–2842, Cambridge, MA, USA, 2014. MIT Press.

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

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

[32] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

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

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

[35] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[36] Microsoft Developer Network. Lambda Expressions in C++. https://msdn.microsoft.com/en-us/library/dd293608.aspx, 2015.

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

[38] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *Proceedings*

of the 24th International Conference on Neural Information Processing Systems, NIPS'11, pages 693–701, USA, 2011. Curran Associates Inc.

[39] Xinghao Pan, Maximilian Lam, Stephen Tu, Dimitris Papailiopoulos, Ce Zhang, Michael I. Jordan, Kannan Ramchandran, Chris Re, and Benjamin Recht. Cyclades: Conflict-free asynchronous machine learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 2576–2584, USA, 2016. Curran Associates Inc.

[40] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS 2017 Autodiff Workshopt: The Future of Gradient-based Machine Learning Software and Techniques*, Long Beach, CA, US, December 9, 2017.

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

[42] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[43] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *Int. J. Comput. Vision*, 115(3):211–252, December 2015.

[44] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

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

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

[47] Jinjun Wang, Jianchao Yang, Kai Yu, Fengjun Lv, Thomas S. Huang, and Yihong Gong. Locality-constrained linear coding for image classification. In *CVPR*, pages 3360–3367. IEEE Computer Society, 2010.

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

[49] Jinliang Wei, Garth A. Gibson, Phillip B. Gibbons, and Eric P. Xing. Automating dependence-aware parallelization of machine learning training on distributed shared memory. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 42:1–42:17, New York, NY, USA, 2019. ACM.

[50] Jinliang Wei, Jin Kyu Kim, and Garth A. Gibson. Benchmarking Apache Spark with Machine Learning Applications. Technical report, Carnegie Mellon University, 2016.

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

[52] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit Dhillon. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *Proceedings of the 2012 IEEE 12th International Conference on Data Mining*, ICDM '12, pages 765–774, Washington, DC, USA, 2012. IEEE Computer Society.

[53] Jinhui Yuan, Fei Gao, Qirong Ho, Wei Dai, Jinliang Wei, Xun Zheng, Eric Po Xing, Tie-Yan Liu, and Wei-Ying Ma. Lightlda: Big topic models on modest computer clusters. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1351–1361, Republic and Canton of Geneva, Switzerland, 2015. International World Wide Web Conferences Steering Committee.

[54] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

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