Efficient and Programmable Distributed Shared Memory Systems for Machine Learning Training

Jinliang Wei

August 16, 2018

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Garth A. Gibson, Co-chair
Eric P. Xing, Co-chair
Phillip B. Gibbons
Vijay Vasudevan, Google Brain

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Copyright © 2018 Jinliang Wei
Keywords: machine learning, distributed computing, distributed shared memory, static analysis, parallelization, scheduling, deep learning
Abstract

Machine learning training involves frequent and often sparse updates to a large number of numerical values called model parameters. Many distributed training systems have resorted to using distributed shared memory (DSM) (e.g. Parameter Server) for efficient sparse access and in-place updates. Compared to traditional programs, machine learning applications tolerate bounded error, which presents opportunities for trading off learning progress for higher computation throughput. In this thesis, I develop efficient and programmable distributed learning systems, by exploiting this trade-off in the design of distributed shared memory systems, along with parallelization and static and dynamic scheduling.

Thanks to this tolerance to bounded error, a machine learning program can often be parallelized without strictly preserving data dependence. Parallel workers may thus observe inconsistent model parameter values compared to a serial execution. More frequent communication to propagate updates and fresher parameter values may reduce such inconsistency, while incurring higher communication overhead. I present a communication management mechanism that automates communication using spare network bandwidth and prioritizes messages according to their importance in order to reduce error due to inconsistency while retaining high computation throughput.

When each model update reads and writes to only a subset of model parameters, it is possible to achieve an efficient parallelization while preserving critical data dependence, exploiting sparse parameter access. Existing systems require substantial programmer effort to take advantage of this opportunity. In order to achieve dependence-preserving parallelization without imposing a huge burden on application programmers, I present a system Orion that provides parallel for-loops on distributed shared memory and parallelizes loops with loop-carried dependence.

At last, I propose to explore dynamic scheduling for dynamic control flow in dataflow systems such as TensorFlow. In TensorFlow, the computation graph is statically partitioned and assigned with computation devices. Static device placement is suboptimal as the operators’ load can no longer be determined statically due to dynamic control flow. A suboptimal static device placement may result in imbalanced load and extra communication. It is promising to address the deficiency of static device placement by dynamically scheduling operations based on their load at runtime.
# Contents

## 1 Introduction

1.1 Performance Metrics for Distributed Training Systems ........................................... 2
1.2 Frequent Sparse Updates in Machine Learning Training ........................................... 2
1.3 Distributed Shared Memory for Efficient Updates .................................................... 3
1.4 Scheduling for Preserving Data Dependence ......................................................... 3
1.5 Programming Model Expressiveness Power ............................................................ 4
1.6 Thesis Statement and Contributions ........................................................................ 4
1.7 Systems for Distributed Offline Learning ............................................................... 5
   1.7.1 Dataflow Systems ............................................................................................... 5
   1.7.2 Distributed Shared Memory Systems ................................................................. 6
   1.7.3 Graph Processing Systems .................................................................................. 7

## 2 Managed Communication for Data-Parallel Training [76]

2.1 Data-Parallel Distributed Training and Synchronization ........................................... 8
   2.1.1 Data-Parallel Machine Learning Training ......................................................... 8
   2.1.2 Parameter Servers Synchronization Schemes .................................................... 9
2.2 Bösen Architecture and API .................................................................................... 10
   2.2.1 Bösen Client API ............................................................................................... 11
   2.2.2 Bounded Staleness Consistency ....................................................................... 11
   2.2.3 Client Library And Server Partitions ................................................................. 12
2.3 Managed Communication in Bösen ........................................................................ 13
   2.3.1 Bandwidth-Driven Communication .................................................................... 13
   2.3.2 Update Prioritization ......................................................................................... 14
2.4 Evaluation ............................................................................................................... 14
   2.4.1 Evaluation Setup ............................................................................................... 14
   2.4.2 Evaluating Managed Communication .................................................................. 16
   2.4.3 Comparing with Manual Mini-batch Size Tuning .............................................. 18
2.5 Conclusion ............................................................................................................. 19

## 3 Dependence-Aware Parallelization

3.1 Introduction ............................................................................................................ 20
   3.1.1 Machine Learning Training .............................................................................. 20
   3.1.2 Data Parallelism ............................................................................................... 21
   3.1.3 Dependence-aware Parallelization .................................................................... 21
3.2 Motivation ............................................................................................................. 23
   3.2.1 Batch Dataflow Systems ................................................................................. 23
   3.2.2 TensorFlow ..................................................................................................... 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3 Graph Processing Systems</td>
<td>24</td>
</tr>
<tr>
<td>3.2.4 Parameter Server Systems</td>
<td>25</td>
</tr>
<tr>
<td>3.2.5 Orion Design Summary</td>
<td>25</td>
</tr>
<tr>
<td>3.3 Orion’s Programming Model</td>
<td>26</td>
</tr>
<tr>
<td>3.3.1 Scripting for Productivity With Small Performance Loss</td>
<td>26</td>
</tr>
<tr>
<td>3.3.2 Distributed Arrays</td>
<td>28</td>
</tr>
<tr>
<td>3.3.3 Parallel For-Loop</td>
<td>28</td>
</tr>
<tr>
<td>3.3.4 Distributed Array Buffers</td>
<td>29</td>
</tr>
<tr>
<td>3.3.5 Program Annotations</td>
<td>30</td>
</tr>
<tr>
<td>3.3.6 Additional Features</td>
<td>30</td>
</tr>
<tr>
<td>3.3.7 Putting Everything Together</td>
<td>30</td>
</tr>
<tr>
<td>3.4 Static Parallelization And Code Generation</td>
<td>30</td>
</tr>
<tr>
<td>3.4.1 Parallelization</td>
<td>32</td>
</tr>
<tr>
<td>3.4.2 Static Scheduling</td>
<td>33</td>
</tr>
<tr>
<td>3.4.3 Partitioning DistArrays for Random Access</td>
<td>34</td>
</tr>
<tr>
<td>3.4.4 Dealing with Skewed Data Distribution</td>
<td>35</td>
</tr>
<tr>
<td>3.5 Preliminary Evaluation</td>
<td>37</td>
</tr>
<tr>
<td>3.5.1 Ease of Use</td>
<td>37</td>
</tr>
<tr>
<td>3.5.2 Orion Abstraction Overhead</td>
<td>39</td>
</tr>
<tr>
<td>3.5.3 Orion vs. Manual Data Parallelism</td>
<td>39</td>
</tr>
<tr>
<td>3.5.4 Orion vs. Manual Data Parallelism w/ Managed Communication</td>
<td>41</td>
</tr>
<tr>
<td>3.5.5 Orion vs. Manual Model Parallelism</td>
<td>43</td>
</tr>
<tr>
<td>3.5.6 Orion vs. Dataflow Systems</td>
<td>44</td>
</tr>
<tr>
<td>3.6 Summary and Planned Case Studies</td>
<td>44</td>
</tr>
<tr>
<td>4 Dynamic Scheduling for Dynamic Control Flow</td>
<td>46</td>
</tr>
<tr>
<td>4.1 Distributed Machine Learning w/ Dataflow Graph</td>
<td>46</td>
</tr>
<tr>
<td>4.2 Conditional Computation for Outrageously Large Neural Networks</td>
<td>46</td>
</tr>
<tr>
<td>4.3 Preliminary Experimental Investigation</td>
<td>48</td>
</tr>
<tr>
<td>4.3.1 Limited Scalability of MoE Networks</td>
<td>49</td>
</tr>
<tr>
<td>4.3.2 Load Distributation among Expert Networks</td>
<td>51</td>
</tr>
<tr>
<td>4.3.3 Throughput Drop</td>
<td>51</td>
</tr>
<tr>
<td>4.4 Dynamic Scheduling for Dynamic Control Flow in Dataflow Graphs</td>
<td>53</td>
</tr>
<tr>
<td>4.4.1 Incorporating Static and Dynamic Device Placement</td>
<td>53</td>
</tr>
<tr>
<td>4.4.2 Operator Scheduling Strategies</td>
<td>53</td>
</tr>
<tr>
<td>4.4.3 Operator Profiling</td>
<td>54</td>
</tr>
<tr>
<td>4.4.4 Memory Swapping and Dynamic Batching</td>
<td>54</td>
</tr>
<tr>
<td>Appendices</td>
<td>55</td>
</tr>
<tr>
<td>A Orion Application Program Examples</td>
<td>56</td>
</tr>
<tr>
<td>A.1 Stochastic Gradient Descent Matrix Factorization</td>
<td>56</td>
</tr>
<tr>
<td>A.2 Sparse Logistic Regression</td>
<td>58</td>
</tr>
<tr>
<td>Bibliography</td>
<td>61</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The range of Machine Learning (ML) applications is fast growing. Examples include personalized recommendations, visual and language understanding, game playing, and autonomous driving, just to name a few. At the core of ML applications is an expert-suggested model, whose parameters are determined by training the model to fit a set of observed data samples. Typically training starts with a random guess of the model parameters and refines them to optimize an **objective function** by iteratively processing a training dataset. This iterative process stops when the quality of the model as measured by the objective function reaches certain criteria or stops improving, which is referred to as **convergence**. Fig 1.1 is a cartoon that demonstrates the learning progress. Depending on the application, the objective function can be maximized (e.g. maximizing log-likelihood in topic modeling) or minimized (e.g. minimizing loss in regression). In both cases, the objective function value approaches an asymptote.

Due to the high complexity of training computation and iterative refinement nature and the large size of training datasets, training a model typically requires a large amount of computing power. The demand for computing power increases as more complex models are invented to serve more challenging real-world tasks. It is thus increasingly important to parallelize and distribute the training computation to reduce training time. In distributed machine learning training, the learning algorithm is parallelized across a set of worker machines which all contribute to refining a model’s parameters. The model parameters can be shared among workers using a distributed shared memory architecture (e.g. parameter servers). The iterative and convergent nature of the learning algorithms allow errors made in earlier iterations to be corrected in fol-

![Figure 1.1: A cartoon depicting the learning progress](image)
lowing iterations when the error is properly bounded. This tolerance to bounded error renders strongly consistent parameter storage an overkill at the expense of computation throughput. In order to achieve high computation throughput, distributed workers read parameter values from local cache and buffer updates until enough computation has been done to compensate for the synchronization overhead, which makes Bulk Synchronous Parallel [73] a commonly used parallel execution model. Even more relaxed synchronization schemes such as Stale Synchronous Parallel [43] or even total asynchrony may find acceptable model solutions while further improving computation throughput. Under the above synchronization models, workers typically observe an inconsistent view of the model parameters compared to a serial execution. This inconsistency can be viewed as error in a serial algorithm. Even though the algorithm tolerates bounded error, as error grows, the model quality improvements from each pass of the data is typically reduced, impeding learning progress. We refer to this tension between throughput and consistency as the consistency-throughput trade-off. In this thesis, we develop system techniques that reduces such inconsistency when exploiting this consistency-throughput trade-off and we build systems with expressive programming models to take advantage those techniques without requiring heavy programmer effort.

1.1 Performance Metrics for Distributed Training Systems

Traditional distributed batch processing systems perform deterministic computation and those systems’ performance are usually measured by its computation throughput, which measures the amount of data processed per second. However, in a distributed machine learning training system, the computation is usually nondeterministic. The number of data samples or the number of data passes needed to achieve certain model quality may differ depending on how the system manages consistency. Thus throughput alone can not measure distributed training systems’ performance. Instead we measure system performance by time to convergence, which is the time needed for the learning system to achieve certain quality for the given model as measured by its objective function. Time to convergence depends on both the system’s computation throughput as well as its convergence rate, i.e. the number of data items or data passes needed to be processed to reach certain model quality.

1.2 Frequent Sparse Updates in Machine Learning Training

In iterative machine learning training, the training dataset is usually divided into mini-batches and the model parameters are updated after each mini-batch. It is often desirable to use small mini-batches to update the model frequently as it speeds up the learning progress (less data samples need to be processed to achieve certain model quality). In the case of distributed training, applying parameter updates while maintaining a consistent view among workers incurs expensive coordination among workers and inter-machine network communication. The desired small mini-batch sizes would incur too frequent updates such that the cost of coordination and communication outweighs the benefit of parallelization if sequential consistency were retained.

In many applications, the updates are sparse where each update operation only reads and updates a small subset of the parameters. Thus updates that access different parameters may be executed in parallel without conflicts. Efficiently exploring such sparsity is key to achieve high throughput with low inconsistency.

Throughout this thesis, we refer to data pass as iteration or epoch interchangably.
1.3 Distributed Shared Memory for Efficient Updates

As opposed to message passing, Distributed Shared Memory (DSM) provides a global address space for variables distributed among a cluster of compute nodes, where network communication is abstracted away from application programmers, and used by the system to implement the memory coherence protocol. Classic DSMs provide data access via primitives such as memory allocation and pointer dereferencing [47, 52, 68], while modern examples [67] provide a key-value interface.

As Resilient Distributed Datasets (RDD) [82] are immutable once created, they can be shared among subsequent operations without conflicting access, which simplifies parallelization. However, updating a single record of an RDD would require creating a new RDD with the rest of the original RDD copied. In contrast, DSM efficiently supports frequent changes that update a small fraction of the stored values. This is well suited for storing shared model parameters in distributed training without too much inconsistency as workers often perform irregular, sparse access to large collections of model parameters.

Parameter Server has become a pivotal component in distributed machine learning training. A typical use case of a parameter server is data-parallel training, where the training dataset is randomly partitioned among the worker machines and the workers process their local data partitions to update the shared model parameters. Under data parallelism, conflicting access arises from different workers reading and updating the same parameters. In order to provide high-throughput, low-latency access to parameter values on top of the low-bandwidth, high-latency data center network, parameter servers actively exploit the error-tolerance property to buffer updates and serve potentially stale parameter values from workers’ local cache, in spite of conflicting accesses. Due to the consistency-throughput trade-off, computing with stale values slows the learning progress while improving process throughput. In Chapter 2 we present a communication management mechanism to manage this trade-off to improve learning progress without losing computation throughput.

1.4 Scheduling for Preserving Data Dependence

Existing approaches that exploit such sparsity for parallelization randomly assign data partitions to workers which works when the probability of conflict among data partitions is low [36, 69]. But random data partitioning doesn’t guarantee conflict-free parallelization. Although more frequent updates may reduce the probability of conflicts, it requires high network bandwidth and low network latency. Conflict-free parallelization would be ideal but existing systems do not achieve it because it requires accurately capturing the fine-grained data dependence among updates and scheduling computation accordingly.

As shown by Kim et. al. [51], properly scheduling the computation to minimize conflicting accesses can significantly improve the overall speed of learning. Existing systems offer little support for application programmers to easily exploit these opportunities. STRADS [51] provides an API for dispatching computation to specific workers, while it is left to the application programmers to design the parallelization and scheduling strategy. Some graph processing systems [57] schedule computation on vertices according to a user-provided graph, which would work if there happen to be convertible existing data graphs. However, when such graphs do not already exist, users are required to manufacture an appropriate dependence graph for the model and data in hand.
1.5 Programming Model Expressiveness Power

For efficient execution, existing distributed computing frameworks usually adopt a specialized programming model that restricts its expressiveness power, compared to programming on general-purpose shared memory. For example, batch dataflow systems require its input data and intermediate results to be immutable once created; graph processing systems restrict data accesses to neighboring vertices; and parameter server systems do not provide dynamic control flow. Also, dataflow systems adopt a declarative programming model [64, 82], while imperative programming is much more popular for shared memory programming. Circumventing such restrictions where possible at all typically requires substantial programmer effort when converting a shared memory program to a distributed one. Moreover, it may suffer a great performance penalty as efficient operations on shared memory such as random memory access might not be efficient supported by the distributed system. For example, machine learning applications on Spark typically represent their model parameters as local variables in its driver program and it is expensive to broadcast the parameters and collect updates from distributed workers [63].

1.6 Thesis Statement and Contributions

This thesis studies mechanisms to minimize inconsistency in distributed machine learning without losing computation throughput advantage of inconsistency under an expressive programming model. Based on my previous research, I propose to support the following thesis:

Efficient distributed machine learning, even with an expressive programming model that substantially reduces application programmer effort, can be realized by enhancing distributed shared memory with parallelization and scheduling.

In support of the above statement, I have made the following research contributions:

• I develop a mechanism for distributed shared memory that improves inter-machine network communication efficiency to reduce inconsistency due to conflicting accesses. This mechanism is generally applicable to data-parallel training. (Chapter 2)

• I develop a Bösen Parameter Server and implement the communication management mechanism in Bösen. I evaluate communication management’s effectiveness on a number of real-world machine learning applications such a collaborative filtering, topic modeling and multinomial logistic regression. (Chapter 2)

• I design a programming model and parallelization mechanism that extends parallel for-loop to imperative programming on distributed shared memory. Parallelization of the loop preserves loop-carried dependence and is designed to expose and take natural and powerful hints that improve computation throughput. (Chapter 3)

• I develop a distributed system Orion that implements the imperative programming model above. Orion is evaluated on a number of machine learning applications and compared with both Parameter Server and dataflow systems.

In order to complete the thesis, I propose to complete the following contributions:

• Perform case studies on using Orion to parallelize a number of popular ML applications that have traditionally required specialized implementations. (Chapter 3)

• Realize dynamic scheduling for dynamic control flow in dataflow graph (TensorFlow) to reduce memory footprint and improve computation efficiency. (Chapter 4)

Wei Dai contributed to the development of the Bösen parameter server and some of its machine learning applications.
The thesis organization is depicted in Fig. 1.2.

Figure 1.2: Distributed machine learning system and thesis organization

1.7 Systems for Distributed Offline Learning

To perform distributed machine learning training, people have used general-purpose batch processing systems [6, 82], developed machine learning oriented systems that exploit the consistency-throughput trade-off [28, 43, 53], developed more specialized frameworks to take advantage of application-specific features of certain subclass of machine learning [39, 57] and even engineered point solutions to specific learning problems [15, 24, 48].

Table 1.1 summarizes the representative and influential distributed batch processing or learning systems that were proposed in recent years. We focus on systems that proposed new programming models or revised existing ones for performance improvements. Such improvements typically come from exploiting application- or data-specific properties for balancing computation and communication, minimizing disk I/O or network communication, mitigating stragglers and employing more efficient in-memory data structures.

1.7.1 Dataflow Systems

Closely related to functional programming, dataflow systems [13, 35, 46, 64, 65, 80, 82] represent computation as a directed graph where vertices are side-effect-free operations. Thus in most dataflow systems, the input data and intermediate results are represented as immutable objects flowing along the directed edges. TensorFlow introduces operations that have mutable states, i.e. variables and queues. These operations produce a reference handle to its mutable state, which can be passed as input to other operations that modify the state. Some systems [82] implicitly partitions each data object among distributed nodes to distribute computation and storage, while others systems [13, 64] requires the application to partition a data object in order for it to be distributed among machines.

These frameworks typically exploit two forms of parallelism:

1. Each vertex operation can be parallelized across multiple cores or machines;
2. Independent vertices can be executed in parallel.

Exploiting parallelism within a vertex requires a large input to the vertex and a worker’s results or updates are not visible to other workers until the entire vertex finishes computing, which slows the learning progress. Exploiting parallelism among vertices requires application
programmers to decompose the computation to many small vertices, which is a substantial burden and may result in a graph whose size is proportional to data size.

TensorFlow additionally supports concurrent execution of overlapping subgraphs. Different subgraphs may update the shared states concurrently and thus an execution may read stale values compared to a serial execution.

### 1.7.2 Distributed Shared Memory Systems

In order to facilitate applications that perform frequent updates, each operating on a small number of the stored values, such as ML training, distributed shared memory (e.g. Parameter Server) is widely used both as an integral component of other systems [13] or as standalone systems [28, 29, 43, 53, 67]. Such systems typically expose a key-value interface for shared states which are hosted by a number of servers. In contrast to traditional databases, they usually offer a specialized and relaxed consistency model that takes advantage of application’s ability to tolerate inconsistency for high throughput. In a distributed shared memory system, applications are implemented as a worker program that executes on each worker machine or core. Although some systems [51, 53, 67] support a driver process to dispatch tasks to workers, it’s still left to the application programmers to design and implement a proper scheduling strategy.

<table>
<thead>
<tr>
<th>System Abstraction</th>
<th>System</th>
<th>Programming Model Features</th>
<th>Applications Benchmarked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dist. Shared Memory</td>
<td>Piccolo [67]</td>
<td>control functions table and key value abstraction table iterator, global barrier</td>
<td>PageRank, K-means, N-body, Matrix Multiply</td>
</tr>
<tr>
<td></td>
<td>IterStore [29]</td>
<td>virtual iteration</td>
<td>PageRank, LDA SGD MF</td>
</tr>
<tr>
<td></td>
<td>Parameter Server [53]</td>
<td>push-pull, driver-program, user-defined filters, server-side UDF</td>
<td>sparse LR, LDA, count-min sketch</td>
</tr>
<tr>
<td></td>
<td>Lazy Table [28,43], Bösen [76]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dataflow</td>
<td>MapReduce [35]</td>
<td>map-reduce, shuffle</td>
<td>grep, sort</td>
</tr>
<tr>
<td></td>
<td>Dryad [46]</td>
<td>arbitrary DAG</td>
<td>select-join-join query histogram</td>
</tr>
<tr>
<td></td>
<td>DryadLINQ [80]</td>
<td>LINQ expression, static compilation</td>
<td>TeraSort, PageRank SQL query, EM</td>
</tr>
<tr>
<td></td>
<td>CIEL [64]</td>
<td>dynamic control flow dynamic task dependency, lazy evaluation</td>
<td>grep, k-means, Smith-Waterman, binomial option pricing</td>
</tr>
<tr>
<td></td>
<td>Spark [82]</td>
<td>RDD, static dependency</td>
<td>logistic regression, k-means, page rank, analytical query, traffic modeling, spam classification, iterative queries</td>
</tr>
<tr>
<td></td>
<td>TensorFlow [13]</td>
<td>mutable states, dense tensors as data representation</td>
<td>image classification, language modeling</td>
</tr>
<tr>
<td></td>
<td>Naiad [65], GraphX [40]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertex Programming</td>
<td>Pregel [58]</td>
<td>vertex programming</td>
<td>SSSP</td>
</tr>
<tr>
<td></td>
<td>PowerGraph [39]</td>
<td>Gather-Apply-Scatter</td>
<td>page rank, SSSP, graph coloring, Triangle Count, LDA</td>
</tr>
<tr>
<td></td>
<td>PowerLyra [23], Gemini [84], CUBE [83], TuX² [79]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Filter-Process</td>
<td>Arabesque [72]</td>
<td>filter-process</td>
<td>motifs, cliques, FSM</td>
</tr>
<tr>
<td>Task Dispatching</td>
<td>STRADS [51]</td>
<td>task dispatching</td>
<td>Lasso, SGD MF, LDA</td>
</tr>
</tbody>
</table>

Table 1.1: Summary of Existing Systems
1.7.3 Graph Processing Systems

Some systems are designed to handle special subsets of applications, such as graph processing and graph mining and they abstract the application computation into a specialized workflow and allow user-defined functions as hooks to customize the computation. For example, the vertex programming model [58] requires applications to provide a data graph and a vertex program that execute on each vertex. The vertex program is restricted to access only its neighboring edges and vertices. With the known data graph, such restrictions allow the framework to rely on the program’s data access pattern to prefetch values before computation and partition the graph to balance workload and minimize inter-machine communication. While some work [57] graph coloring to execute independent vertices in parallel, most such systems still rely on a Bulk Synchronous Parallel or asynchronous model that either delays the updates until the end of executing all vertices or provides no consistency guarantees.
Chapter 2

Managed Communication for Data-Parallel Training [76]

1 In a data-parallel execution, inconsistency arises from conflicting parameter access from workers. While more sophisticated parallelization and scheduling may preserve dependence and avoid conflicting access, realizing such parallelization requires exploiting applications’ computation pattern. This typically requires nontrivial programmer effort. In this chapter, we present a mechanism for distributed shared memory to reduce inconsistency errors in machine learning training due to conflicting access and thus improve overall convergence time. Our key insights are, first, that spare network bandwidth exists due to bursty communication in distributed machine learning training; and, second, that some updates are more important for learning algorithm convergence, thus one can prioritize the communication of important updates using spare network bandwidth to make the up-to-date values available sooner.

2.1 Data-Parallel Distributed Training and Synchronization

2.1.1 Data-Parallel Machine Learning Training

Training a model is essentially finding a set of model parameter values that optimize a certain objective function. This is typically done using an iterative convergent learning algorithm, which can be described by Alg. 1.

<table>
<thead>
<tr>
<th>Algorithm 1: Serial Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t \leftarrow 0$</td>
</tr>
<tr>
<td>for $\text{iteration} = 1, \ldots, T$ do</td>
</tr>
<tr>
<td>for $i = 1, \ldots, N$ do</td>
</tr>
<tr>
<td>$A_{t+1} \leftarrow A_t \oplus \Delta(A_t, D_i)$</td>
</tr>
<tr>
<td>$t \leftarrow t + 1$</td>
</tr>
</tbody>
</table>

In Alg. 1, $A_t$ denotes the parameter values at time step $t$, and $D_i$ denotes the $i$-th mini-batch in the training dataset $D = \{D_i|1 \leq i \leq N\}$. $D_i$ may contain one or multiple data items. The update function $\Delta()$ computes the model updates from a mini-batch of data items and the current parameter values, which are applied to generate a new set of parameter values. $\Delta$ may include some tunable hyper-parameters, such as step size in a gradient descent algorithm which

1Most content in this chapter is borrowed from my SoCC’15 paper [76].
requires manual or automatic tuning for the algorithm to work well. \(\oplus\) represents the operation to apply parameter updates, which is usually addition. The algorithm repeats many *iterations* until \(A\) stops changing, i.e., converges. In each iteration, the algorithm takes a full pass over the training dataset.\(^2\)

*Data parallelism* distributes \(K\) data mini-batches \(\{D_k|1 \leq k \leq K\}\) to \(K\) workers. The workers compute \(\Delta (A_t, D_k)\) in parallel using the current parameter values. A common practice is to pre-partition the dataset \(D\) into \(K\) subsets \(D^1,\ldots,D^K\) and assign them to workers. The workers synchronize with each other by sending their updates to a master copy of parameters and fetching updated parameter values. We refer to each synchronization as a *clock*, which may consist of one or many mini-batches (i.e., parameter updates).

The size of a clock may be tuned to control the network communication overhead. For applications that require small amount of computation per mini-batch, a single clock may contain hundreds or thousands of data mini-batches in order to balance the communication overhead. Many existing work [28, 29, 79] synchronizes once per iteration, i.e., after a worker has processed all data samples in its data partition. When the updates are additive, data-parallel training can be expressed as Alg. 2 where workers synchronize once per pass of local data partition.

---

### Algorithm 2: Data Parallelism

\[ t \leftarrow 0 \]

\[ \text{for} \ iteration = 1,\ldots,T \text{ do} \]

\[ \text{for} \ i \leftarrow 1 \text{ to } K \text{ in parallel do} \]

\[ u_i \leftarrow \sum_{j=1}^{D_i} \Delta \left( A_t, D^j \right) \]

\[ \text{synchronize with master model copy} \]

\[ A_{t+1} \leftarrow A_t + \sum_{i=0}^{K} u_i \]

---

### 2.1.2 Parameter Servers Synchronization Schemes

Parameter Server is the name used to refer to a commonly used software architecture that provides a global address space for shared model parameters. A classic scheme for synchronization in distributed computing is Bulk Synchronous Parallel (BSP), which enforces a synchronization barrier based on data items processed. In the case of data parallel training, at each synchronization barrier, all workers communicate their updates to the parameter server and after all updates are received by the server, the server broadcasts the updated parameters back to workers to start the next clock. However, BSP suffers from the well-studied *stragglers* problem [17, 18, 26, 28], in which the synchronization barrier between clocks causes the computation to proceed only at the rate of the slowest worker in each clock. Another example is *Total Asynchronous Parallelization* (TAP) where the parameter values a worker uses is whatever instantaneous state in the server results from the aggregation of out-of-sync (hence inconsistent) updates from different workers. Although highly efficient and sometimes accurate, TAP does not enjoy a theoretical guarantee and can diverge due to late updates disrupting training.

In this chapter, we explore a middle ground between BSP and TAP, namely *bounded staleness parallelization* [43, 53], in which each worker maintains a possibly *stale* local copy of \(A\), where the degree of staleness is bounded by a target staleness threshold \(S\), i.e., no worker is allowed to be more than \(S\) clock units ahead of the slowest worker. This idea has been shown experimentally

\(^2\)Iteration has also been referred to as *epoch* in some literatures.
August 16, 2018
DRAFT

(a) Bösen Parameter Server Architecture

(b) Bösen Client API

Figure 2.1: Bösen System Architecture and Client API

to considerably improve time to convergence \[28, 53\], and theoretical analysis on convergence rates and error bounds has been reported for some applications \[32, 43\].

Although ML algorithms may tolerate bounded staleness and still achieve correct convergence, the algorithm performance (i.e., convergence per data sample processed) may suffer from staleness, resulting in suboptimal performance. Previous implementations of bounded staleness either trigger communication when reaching the staleness threshold or synchronization barriers \[28, 29\] or rely on explicit application control of communication \[53\]. Our goal is to implement a system that supports sophisticated yet effective consistency models and resource-driven communication.

Some parameter server implementations expose a Push/Pull interface and let its applications determine when and what to communicate \[53\]. Others hide communication from application programmers via a Get/Update interface and additionally provide a Clock API for application to report their progress to support the consistency model \[28, 29\]. Network communication is internally invoked by the Parameter Server to satisfy the consistency model. To provide a Parameter Server API that abstracts network communication away from users, we adopt the Get/Update interface.

2.2 Bösen Architecture and API

Bösen is a Parameter Server with a ML-consistent, bounded-staleness parallel scheme that uses bandwidth-managed communication mechanisms. It realizes bounded staleness consistency, which offers theoretical guarantees for iterative convergent ML programs (unlike TAP), while enjoying high iteration throughput that is better than BSP and close to TAP systems. Additionally, Bösen transmits model updates and up-to-date model parameters proactively without exceeding a bandwidth limit (i.e. before the end of a clock if spare network bandwidth is available), while making better use of the bandwidth by scheduling the bandwidth budget based on the contribution of the messages to algorithm progress — thus improving per-data-sample convergence compared to an agnostic communication strategy, that is, one that simply transmits all messages or a random subset.
Figure 2.2: Exemplar execution under bounded staleness (without communication management). The system consists of 5 workers, with staleness threshold $S = 3$; “clock $t$” refers to clock starting from $t$. Worker 2 is currently running in clock 4 and thus according to bounded staleness, it is guaranteed to observe all updates generated before (exclusively) clock $4 - 3 = 1$ (black). It may also observe local updates (green) as updates can be optionally applied to local parameter cache. Updates that are generated in completed clocks by other workers (blue) are highly likely visible as they are propagated at the end of each clock. Updates generated in incomplete clocks (white) are not visible as they are not yet communicated. Such updates could be made visible under managed communication depending on the bandwidth budget.

2.2.1 Bösen Client API

Bösen PS consists of a client library and parameter server partitions (Figure 2.1a); the former provides the Application Programming Interface (API) for reading/updating model parameters, and the latter stores and maintains the model $A$. In order to reduce network communication, where possible, the client library caches model parameter values and buffers updates. In terms of usage, Bösen closely follows other key-value stores: once a ML program process is linked against the client library, any thread in that process may read/update model parameters concurrently. The user runs a Bösen ML program by invoking as many server partitions and ML application compute processes (which use the client library) as needed, across multiple machines.

Bösen’s API abstracts consistency management and networking operations away from the application, and presents a simple key-value interface (Table 2.1b). Get() is used to read parameters and Inc() is used to increment the parameter by some delta. To maintain consistency, the client application signals the end of a unit of work e.g. a full pass over a worker’s data partition or a fixed number of data samples processed, via Clock(). In order to exploit locality in ML applications and thus amortize the overhead of operating on concurrent data structures and network messaging, Bösen allows applications to statically partition the parameters into batches called rows – a row is a set of parameters that are (usually) accessed together. A row can also be the unit of communication between client and server. RowGet() is provided to read a row by its key, and RowInc() applies a set of deltas to multiple elements in a row. Bösen supports user-defined “stored procedures” to be executed on each server—these can be used to alter the default increment behavior of Inc() and RowInc().

2.2.2 Bounded Staleness Consistency

The Bösen client library and server partitions have a consistency manager, which is responsible for enforcing consistency requirements on the local parameter cache $\tilde{A}$, thus ensuring correct ML program execution even under worst-case delays (whether computation or communication). The ML program’s tolerance to stale parameters is specified as the staleness threshold, a non-negative integer supplied by user.
The consistency manager works by blocking client worker threads when reading parameters, unless the local parameter cache $\tilde{A}$ has been updated to meet the consistency requirements. Bounded staleness puts constraints on parameter age; Bösen will block if any parameter in $\tilde{A}$ is older than the worker’s current clock by $S$ or more (i.e., $\text{CurrentClock}(\text{worker}) - \text{Age}(\tilde{A}) \geq S$), where $S$ is the user-defined staleness threshold. $\tilde{A}$’s age is defined as the oldest clock such that some updates generated within that clock are missing from $\tilde{A}$.

The bounded staleness model enjoys BSP-like ML execution guarantees, theoretically explored in [32, 43], which are absent from total asynchronous parallelization (TAP). Bösen’s bounded staleness consistency is closest to ESSP [32]. Similar to ESSP, Bösen propagates updates upon completion of each clock even though they are not required yet. With eager end-of-clock communication, parameters read typically have staleness 1 regardless of the staleness threshold as the end-of-clock communication typically completes within 1 clock. Managed communication allows updates to be propagated even earlier, before completing a clock. Our experiments used a staleness threshold of 2 unless otherwise mentioned, which has been demonstrated to be effective in Cui et al. [28]. An exemplar execution of 5 workers under Bounded Staleness is depicted in Fig 2.2.

**Bulk Synchronous Parallel (BSP).** When the staleness threshold is set to 0, bounded staleness consistency reduces to the classic BSP model. The BSP model requires all updates computed in previous clocks to be made visible before the current clock starts. A conventional BSP implementation may use a global synchronization barrier; Bösen’s consistency manager achieves the same result by requiring calls to $\text{PS.Get()}$ and $\text{PS.GetRow()}$ at clock $t$ to reflect all updates, made by any thread, before its $(t-1)$-th call to $\text{PS.Clock()}$ — otherwise, the call to $\text{PS.Get()}$ or $\text{PS.GetRow()}$ blocks until the required updates are received.

### 2.2.3 Client Library And Server Partitions

The client library provides access to the model parameters $A$ on the server partitions, and also caches elements of the model $A$ for faster access, while cooperating with server processes in order to maintain consistency guarantees and manage bandwidth. This is done through three components: (1) a **parameter cache** that caches a partial or complete image of the model, $\tilde{A}$, at the client, in order to serve read requests made by compute threads; (2) an **update buffer** that buffers updates applied by compute threads via $\text{PS.Inc()}$ and $\text{PS.RowInc()}$; (3) a group of **client communication threads** (distinct from compute threads) that synchronizes the local model parameter cache with the servers’ master copy, while the compute threads executes the application training algorithm.

The parameters cached at a client are hash partitioned among the client communication threads. Each client communication thread needs to access only its own parameter partition when reading the computed updates and applying up-to-date parameter values to minimize lock contention. The client parameter cache and update buffer allow concurrent reads and writes from worker threads, and similar to Cui et al. [29], the cache and buffer use static data structures, and pre-allocate memory for repeatedly accessed parameters to minimize the overhead of maintaining a concurrent hash table.

In each compute process, locks are needed for shared access to parameters and buffered update entries. In order to amortize the runtime cost of concurrency control, we allow applications to define parameter key ranges we call **rows** (as noted above). Parameters in the same row share one lock for accesses to their parameter caches, and one lock for accesses to their update buffers.

When serving read requests ($\text{Get()}$ and $\text{RowGet()}$) from worker threads, the client parameter cache is searched first, and a read request is sent to the server processes only if either the
requested parameter is not in the cache or the cached parameter’s staleness is not within the staleness threshold. The reading compute thread blocks until the parameter’s staleness is within the threshold. When writes are invoked, updates are inserted into the update buffer, and, optionally, the client’s own parameter cache is also updated.

Communication threads run on the same CPUs that the compute threads run on and take away CPU cycles from computation for marshalling. Client compute threads also compete with communication threads for mutual exclusive access on parameter cache and update buffer. Communication threads more frequently visit the parameter cache and update buffer when communicating more frequently. With a high bandwidth budget, the high-frequency communication may slow computation due to lock contention.

Once all compute threads in a client process have called \texttt{PS.Clock()} to signal the end of a unit of work (e.g. one clock), the client communication threads release buffered model updates to servers. Note that buffered updates may be released sooner under managed communication if the system detects spare network bandwidth to use.

The master copy of the model’s parameters, $A$, is hash partitioned, and each partition is assigned to one parameter server thread. The server threads may be distributed across multiple server processes and physical machines. As model updates are received from client processes, the addressed server thread updates the master copy of its model partition. When a client read request is received, the corresponding server thread registers a callback for that request; once a server thread has applied all updates from all clients for a given unit of work, it walks through its callbacks and sends up-to-date model parameter values.

Bounded staleness is ensured by coordination of clients and server partitions using clock messages. On an individual client, as soon as all updates generated before and in clock $t$ are sent to server partitions and no more updates before or in that clock can be generated (because all compute threads have advanced beyond that clock), the client’s communication threads send a client clock message to each server partition, indicating “all updates generated before and in clock $t$ by this client have been made visible to this server partition” (assuming reliable ordered message delivery).

After a server partition sends out all parameters modified in clock $t$, it sends an server clock message to each client communication thread, indicating “all updates generated before and in clock $t$ in the parameter partition have been made visible to this client”. Upon receiving such a clock message, the client communication thread updates the age of the corresponding parameters and permits the relevant blocked compute threads to proceed on reads if any.

### 2.3 Managed Communication in Bösen

Bösen’s client library and server partitions features a communication manager. The communication manager has two objectives: (1) communicate as many updates per second as possible (full utilization of the bandwidth budget) without overusing the network (which could delay update delivery and increase message processing computation overhead); and (2) prioritize more important model updates to improve ML learning progress per data pass. The first objective is achieved via bandwidth-driven communication with rate limiting, while the second is achieved by choosing a proper prioritization strategy.

#### 2.3.1 Bandwidth-Driven Communication

In order to fully utilize the given bandwidth budget without exceeding the threshold, Bösen communication threads query the communication manager for opportunities to communicate and sends small packets (from hundreds of KBs to a few MBs, referred to as “queue size”) each
time spare bandwidth is available. The communication manager keeps track of the number of bytes that is communicated and derive the time for next communication event based on network bandwidth budget.

**Coping with network fluctuations:** In real cloud or data centers with multiple users, the available network bandwidth may fluctuate and fail to live up to the bandwidth budget $B$. Hence, the Bösen communication manager regularly checks to see if the network is overused by monitoring how many messages were sent without acknowledgement in a recent time window (i.e. message non-delivery). If too many messages fail to be acknowledged, the communication manager assumes that the network is overused, and waits until the window becomes clear before permitting new messages to be sent.

**Update quantization:** Since ML applications are error tolerant, training may use low-precision floating-point or fixed-point numbers as demonstrated by a number of empirical studies [27, 41, 70]. Bösen applications have the option to communicate updates and parameter values in 16-bit floating point numbers, reducing bandwidth consumption in half compared to 32-bit floats.

### 2.3.2 Update Prioritization

Bösen spends available bandwidth on communicating information that contributes the most to convergence. For example, gradient-based algorithms (including Logistic Regression) are iterative-convergent procedures in which the fastest-changing parameters are often the largest contributors to solution quality — in this case, we prioritize communication of fast-changing parameters, with the largest-magnitude changes going out first. When there is opportunity for communication due to spare bandwidth, the server or client communication threads pick a subset of parameter values or updates to send. The prioritization strategy determines which subset is picked at each communication event. By picking the right subset to send, the prioritization strategy alters the communication frequency of different parameters, effectively allocating more network bandwidth to more important updates. It should be noted that the end-of-clock communication needs to send all up-to-date parameters or updates older than a certain clock number to ensure the consistency guarantees.

Bösen’s bandwidth manager supports multiple prioritization strategies. The simplest possible strategies are **Randomized**, where communications threads send out randomly-chosen rows and **Round-Robin**, where communication threads repeatedly walk through the rows following a fixed order, and sends out all non-zero updates or updated parameters encountered. These strategies are baselines; better strategies prioritize according to significance to convergence progress. We study the following two better strategies.

- **Absolute Magnitude prioritization**: Updates/parameters are sorted by their accumulated change in the buffer, $|\delta|$.
- **Relative Magnitude prioritization**: Same as absolute magnitude, but the sorting criteria is $|\delta/a|$, i.e. the accumulated change normalized by the current parameter value, $a$.

### 2.4 Evaluation

#### 2.4.1 Evaluation Setup

**Cluster setup:** Most of our experiments were conducted on the PRObE Nome cluster [38] consisting of 200 server computers running Ubuntu 14.04. Our experiments used different number of computers, varying from 8 to 64. Each machine contains $4 \times$ quad-core AMD Opteron 8354 CPUs (16 physical cores per machine) and 32GB of RAM. The machines are distributed over multiple racks and connected via a 1 Gbps Ethernet and 20 Gbps Infiniband. A few experiments
### Table 2.1: Bösen system and application configurations. The queue size (in number of rows) upper bounds the send size to control burstiness; the first number denotes that for client and the second for server. LDA experiments used hyper-parameters $\alpha = \beta = 0.1$. Experiments whose bandwidth budget exceeds 1Gbps used 20Gbps infiniband or 40Gbps Ethernet otherwise they used 1Gbps Ethernet.

<table>
<thead>
<tr>
<th>Application &amp; Dataset</th>
<th>Cluster</th>
<th># Machines</th>
<th>Per-node Bandwidth Budgets</th>
<th>Queue Size</th>
<th>Initial Step Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD MF, Netflix</td>
<td>Nome</td>
<td>8</td>
<td>200Mbps, 800Mbps</td>
<td>100, 100</td>
<td>0.08</td>
</tr>
<tr>
<td>LDA, NYTimes</td>
<td>Nome</td>
<td>16</td>
<td>320Mbps, 640Mbps, 1280Mbps</td>
<td>5000, 500</td>
<td>N/A</td>
</tr>
<tr>
<td>LDA, ClueWeb10%</td>
<td>Nome</td>
<td>64</td>
<td>800Mbps</td>
<td>5000, 500</td>
<td>N/A</td>
</tr>
<tr>
<td>MLR, ImageNet5%</td>
<td>Susitna</td>
<td>4</td>
<td>100Mbps, 200Mbps, 1600Mbps</td>
<td>1000, 500</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.2: Descriptions of ML models and datasets. Workload refers to the total number of data samples in the input data set.

<table>
<thead>
<tr>
<th>Application</th>
<th>Dataset</th>
<th>Workload</th>
<th>Description</th>
<th># Rows</th>
<th>Data Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGD MF</td>
<td>Netflix</td>
<td>100M ratings</td>
<td>480K users, 18K movies, rank=400</td>
<td>480K</td>
<td>1.3GB</td>
</tr>
<tr>
<td>LDA</td>
<td>NYTimes</td>
<td>99.5M tokens</td>
<td>300K documents, 100K words, 1K topics</td>
<td>100K</td>
<td>0.5GB</td>
</tr>
<tr>
<td>LDA</td>
<td>ClueWeb10%</td>
<td>10B tokens</td>
<td>50M webpages, 160K words, 1K topics</td>
<td>160K</td>
<td>80GB</td>
</tr>
<tr>
<td>MLR</td>
<td>ImageNet5%</td>
<td>65K samples</td>
<td>1000 classes, 21K of feature dimensions</td>
<td>1K</td>
<td>5.1GB</td>
</tr>
</tbody>
</table>

were conducted on the PRObE Susitna cluster [38]. Each Susitna machine contains $4 \times 16$-core AMD Opteron 6272 CPUs (64 physical cores per machine) and 128GB of RAM. The machines are distributed over two racks and connected to two networks: 1 Gbps Ethernet and 40 Gbps Ethernet. In both clusters, every machine is used to host Bösen server, client library, and worker threads (i.e. servers and clients are collocated and evenly distributed).

**Performance metrics:** Our evaluation measures performance as the absolute convergence rate on the training objective value; that is, our goal is to reach convergence to an estimate of the model parameter $\theta$ that best represents the training data (as measured by the training objective value) in the shortest time. Once trained, the model’s performance on application tasks also depends on the selection of an objective function and proper regularization, which are usually the purview of user expertise and are out of the scope of this research.

Bösen is executed under different modes in this section:

**Single Node:** The ML application is run on one shared-memory machine linked against one Bösen client library instance with only consistency management. The parameter cache is updated upon write operations. Thus updates become immediately visible to compute threads. It represents a gold standard when applicable. It is denoted as “SN”.

**Linear Scalability:** It represents an ideal scenario where the single-node application is scaled out and linear scalability is achieved. It is denoted as “LS”.

**Bounded Staleness:** Bösen is executed with only consistency management enabled, that is, communication management is disabled. It is denoted as “BS”.

**Bounded Staleness + Managed Communication:** Bösen is executed with both consistency and communication management enabled. It is denoted as “MC-X-P”, where $X$ denotes the per-node bandwidth budget (in Mbps) and $P$ denotes the the prioritization strategy: “R” for Randomized, “RR” for Round-Robin, and “RM” for Relative-Magnitude.

**Bounded Staleness + Fine-Grained Clock Tick Size:** Bösen is executed with only consistency management enabled, communication management is disabled. In order to communicate up-
Figure 2.3: Algorithm performance under managed communication

Figure 2.4: Topic Modeling with Latent Dirichlet Allocation

2.4.2 Evaluating Managed Communication

In this section, we show that the algorithm performance improves with more immediate communication of updates and model parameters. Moreover, proper bandwidth allocation based on the importance of the messages may achieve better algorithm performance with less bandwidth consumption. To this end, we compared managed communication with the standard static communication schedule (i.e., only the consistency manager is enabled). The commu-
Effect of increasing bandwidth budget. We demonstrate the effect of larger bandwidth budget via the Matrix Factorization (MF) and Latent Dirichlet Allocation (LDA) experiments (Fig. 2.3). First of all, we observed that enabling communication management significantly reduces the number of iterations needed to reach convergence (objective value of $2e7$ for MF and $-1.022e9$ for LDA). In MF, communication management with bandwidth budget of 200Mbps reduces the number of iterations needed to reach $2e7$ from 64 (BS) to 24 (MC-200-R). In LDA, a bandwidth budget of 320Mbps reduces the number of iterations to convergence from 740 (BS) to 195 (MC-320-R). Secondly, increasing the bandwidth budget further reduces the number of iterations needed. For example, in LDA, increasing the bandwidth budget from 320Mbps (MC-320-R) to 640Mbps (MC-640-R) reduces the number iterations needed from 195 to 120.

Effect of prioritization. As shown Fig. 2.3(b), in the case of LDA, prioritization by Relative-Magnitude (RM) consistently improves upon Randomization (R) when using the same amount of bandwidth. For example, with 320Mbps of per-node bandwidth budget MC-320-RM reduces the number of iterations needed to reach $-1.022e9$ from 195 (MC-320-R) to 145.

Prioritization appears to be less effective for MF. The server user defined functions computes the step size which scales the gradient, altering the gradient by up to orders of magnitude. Since the adaptive revision algorithm tends to apply a larger scaling factor for smaller gradients [60], the raw gradient magnitude is a less effective indicator of significance.
Overhead of communication management and absolute convergence rate. Under managed communication, the increased volume of messages incurs noticeable CPU overheads due to sending and receiving the messages and serializing and deserializing the content. Computing importance also costs CPU cycles. Fig. 2.5 presents the per-iteration runtime and network bandwidth consumption corresponding to Fig. 2.3. Fig. 2.5(a) shows that Bulk Synchronous Parallel has fast iterations, that do not fully utilize network bandwidth but also makes slow progress per iteration as shown in Fig. 2.3(a).

Fig. 2.5 shows, for example, enabling communication management with a 200Mbps bandwidth budget (MC-200-R) incurs a 12% per-iteration runtime overhead relative to BS. However, the improved algorithm performance significantly outweighs such overheads and results in much higher absolute convergence rate in wall clock time, as shown in Fig. 2.6 (MF and MLR) and Fig. 2.7(a). For example, for MF, we observed a $2.5 \times$ speedup in absolute convergence rate using bandwidth budget of 800Mbps and Relative-Magnitude prioritization compared the bounded staleness baseline.

Comparison with Yahoo!LDA. We also compare Bösen LDA with the popular Yahoo!LDA using the NYTimes and 10% of the ClueWeb [3] data set, using 1Gbps Ethernet and 20Gbps Infiniband respectively. The former is plotted in Fig. 2.4(b). Yahoo!LDA employs a Parameter Server architecture that’s similar to Bösen’s, but uses total asynchronous parallelization. The compute threads of Yahoo!LDA process roughly the same number of data points as Bösen. Each Yahoo!LDA worker (node) runs one synchronizing thread that iterates over and synchronizes all cached parameter in a predefined order. We observed that Bösen significantly outperformed Yahoo!LDA on the NYTimes dataset, but converged at similar rate on the ClueWeb10% data set.

2.4.3 Comparing with Manual Mini-batch Size Tuning

In summary, by making full use of the 800Mbps and 640Mbps bandwidth budget, communication management with Randomized prioritization improved the time to convergence of the MF and LDA application by $2.5 \times$ and $2.8 \times$ in wall clock time and $5.3 \times$ and $6.1 \times$ in number of iterations, compared to a bounded staleness execution. Relative-Magnitude prioritization further improves the convergence time of LDA by 25%. Communication management with bandwidth budget of 200Mbps and Relative-Magnitude prioritization improved the convergence time of MLR by $2.5 \times$ (Fig 2.6(b)).

Another way of reducing parallel error on a BSP or bounded staleness system is to divide a full data pass into multiple clocks to achieve more frequent synchronization, while properly
adjusting the staleness threshold to ensure the same staleness bound. This approach is similar to mini-batch size tuning in ML literature. We compare Bösen’s communication management with application-level clock tick size tuning via the LDA application and the result is plotted in Fig 2.7. For each setting, we adjust the staleness threshold so all runs share the same staleness bound of 2 data passes.

Firstly, from Fig. 2.7(b) we observe that as the clock tick size halves, the average bandwidth usage over the first 280 iterations doubles but the average time per iteration doesn’t change significantly. From Fig. 2.7(a), we observe that the increased communication improves the algorithm performance. Although simply tuning clock tick size also improves algorithm behavior, it doesn’t enjoy the benefit of prioritization. For example, MC-640-RM used only 63% of the bandwidth compared to BS-8 but converged 28% faster. The difference is due to careful optimization which cannot be achieved via application-level tuning.

2.5 Conclusion

We demonstrated that in data-parallel training, spare network bandwidth and message prioritization can be exploited to improve convergence rate. In our prototype implementation, too frequent communication reduces application computation throughput due to marshalling that competes for CPU cycles and lock contention. Nevertheless, managing network communication improves overall convergence rate by 2 to 3×. Further improvement is possible by reducing marshalling overhead and more efficient concurrency control on shared data structures.
Chapter 3

Dependence-Aware Parallelization

We present a system Orion that uses static dependence analysis to automate dependence-aware parallelization of serial machine learning programs for distributed execution, exploiting access sparsity. Besides using classic compiler techniques, Orion supports programmer manageable violation of dependence to trade learning progress for computation throughput. By automatically parallelizing imperative application programs implemented in a scripting language, Orion reduces the lines of code of application programs by up to 11.6× compared to manually parallelized machine learning programs, with a convergence rate that’s at least comparable. Orion parallelized programs achieve a comparable and sometimes lower processing throughput (up to $1.5 - 2.46 \times$) compared to hand-optimized C++ parallel implementations, mainly due to language differences.

3.1 Introduction

3.1.1 Machine Learning Training

Machine Learning (ML) training finds the parameter values of a parametric model that minimizes (or maximizes) certain objective function by processing a set of data items. Let $A_t$ denote the parameter values at time step $t$, $D$ denote the training dataset and $D = \{D_i | 1 \leq i \leq N\}$ where $D_i$ denotes the $i$-th mini-batch which may contain one or multiple data items. The core of a typical ML training program can be expressed as sequentially executing an update equation for each mini-batch $D_i$, which repeats until some stopping criteria are met (Alg. 3):

\begin{algorithm}
\caption{Serial Execution}
\begin{algorithmic}
\State $t \leftarrow 0$
\While{not converged}
\For{$i = 1, ..., N$}
\State $A_{t+1} \leftarrow A_t \oplus \Delta(A_t, D_i)$
\State $t \leftarrow t + 1$
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

In Alg. 3, $\Delta$ denotes a generic function that computes parameter refinements using the current parameter values and a mini-batch of data, such as computing gradients in stochastic gradient descent. The parameter values are then updated using the generic operator $\oplus$.

As each mini-batch reads and updates model parameters $A$, there is data dependence between mini-batches. It is usually equivalent to process mini-batches in any sequential order as different orderings result in different but equally good parameter values. Therefore, data
dependence among mini-batches indicates only mutual exclusiveness but not ordering. When mini-batches are processed in parallel, serializability is sufficient and necessary to ensure equivalence between parallel and serial execution [57].

3.1.2 Data Parallelism

Data parallelism distributes $K$ data mini-batches $\{D_{i+k-1}\}_{1 \leq k \leq K}$ to $K$ workers. The workers compute $\Delta(A_t, D_{i+k-1})$ in parallel using the current parameter values and their assigned data mini-batch (Alg. 4). Conceptually, there exists a master copy of parameters that is updated using updates $\{u_k\}_{1 \leq k \leq K}$ computed by workers, and is then distributed to workers for processing the next mini-batch.

Algorithm 4: Data Parallelism

\[
\begin{align*}
    t &\leftarrow 0 \\
    \text{while not converged do} \\
    &\text{for } i \leftarrow 1 \text{ to } N \text{ by } K \text{ do} \\
    &\quad \text{for } k = 1, \ldots, K \text{ in parallel do} \\
    &\quad & u_k &\leftarrow \Delta(A_t, D_{i+k-1}) \\
    &\quad \text{for } k = 1, \ldots, K \text{ do} \\
    &\quad & A_{t+1} &\leftarrow A_t \oplus u_k \\
    &\quad & t &\leftarrow t + 1
\end{align*}
\]

Technically data parallelism is not equivalent to serial execution because the all $K$ workers compute updates using the same parameter values $A_t$ and updates are not applied until all $K$ workers have finished processing their mini-batch (not serializable). In a serial execution, parameter values are updated after each mini-batch and the updated parameters are used for processing the next mini-batch. Under data parallelism, updates are computed from a data mini-batch $D_{t+i}$ using a staler version of parameters compared to a serial execution ($A_t$ vs. $A_{t+i}$). The staleness grows with increasing number of workers.

Nevertheless, data parallelism may still train a valid model. First of all, in some algorithms such as stochastic gradient descent, the mini-batch size $|D_i|$ is an algorithmic hyper-parameter and data items within a mini-batch are processed independently from each other. Data parallelism over $K$ workers is thus equivalent to computing updates from a mini-batch of size $\sum_{k=1}^K |D_{i+k-1}|$, which can be viewed as a form of hyper-parameter tuning. Secondly, iterative convergent ML algorithms tolerate bounded error [43, 69]. These algorithms still converge to a reasonable (but probably different) solution even when the computation involves bounded error. Since $A_t$ is usually a close enough approximation of $A_{t+i}$, data parallelism still converges. However, it is known that increasing staleness slows convergence. The, the learning algorithm needs to process more data items (i.e. more iterations) to reach the same model quality [43, 51, 76]. When training neural networks using stochastic gradient descent, it has been widely observed that the model’s performance degrades significantly on test dataset when mini-batch size is too large. This problem is referred to as the generalization gap [44, 50].

3.1.3 Dependence-aware Parallelization

Depending on the model and the training algorithm, many ML programs access parameters sparsely, where the $\Delta(\cdot)$ computation 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 $\Delta (A_t, \mathcal{D}_k)$ computation are disjoint, then the parallel execution is serializable. Thus the key challenge is to divide the training dataset into groups of mini-batches such that the mini-batches within each group are independent in terms of parameter access. \footnote{Technically, two mini-batches $\mathcal{D}_i$ and $\mathcal{D}_j$ are independent if $\text{ReadSet}(\mathcal{D}_i) \cap \text{WriteSet}(\mathcal{D}_j) = \emptyset$ and $\text{ReadSet}(\mathcal{D}_j) \cap \text{WriteSet}(\mathcal{D}_i) = \emptyset$.} We refer to this style of parallelization that preserves data dependence among mini-batches as \textit{dependence-aware parallelization} (Alg. 5).

**Algorithm 5: Dependence-aware Parallelization**

$t \leftarrow 0$

\[ \textbf{while not converged do} \]

\[ \textbf{while } \mathcal{D} \text{ not empty do} \]

\[ \{ \mathcal{D}_{t_1}, ..., \mathcal{D}_{t_k} \} \leftarrow \text{get } K \text{ independent mini-batches in } \mathcal{D} \]

\[ \mathcal{D} \leftarrow \mathcal{D} - \{ \mathcal{D}_{t_1}, ..., \mathcal{D}_{t_k} \} \]

\[ \text{for } k = 1, ..., K \text{ in parallel do} \]

\[ u_k \leftarrow \Delta (A_t, \mathcal{D}_{t_k}) \]

\[ \text{for } k = 1, ..., K \text{ do} \]

\[ A_{t+1} \leftarrow A_t \oplus u_k \]

\[ t \leftarrow t + 1 \]

Fig. 3.1 compares data parallelism with dependence-aware parallelization on a toy example. Data parallelism randomly assigns mini-batches to workers $W_1$ and $W_2$ regardless of which parameters are accessed for processing each mini-batch. The parallel execution is not serializable as it is not equivalent to any sequential ordering of the mini-batches. With dependence-aware parallelization, different workers do not access the same parameters at the same time. The parallel execution is equivalent to sequentially processing mini-batches $\mathcal{D}_1, \mathcal{D}_4, \mathcal{D}_2,$ and $\mathcal{D}_3$.

This property has been exploited in previous work, which demonstrated that training algorithms converge considerably faster when computation is scheduled to avoid conflicting parameter access compared to data parallelism \cite{51}. However, existing systems such as STRADS \cite{51} require the training program to be manually parallelized to take advantage of this opportunity, which demands heavy programmer effort and is error-prone. In this chapter we present a system Orion that automates dependence-aware parallelization of imperative ML programs for efficient distributed execution, exploiting sparse parameter access. Orion statically analyzes data dependence of the serial program and parallelizes the heavy computation (for-loops) while preserving its dependence.

---

**Figure 3.1:** Data parallelism vs. dependence-aware parallelism

---

1. Technically, two mini-batches $\mathcal{D}_i$ and $\mathcal{D}_j$ are independent if $\text{ReadSet}(\mathcal{D}_i) \cap \text{WriteSet}(\mathcal{D}_j) = \emptyset$ and $\text{ReadSet}(\mathcal{D}_j) \cap \text{WriteSet}(\mathcal{D}_i) = \emptyset$. 

---
Distributed machine learning systems commonly use distributed shared memory (e.g. Parameter Server) for efficiently sharing model parameters among workers [13, 51, 76]. While an imperative programming model and a shared memory abstraction is expressive and natural to programmers, parallelization is much harder compared to functional programming with immutable objects. The challenge arises from the difficulty of accurately capturing fine-grained dependence among memory operations, efficient parallelization and concurrency control. Decades of research on automatic parallelizing compilers have shown that static dependence analysis can enable dependence-preserving parallelization. These techniques are applicable to machine learning programs. However, traditional techniques are not enough to exploit ML-specific properties. Our key insight is that error-tolerant machine learning programs have dependence that’s critical to preserve as well as less critical dependence allowing manageable violation of dependence to enable parallelization that’s otherwise impossible.

We evaluated Orion on a number of machine learning applications and compared Orion to three representative machine learning systems, including a Parameter Server (Bösen [76]), a model scheduling system (STRADS [51]) and a dataflow system (TensorFlow [13]). Programs parallelized by Orion achieve a convergence rate that’s at least comparable to manual parallelization, while reducing lines of code needed by up to 90%.

3.2 Motivation

Various distributed systems have been developed for offline machine learning training. We categorize those systems according to their programming model and summarize their major differences in Table 3.1. In this section, we discuss why existing systems fail to automate dependence-aware parallelization and motivate Orion’s design.

### 3.2.1 Batch Dataflow Systems

Many systems [13, 64, 80, 82] adopt a dataflow execution model, where a Directed Acyclic Graph (DAG) is constructed first to describe the computation to be performed, and the computation DAG is lazily evaluated when certain output is requested. For example, in Spark [82]’s computation DAG, each node 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 in order to achieve deterministic execution, their computation DAG contains no mutable states which makes updating model parameters an expensive operation. For example, aggregating updates and broadcasting updated model parameters to workers takes 20 seconds on 5 EC2 nodes as reported by Moritz et al. [63].

### 3.2.2 TensorFlow

TensorFlow [13] is a deep learning system which also adopts a dataflow programming model, where nodes of the computation DAG represent operations whose inputs and outputs are multi-dimensional tensors flowing along the direction of the edges. TensorFlow overcomes the limitation of traditional dataflow systems by adding *variable* and *queue* operations to its computation DAG, which contain mutable states that are shared by different executions on the same graph.

A typical TensorFlow program constructs a computation DAG that implements the update equation in Alg. 3 for a single mini-batch of data, where variables are used to hold trainable model parameters. During training, the computation DAG is executed repeatedly for all data mini-batches. By using a large set of parallelized operators and executing multiple operation in parallel, TensorFlow utilizes multiple cores of a computing device (e.g. a CPU, GPU or TPU). However, for distributed training, TensorFlow relies on application programmers to manually assign devices to operations in the graph for distributing the computation. Distributed TensorFlow applications typically use data parallelism by replicating the computation DAG on each device and sharing the same set of model parameter variables among all replicas.

Even if TensorFlow was able to automatically parallelize the computation DAG across devices, automating dependence-aware parallelization requires a computation DAG that contains every data mini-batch and their dependence to model parameters. The size of this graph is proportional to training dataset size. Such a graph might be prohibitively expensive to store and analyze.

Moreover, in the computation DAG, access to variables or queues alone doesn’t indicate mutable exclusiveness nor ordering between operations that access variables or queues, which would have to be explicitly specified by the application program (for example, by using control dependencies [1]). Consider an example where two data mini-batches $D_1$ and $D_2$ are processed. The computation DAG in Fig. 3.2 allows $D_1$ and $D_2$ to be processed in parallel despite their conflicting access to variable $x$. Ensuring mutual exclusiveness requires a dependence edge between one mini-batch’s `AssignAdd` operation that updates variable $x$ and the other mini-batch’s `Read` operation. As all dependence edges are directed, it is impossible to enforce mutable exclusiveness without ordering in TensorFlow, which may restrict the degree of parallelism. Constructing a computation DAG that’s amenable to parallelization is labor-intensive and error-prone.

### 3.2.3 Graph Processing Systems

Graph processing systems [23, 39, 56, 57, 79, 83, 84] take a user-provided data graph as input and execute a vertex program on each vertex. Since a vertex program is restricted to access only data that is stored in the vertex’s edges and its neighboring vertices, the graph naturally describes the dependence among vertex programs. This allows some systems to achieve serializability using graph coloring or pessimistic concurrency control [39, 56, 57]. There are many applications that deal with non-graph data, which would require users to manually generate a dependence graph and the size of the graph is quadratic to the number of mini-batches. More importantly, it was observed that ensuring serializability using graph coloring or pessimistic concurrency control considerably reduces computation throughput [39], and later systems typically adopt a bulk-
synchronous or asynchronous engine and give up serializability \[23, 79, 83, 84\].

3.2.4 Parameter Server Systems

Parameter Server provides a mutable shared memory abstraction. Typically Parameter Server applications are manually parallelized by programmers using data parallelism and application programmers implement a program that executes on each worker machine \[29, 53, 76\].

In order to exploit dependence-aware parallelization, STRADS \[51\] proposes to schedule independent update computation to distributed workers. STRADS relies on application programmers to manually parallelize the program and implement a scheduler. Moreover, the worker program might become more complicated in order to efficiently support a computation schedule. For example, the worker program might need to implement pipelined communication to overlap communication and computation time. Although STRADS applications achieve state-of-the-art performance on many sparse learning problems, implementing a STRADS application requires significant programmer effort and STRADS programs typically contain at least several times more lines of code than a corresponding serial implementation.

3.2.5 Orion Design Summary

Imperative programming. Section 3.2.2 has discussed the challenges a dataflow system faces for automating dependence-aware parallelization. Moreover, the dataflow model presents a challenge for most programmers who are used to imperative programming. The dataflow model requires dynamic control flow to be expressed within the graph \[81\] and is more difficult to understand and debug. In order to overcome the usability limitation of the dataflow execution model TensorFlow additionally supports eager execution for imperative programming \[4\]. However, TensorFlow’s eager mode can’t be used in distributed execution \[5\].

Orion adopts imperative programming. Orion application programmers implement a serial driver program in Julia \[21\] that executes instructions locally or in Orion’s distributed runtime. Orion is designed as a Julia library without changing the Julia language. The driver program may use any Julia control flow primitive for dynamic control flow. This design also allows the driver program to use any existing Julia library, such as automatic differentiation.

Mutable shared memory. As discussed above, shared mutable states is essential for efficiently storing the frequently updated model parameters. Moreover, shared memory offers a convenient programming abstraction. Orion’s shared memory abstraction is a distributed multi-dimensional array, which is referred to as DistArray. A DistArray supports both set operations such as map and groupBy and random access.

Distributed parallel for-loops. Orion provides several macros for executing Julia instructions in the distributed cluster. Among them the most important one is @parallel_for that parallelizes a serial for-loop whose body reads and writes to DistArrays. As shown in Alg. 3, a
for-loop is natural for describing the loop that sequentially processes the data mini-batches, where the set of mini-batches constitute the loop’s iteration space. It is left to Orion to transform the serial for-loop to a dependence-aware parallelization (Alg. 5), where the key is to find independent mini-batches.

**Static dependence analysis.** Automatic parallelizing compilers parallelizes serial for-loops while preserving loop-carried dependence using static dependence analysis. Static dependence analysis uses dependence vectors to concisely represent data dependence. A single dependence vector represents a dependence between any two iterations whose difference matches the dependence vector. Compared to materializing a full dependence graph, dependence vectors are much more compact and more efficient to analyze. However, it might introduce false positive dependences that potentially restricts parallelization.

**Semantic relaxation and programmer hints.** Orion supports multiple natural relaxations and programmer hints to take advantage of programmers’ domain knowledge for generating efficient distributed programs. Among them one important relaxation is to allow programmers to selectively discard some write accesses for dependence analysis. This permits parallelization of programs that aren’t parallelizable otherwise by violating less sensitive dependence. In the extreme, discarding all write accesses enables regular data parallelism.

**Application programming in a scripting language (Julia [21]).** Scripting languages greatly improves programmer productivity at the cost of system efficiency [66]. We choose Julia for its productivity and efficiency. Although a Julia program is often still slower than its well optimized C or C++ counterpart. We believe a $2 \times$ to $3 \times$ overhead is reasonable to pay considering the substantial improvement in programmer productivity.

### 3.3 Orion’s Programming Model

Orion provides an application library and a runtime system. The runtime system consists of a master process and a number of distributed worker executors. Application programmers write a driver program in Julia [21] that uses the application library to dispatch computation and manipulate data stored in the distributed executors by executing commands on Orion’s runtime master.

**3.3.1 Scripting for Productivity With Small Performance Loss**

Python is probably the most popular programming language for machine learning today. As a scripting language, Python brings a rich set of language features and libraries and allows programmers to quickly “glue” functions from various libraries to create an application. The productivity gain from scripting languages has been advocated before [66], and Python’s popularity makes a great argument for it. However, Python suffers low performance due to its interpreter overhead and there are several high-profile but yet failed attempts to use JIT compilation to improve Python performance (Unladen Swallow [11] and Pyston [9]). In order to bridge that performance gap, Python programmers often rely on efficient kernels implemented in system languages for heavy computation, such as NumPy [8].

Conceptually, the techniques presented in this chapter can be implemented for any imperative programming language including Python. Orion currently supports a programming interface in Julia in order to achieve computation efficiency while enjoying the high productivity of a scripting language. Julia offers a rich set of language features with a simple syntax and uses a sophisticated type inference algorithm and JIT compilation to generate efficient native instructions. The productivity gain and efficiency makes Julia a promising language for machine learning [45]. Fig. 3.6 shows a function that computes some statistics of a random matrix imple-
August 16, 2018
DRAFT

function randmatstat(t)
    n = 5
    v = zeros(t)
    w = zeros(t)
    for i=1:t
        a = randn(n,n)
        b = randn(n,n)
        c = randn(n,n)
        d = randn(n,n)
        P = [a b c d]
        Q = [a b; c d]
        v[i] = trace((P'*P)^4)
        w[i] = trace((Q'*Q)^4)
    end
    return (std(v)/mean(v), std(w)/mean(w))
end

Figure 3.4: Julia Implementation

def randmatstat(t):
    n = 5
    v = zeros(t)
    w = zeros(t)
    for i in range(t):
        a = randn(n, n)
        b = randn(n, n)
        c = randn(n, n)
        d = randn(n, n)
        P = concatenate((a, b, c, d), axis=1)
        Q = concatenate((concatenate((a, b), axis=1),
                         concatenate((c, d), axis=1)), axis=0)
        v[i] = trace(matrix_power(dot(P.T,P), 4))
        w[i] = trace(matrix_power(dot(Q.T,Q), 4))
    return (std(v)/mean(v), std(w)/mean(w))

Figure 3.5: Python Implementation

Figure 3.6: Comparing Julia and Python syntax - computing random matrix statistics

The Julia program executes 10× faster than Python’s (Python uses NumPy [8] for linear algebraic operations). A greater gap of 39× was shown on quick sort.

Julia offers a number of features that are particularly attractive for Orion:

1. A Julia program is JIT compiled. To be more specific, when a global statement is first encountered, the statement is compiled and executed before its following statements are compiled. JIT compilation allows compilation to access the program’s runtime information (such as DistArray’s size), which Orion use to generate efficient parallelization.

2. Julia provides a powerful meta-programming system via macros. A Julia macro is a function that takes an abstract syntax tree (AST) as input and produces a new AST. When a Julia statement is compiled, the macros that are applied to it are invoked first to rewrite that statement. Orion takes advantage of this feature to implement many of its functionalities, including the @parallel_for macro that rewrites a serial for-loop into a sequence of statements that distributes the loop’s computation to Orion’s distributed runtime.

3. Because of JIT compilation, Julia allows generating Julia code at runtime, including macro expansion. This makes it easy for Orion to generate a number of functions, including bulk prefetching to improve execution efficiency without burdening the application programmer.

However, one major limitation of Julia is that it does not support multi-threading. ² Orion’s runtime is implemented in C++ to achieve greater efficiency.

²This was true when the project Orion started (Julia 0.5). Julia recently added an experimental interface for multi-threading in its latest release.
3.3.2 Distributed Arrays

Orion’s main abstraction for distributed shared memory is a multi-dimensional matrix, which we refer to as a Distributed Array or DistArray. An application program typically uses multiple DistArrays. DistArray implements Julia’s native AbstractArray interface for compatibility. Elements of a DistArray can be of any Julia type and a DistArray can be either dense or sparse. A DistArray is automatically partitioned and stored across multiple distributed executors. DistArrays support both set transformations and random access.

Set Transformations. Similar to RDDs, 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 fusing the user-defined functions across operations and avoids memory allocation for intermediate results. The application program may thus load data into memory once and extract different features into different DistArrays using different maps. Unlike RDDs, set operations that may cause shuffling such as groupBy that are evaluated eagerly for simplicity.

Random Access. The driver program may use the @parallel_for primitive (see Section 3.3.3) to distribute the computation of a for-loop to a cluster of machines. The for-loop’s loop body may randomly read and write to elements of DistArrays that have been materialized before the loop. During parallelization of the for-loop, Orion repartitions the accessed DistArrays to minimize remote accesses.

3.3.3 Parallel For-Loop

The driver program may iterate over a DistArray’s elements using a vanilla Julia for-loop and the loop body of different iterations can be distributed among Orion workers by applying Orion’s @parallel_for macro. Thus the DistArray’s key space naturally represents the loop’s iteration space. As DistArrays are multi-dimensional, such a for-loop conveniently represents a perfectly nested loop.

Since the loop body may randomly read and write to elements of any DistArray that has been created before the loop, different iterations may have a loop-carried dependence between them if their computation accesses the same DistArray element and one of the accesses is a write. In Orion’s parallelization, the unit of scheduling is one iteration, which is the same as in many automatic parallelizing compilers including [77] and [31]. Scheduling individual instructions [22, 54] may potentially be able to parallelize cases where Orion couldn’t. However, implementation requires much more effort and the additional benefit is low for machine learning programs according to our experience. Since the loop body is always executed serially in Orion, loop independent dependences (i.e. dependences that exists between different statements within the loop body) are always preserved.

Let $\mathcal{P} = \{(p_1, p_2, ..., p_n) | \forall i \in [1,n] : 1 \leq p_i \leq s_i\}$ represent the iteration space of a $n$-dimension 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 is able to find a parallel execution that ensures the loop-carried dependence if one of the following cases is true:

1. 1D Parallelization: There exists a dimension $i$ such that when $p_i \neq p'_i$, iteration $\vec{p}$ and iteration $\vec{p'}$ do not have any loop-carried dependence between them;

2. 2D Parallelization: There exist two dimensions $i$ and $j$ such that when $p_i \neq p'_i$ and $p_j \neq p'_j$, iteration $\vec{p}$ and iteration $\vec{p'}$ do not have any loop-carried dependence;
3. **2D Parallelization w/ Unimodular Transformation**: The dependence vectors have no elements that are negative infinity or infinity. In this case, Orion applies a classic technique that uses unimodular transformations to transform the iteration space to a parallelizable form [77]. The 2D parallelization can be applied.

In machine learning training programs, for-loops typically traverse a dataset or dimensions of the parameter space. Even though different ordering may result in a different set of numerical values as the learned model parameters, they are usually equally acceptable. This freedom permits an opportunity to reorder loop iterations for improving parallelism. By default, Orion assumes the loop to be parallelized is unordered. This means dependence between iterations only indicates mutual exclusivity, not ordering. Application programmers may enforce loop ordering by applying the `ordered` argument to `@parallel_for`.

The loop body may also read variables that are defined in the loop’s parent scope. In order to resemble shared memory programming, Orion automatically takes a snapshot of these variables and broadcasts their values to Orion executors that execute the loop body. The snapshot and broadcast are performed by statements generated by Orion, which are inserted before the parallel loop itself. Since a parallel loop may be executed multiple times (e.g. the parallel loop itself is executed inside a loop), this makes sure that Orion correctly captures the up-to-date variable values for distributed execution.

It should also be noted that the driver program may contain any number of parallel for-loops, iterating over and accessing the same or different DistArrays. This is a lot more flexible than graph processing systems such as PowerGraph [39] which is essentially a single for-loop that iterates over the vertices or edges; and Parameter Server systems, which allow applications to iterate over the datasets but usually not model parameters stored in the server.

### 3.3.4 Distributed Array Buffers

Orion applies static dependence analysis on DistArray accesses to find independent computation for parallelization. This means that each DistArray access may create a dependence vector, making parallelization harder. Application programmers may selectively exempt less critical DistArray writes from dependence analysis by applying them to Distributed Array Buffers (i.e. DistArray Buffers) instead.

A DistArray Buffer is a write-back buffer for each worker. DistArray Buffers share the same programming interface for random access as DistArrays. An application may create different DistArray Buffers to buffer writes for different DistArrays and may customize how the writes are applied by supplying a user-defined function. This function works on each individual pair of write and DistArray element and may additionally read and update elements of other DistArrays with the same matrix index. This allows applications to implement more sophisticated schemes to apply writes such as using adaptive gradient algorithms and deal with conflicting writes.

By default, the buffered writes are applied to corresponding DistArrays at the end of each partition of the iteration space. The driver program may optionally specify an upper bound in number of loop iterations on how long the writes may be delayed. In essence, using DistArray Buffers resembles update buffering that is commonly used in data-parallel training via parameter servers. Unlike typical data-parallel training where all dependencies suffer violation, Orion preserves the critical dependency while trading less critical dependencies for high parallelism.
3.3.5 Program Annotations

Application programmers may optionally provide additional information to Orion’s static analysis via annotations, which may be used to improve execution throughput. For example, the parallel for-loop may be annotated with a repeated argument, indicating the for-loop is expected to be executed many times. This enables Orion runtime to cache the for-loop execution context as well as runtime information collected from earlier executions to improve performance for later iterations.

3.3.6 Additional Features

Arbitrary Statements. Thanks to Julia’s JIT compilation, the driver program may execute any arbitrary Julia statement on all workers and collect their results. This enables the driver program to define functions, global variables or change the value of a global variable or even read the value of a workers’ local variable.

Accumulators. The driver program may obtain runtime information by querying the value of workers’ local variables, or read elements from a DistArray. To facilitate common use cases, Orion additionally provides accumulator variables. An accumulator variable creates a variable in each worker’s local memory, which can be written during any worker computation, such as DistArray transformations or parallel for-loops. The driver may query the aggregated value of different variable instances using a user-defined commutative and associative operator.

3.3.7 Putting Everything Together

Fig. 3.7 shows a machine learning program, Stochastic Gradient Descent Matrix Factorization (SGD MF), parallelized by Orion. The whole program consists of 87 lines of code while an implementation based on parameter server consists of 300 to 500 lines of code. Compared to a serial Julia implementation, the Orion program loads the training dataset into a DistArray using a custom parse function. The model parameters (W and H) are initialized as random matrices. The program contains two parallel for-loops. One loop implements the SGD algorithm and the other loop evaluates the model on the training dataset. The training error is collected using an accumulator variable. The training error can be used to implement more a sophisticated stopping criteria based on the learning progress, using Julia’s native dynamic control flow.

3.4 Static Parallelization And Code Generation

The @parallel_for construct is implemented as a Julia macro. The macro is expanded once at runtime when the for-loop statement is compiled. Orion requires that all DistArrays and DistArray Buffers that are accessed by the loop body are materialized before the loop, which allows our macro function to use runtime information of the related DistArrays during parallelization. The macro function takes as input the AST of the loop to be parallelized and an optional list of arguments. The macro applies static analysis and generates a sequence of statements that distribute the loop’s iterations to distributed workers.

Static analysis for loop parallelization takes the following steps:

1. Context Snapshot. Orion identifies all variables used in the loop body and classifies them into inherited and local variables (variables that are defined within the loop body). Orion generates statements to take a snapshot of inherited variables and broadcast their values to workers, before each execution of the loop.

2. Parallelization and Static Scheduling. Orion extracts all random accesses to DistArrays within the loop body and parallelizes the loop via dependence analysis, which determines
# Define local variables like step_size and functions like parse_line
Orion.@dist_array ratings = Orion.text_file(data_path, parse_line)
Orion.materialize(ratings)
dim_x, dim_y = size(ratings)
Orion.@dist_array W = Orion.randn(K, dim_x)
Orion.@dist_array W = Orion.map(W, map_init_param, map_values = true)
Orion.materialize(W)

# Create DistArray H
Orion.@accumulator err = Float32(0.0)
for iteration = 1:num_iterations
  Orion.@parallel_for for (key, rv) in ratings
    x_idx = key[1]
y_idx = key[2]
    W_row = @view W[:, x_idx]
    H_row = @view H[:, y_idx]
    # Compute W and H updates
    W[:, x_idx] .= W_row + W_updates
    H[:, y_idx] .= H_row + H_updates
  end
  Orion.@parallel_for for (key, rv) in ratings
    # Compute the predicted rating
    err += abs2(rv - pred)
  end
  err = Orion.get_aggregated_value(:err, :+)
  Orion.reset_accumulator(:err)
  println("iteration = ", iteration, " err = ", err)
end

Figure 3.7: Stochastic Gradient Descent Matrix Factorization Implemented On Orion
how the iteration space is partitioned and the dependence among partitions. According to the dependence among partitions, Orion fits the loop execution to a static schedule.

3. Repartition DistArrays for Random Access. Based on the DistArrays’ access pattern, Orion determines how each accessed DistArrays are partitioned to minimize remote accesses. When possible, a DistArray is partitioned among the Orion executors that executes the for loop computation (referred to as workers) and thus no remote accesses are needed during loop execution. Otherwise, the DistArray is partitioned and served by a set of executors referred to as servers which build global indices to provide efficient random access.

4. Code Generation. Orion generates a function that applies the loop body computation to a partition of the iteration space, which is executed by workers. If needed, Orion generates additional functions such as bulk prefetching to improve execution throughput. The generated functions are broadcasted to all executors involved in the loop execution. Orion also generates statements to repartition the iteration space and randomly accessed DistArrays. Finally Orion generates statements that command the Orion runtime master to execute the chosen schedule.

3.4.1 Parallelization

During parallelization, Orion partitions the iteration space and identifies dependence among the partitions, which is done using static dependence analysis.

**Dependence Vectors.** A lexicographically positive vector\(^3\) \(\vec{d}\) denotes a dependence vector of an n-loop nest if there exist two dependent iterations \(\vec{p}_1\) and \(\vec{p}_2\) such that \(\vec{p}_1 = \vec{p}_2 + \vec{d}\). Thus each dependence vector represents a set of edges and the number of dependence vectors is independent of the size of the iteration space. The elements of a dependence vector could be numbers, positive or negative infinity or infinity\(^4\). Dependence vectors are the basis on top of which dependence analysis is performed. Many previous works discussed how to compute dependence vectors [49, 59].

**Parallelization Strategy.** Orion computes dependence vectors\(^5\) to determine whether the loop is parallelizable and how it is parallelized. Let \(\mathcal{D}\) denote the set of dependence vectors of an n-dimensional iteration space. Orion determines the parallelization strategy of the for-loop with the following procedure.

1. 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 if \(p_i \neq p_i'\). Partitioning by dimension \(i\) ensures that any two iterations \(\vec{p}\) and \(\vec{p'}\) from two different partitions are independent. This is referred to as 1-dimensional (i.e. 1D) parallelization. Note that all such dimensions \(i\) that satify the above condition are candidate dimensions for partitioning the iteration space.

2. Else 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 if \(p_i \neq p_i'\) 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.

---

\(^3\)A vector \(\vec{d}\) is lexicographically positive if \(\exists i : d_i > 0\) and \(\forall j < i : d_j \geq 0\)

\(^4\)Given a dependence vector \(\vec{d} = (d_1, d_2, ..., d_n)\), \(d_i = \infty\) means \(\forall x \in \mathbb{R}, (d_1, d_2, ..., d_i-1, x, d_{i+1}, ..., d_n)\) is a dependence vector.

\(^5\)Currently, Orion computes dependence vector for subscript expressions that involve a single loop induction variable whose coefficient is 1, otherwise the corresponding dependence vector element is assigned infinity.
All such dimension pairs of $i$ and $j$ that satisfy the above condition are candidate dimensions for partitioning the iteration space.

3. Else if none of the dependence vectors contains a negative infinity or infinity component, then 2D parallelization with unimodular transformation can be applied. Parallelizing for-loops using unimodular transformations was introduced by [77]. The algorithm firstly skews the iteration space to transform the loop to a fully permutable loop nest. Then the wavefront transformation is applied to the fully permutable loop nest. Let $D'$ denote the set of dependence vectors after transformation, $\forall \vec{d} = (d_1, d_2, ..., d_n) \in D'$ : $d_1 > 0$. In other words, let $L_1, L_2, ... L_n$ denote the transformed iteration space, there's no dependence between iterations of the innermost loop nest $L_2, L_3, ... L_n$. 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 2 dimensional partitioning of the original iteration space.

In order to satisfy the ordering constraints of the loop iterations, the iteration space is range partitioned in the above cases. Note that each parallelization strategy may propose several candidate dimensions for partitioning the iteration space, Orion uses a simple heuristic to choose the partitioning dimension(s) that minimizes the number of DistArray elements that need to be communicated among Orion executors during loop execution.

3.4.2 Static Scheduling

1D parallelization can be easily scheduled by assigning different iteration space partitions to different workers. Fig. 3.8a shows a 2 dimensional iteration space and the dependence among iterations. This loop can be parallelized by partitioning the iteration space by dimension $j$. The workers execute their local partitions without synchronizing with each other until the end of the loop.

Under 2D parallelization and 2D parallelization with unimodular transformation, the loop nest collapses into a 2-dimensional loop nest, where the outer loop iterations are executed sequentially and inner loop iterations are executed in parallel. Thus the outer loop can be executed as a sequence of global time steps and within each time step, the inner loop is partitioned among workers. We refer to the outer dimension as the time dimension and inner dimension as the space dimension. We observe that in both cases, a partition depends only on two partitions from the previous time step. Thus by properly assigning partitions to workers, Orion avoids global synchronization barriers between global time steps and a worker depends on only one other worker. Fig. 3.8b shows a 2 dimensional iteration space with slightly more complicated dependence among iterations. When 2D parallelization is applied, each space partition is assigned to a worker and iterations of the same color can be executed in parallel.

**Relaxing the ordering constraints.** In a traditional compiler, dependence indicates the ordering in which dependent iterations should be executed, such as shown in Fig. 3.8b. Such ordering is necessary for many applications to produce correct results. With the ordering constraints, simultaneous execution of two iterations might not be possible even when they do not share any data. For example, in Fig. 3.8b, iteration $(1, 1)$ and $(4, 2)$ doesn't share any data, but $(4, 2)$ cannot be executed in parallel with $(1, 1)$. This is because $(4, 2)$ depends on iteration $(4, 1)$ which in turn depends on $(1, 1)$. Thus when a worker is executing iteration $(1, 1)$, all other workers remains idle. This parallel for-loop requires 7 sequential time steps to finish.

In many machine learning programs, loop iterations can be executed in any order even when there are dependences among them. Different orderings produce different but equally good
results. By relaxing the ordering constraints, Orion schedules workers to start from different
time steps instead of leaving some workers idle. This allows reducing the number of sequential
time steps by about 50%. As shown in Fig. 3.8c, the for-loop completes in 4 sequential time steps.
Note that this relaxation is applicable only when the loop is 2D parallelized.

### 3.4.3 Partitioning DistArrays for Random Access

Generally, all DistArrays that are accessed by a parallel for-loop can be stored and served by
a cluster of server processes via a key-value interface. However each random access would
potentially result in one remote access over the inter-machine network. The overhead of such
random accesses is significant even when workers cache DistArray values and buffer updates.

#### Locality

Under 1D and 2D parallelization, suppose the iteration space is partitioned by space
dimension $i$. Given an iteration $\vec{p} = (p_1, p_2, ..., p_n)$ if for all accesses to DistArray $A$ in the form of
$A[s_1, s_2, ..., s_m]$, there exists a dimension $j$ such that $s_j$ matches $p_i$ then $A$ can be range partitioned
among workers such that its elements are served locally throughout the loop execution.

#### Pipelining

Under 2D parallelization, suppose the time dimension is $i$. Given an iteration $\vec{p} = (p_1, p_2, ..., p_n)$ if for all accesses to DistArray $D$ in the form of $D[s_1, s_2, ..., s_m]$, there exists a dimension $j$ such that $s_j$ matches $p_i$ then $D$ can be range partitioned among workers such that its elements are served locally within one time step. As $p_i$’s value changes upon beginning a new
time step, the worker retrieves a different partition of $D$ from another worker. A careful sched-
ule ensures a worker always receives the new partition from one statically designated worker
called its predecessor. Thus the communication of $D$ forms a pipelined ring pattern. With dou-
ble buffering, the communication of $D$ may be overlapped with computation. Fig. 3.9 shows the
execution of a 2D parallelized unordered for-loop with DistArray $D$ circulated among workers.
The ring communication pattern was also described in STRADS [51], but Orion automates the
decision and implementation while STRADS relies on application programmers to determine
and implement this communication strategy.

#### Bulk Prefetching

When local partitions can’t be realized for a DistArray, or a DistArray Buffer
is used to buffer its updates, the DistArray is partitioned among server processes. In order to
minimize the overhead of random remote accesses, Orion Prefetches the read DistArray values
in bulk. In order to preserve dependence among iteration space partitions and avoid running
out of memory, Orion prefetches only the read values for each partition.

In order to accurately determine which values to prefetch, existing systems rely on programmers’ effort. For example, IterStore [29] requires the application program to implement and execute a virtual iteration at the beginning of the worker program. The virtual iteration is supposed to access parameter values as if it is the actual computation but the parameter server returns an arbitrary value so that the access pattern can be recorded. This approach is likely to face limitations when the program contains different loops that have vastly different access patterns. Li et. al. [53] exposes a communication (Push/Pull) interface, and relies on the application programmer to implement bulk prefetch, including determining which values to prefetch and cache management. Both approaches rely on programmers to be careful to separate indirect accesses from accesses that can be prefetched.

Orion introduces an algorithm to automatically generate a function that computes which DistArray values should be prefetched with indirect accesses excluded. The generated function takes a partition of the iteration space as input and produces a list of matrix indices to be prefetched for each accessed DistArray that is partitioned among the server processes.

The pseudocode for this algorithm is presented in Alg. 6. The algorithm takes as input the control flow graph of the loop body that’s already in Static-Single-Assignment (SSA) form as well as a symbol table that contains the information for all SSA variables. The algorithm firstly eliminates DistArray access subscripts that have a data or control flow dependence on DistArray reads. Then it starts from the rest of the access subscripts, recursively add statements and control flows that they depend on, similar to dead code elimination.

3.4.4 Dealing with Skewed Data Distribution

As the iteration space is often sparse with a skewed data distribution (for example when iterating over a skewed dataset), partitioning the iteration space into equal ranges results in an imbalanced work partition among workers. Orion DistArrays support a randomize operation that randomizes a DistArray along one or multiple dimensions to achieve a more balanced distribution of iterations. Further more, during parallelization, Orion computes a histogram along each partitioning dimension. The histogram approximates the data distribution along a dimension and is used to find a more balanced partitioning of the iteration space. We found that histogram
Algorithm 6: Code generation to compute DistArray indices for bulk prefetching.

**input**: The set of DistArrays to be prefetched $D$

**input**: A control flow graph $loop\_body$ of the parallel for-loop’s loop body

**input**: A symbol table $sym\_tab$ that contains the information for all SSA variables

**output**: A function which computes the set of DistArray indices to be prefetched given a partition of the iteration space

**function** MarkStmt($stmt$):

/* Marks the statement $stmt$ so it will be included in the generated code. If $stmt$ contains DistArray reads, $stmt$ is transformed so that computation that depends on the read values is eliminated. */

$syms\_deleted \leftarrow \text{Set}()$;

$new\_delete \leftarrow \text{true}$;

while $new\_delete$ do

    $new\_delete \leftarrow \text{false}$;

    for $sym \in sym\_tab$ do

        if $sym$ depends on a DistArray or DistArray Buffer access or a symbol $\in syms\_deleted$ then

            $syms\_deleted \leftarrow syms\_deleted \cup \{sym\}$;

            $new\_delete \leftarrow \text{true}$;

    end

end

$syms\_in\_use \leftarrow \text{Set}()$;

$new\_use \leftarrow \text{true}$;

while $new\_use$ do

    $new\_use \leftarrow \text{false}$;

    for each statement $stmt$ in $loop\_body$ do

        if $\neg\text{isempty}(stmt.\text{uses} \cap syms\_deleted)$ then

            continue;

        end

        if $stmt$ contains a read from a DistArray $\in D$ or $\neg\text{isempty}(stmt.\text{defs} \cap syms\_in\_use)$ then

            $syms\_in\_use \leftarrow syms\_in\_use \cap stmt.\text{defs}$;

            $new\_use \leftarrow \text{true}$;

            MarkStmt($stmt$);

        end

    end

end

/* Reconstruct a function from the marked statements. The function is executed with virtual DistArrays which only records the accessed indices without returning a value. */
Table 3.2: Comparing the lines of code needed to implement the same application.

Based partitioning can reduce the per-iteration runtime of Latent Dirichlet Allocation by 10% on a skewed dataset (PubMed) even after randomization.

### 3.5 Preliminary Evaluation

Orion is implemented in about 17,000 lines of C++ code and 6,300 lines of Julia code (mostly for static analysis and code generation). In this section, we present a preliminary evaluation of Orion that evaluates both ease of use and execution efficiency. For execution efficiency, we conduct experiments on a 42-node cluster where each machine contains two Intel E5-2698Bv3 Xeon CPUs and 64GiB of memory. Each CPU contains 16 cores and 32 hardware threads. The machines are connected with 40Gbps ethernet.

#### 3.5.1 Ease of Use

It is difficult to objectively measure programmer effort. We instead report the lines of code (LOC) needed to implement various applications on Orion and compare it with implementations on other open-source distributed learning systems in Table 3.2. While in most cases, most programmer effort is spent on algorithm design, parallelization and performance tuning rather than typing, a shorter program often indicates fewer details to worry about and thus lower programmer effort. The reported applications are provided in the system’s open-source package except for SGD matrix factorization on TensorFlow, which was implemented by the authors. The lines of code was counted by the Count Lines Of Code [2] tool, which excluded comments and empty lines.

**Parallelization.** Our benchmark consists of a logistic regression model and a matrix factorization model solved using Stochastic Gradient Descent (SGD) and a Latent Dirichlet Allocation model solved using Gibbs Sampling, including their variants using adaptive gradients when applicable. We implemented serial Julia programs for the above applications and parallelized them by applying Orion’s primitives. In general, the parallelization requires adding 10 to 20 lines of code to connect the driver program to Orion’s runtime master and a small number of lines of code to randomize the dataset, collect runtime results, etc. The major code changes in the driver program code include: 1) creating DistArrays instead of local matrices; 2) apply the `@parallel_for` macro; 3) execute necessary expression on workers, mostly for sharing function
definitions with workers.

Orion’s parallelization reduces the time taken to perform one data pass on the Netflix from 750 seconds to 11 seconds using 4 machines (64 physical cores with hyperthreading), achieving a $65 \times$ speedup, without harming per-iteration learning progress.

**Comparing with Other Systems.** Implementations of SGD matrix factorization and Latent Dirichlet Allocation on parameter server systems (Bösen and IterStore) are parallelized by manually partitioning the training data file and implementing a client program that executes on each distributed machine. The PowerGraph implementation requires a preprocessing step that converts the input data to a proper graph format. Partitioning and converting data formats require additional code that’s not included in the line count. In contrast, the Orion implementations included user-defined parsers to read and parse data from raw data files, which are included in the line count.

None of the implementations on parameter server systems, PowerGraph, and TensorFlow preserves data dependence between updates. As we show in Section 3.5.3, violating these dependencies slows the per-data-pass learning progress by $10 \times$ or more. C++ implementations generally require 3 to $6 \times$ more lines of code to implement the same program. The difference is the result of Orion’s automation, which require little manual system maintenance such as explicit virtual iteration or parameter subscription. It also shows the benefit of a scripting language.

Yahoo!LDA is a standalone distributed asynchronous LDA implementation. Its large LOC includes code for network communication, cache management, coordination, etc.

**Comparing with STRADS.** Besides Orion, STRADS is the only system whose applications use dependence-aware parallelization but it requires programmers’ manual parallelization. The STRADS LDA program consists of a scheduler program and a worker program that execute as separate processes. The scheduler program monitors workers’ progress and assigns tasks accordingly. The worker program implements the training algorithm and evaluation. The worker program also implements a ring model for pipelining communication of the model parameters. The manual dependence-aware parallelization requires more than 4500 lines of code which is $11.34 \times$ more than Orion’s automatic parallelization.

**Bulk Prefetching.** When training sparse logistic regression using stochastic gradient descent, each data sample reads a number of weight values depending on which features of this data sample are non-zero and update those weights accordingly. Thus which weights are accessed is unknown until the data sample is processed. Fig 3.10 shows how the training algorithm sequentially iterates over the non-zero features of a data sample and reads weight values ($weights[fid]$ where $weights$ is a DistArray) according to the feature ID. The sequence of DistArray accesses causes a sequence of inter-process communication, possibly over inter-machine networks. The KDD2010 (Algebra) [37] dataset contains 8407752 data samples and in average 36.3 non-zero features per data sample. When executed on a single machine with 16 worker processes and 16 server processes, each data pass takes 7682 seconds. Orion can automatically generate a function to compute the indices of the remote DistArray values that are accessed during a number of to-be-executed for-loop iterations and prefetch those values in bulk (see Section 3.4.3) with no additional effort by the application programmer. Bulk prefetching reduces the per-data-pass runtime to 9.2 seconds. Since the application program takes repetitive data passes over the same dataset that is unchanged, the DistArray indices for prefetching only need to be computed once and re-used in later data passes. This can be provided as a programmer hint to Orion by applying the repeated argument to @parallel_for, which further reduces the per-data-pass time to 6.3
for feature in features
    fid = feature[1]
    fval = feature[2]
    sum += weights[fid] * fval
end

Figure 3.10: Sparse logistic regression iterates over non-zero features.

<table>
<thead>
<tr>
<th>Application</th>
<th>Implementation</th>
<th>Avg. Time Per Iteration in seconds (overhead vs. serial)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latent Dirichlet Allocation (K = 100)</td>
<td>Serial Julia</td>
<td>72.92</td>
</tr>
<tr>
<td></td>
<td>Serial Julia w/ compact keys</td>
<td>82.23 (12.8%)</td>
</tr>
<tr>
<td></td>
<td>Orion Parallelized (1 thread)</td>
<td>91.85 (20.61%)</td>
</tr>
<tr>
<td>Latent Dirichlet Allocation (K = 1000)</td>
<td>Serial Julia</td>
<td>263.11</td>
</tr>
<tr>
<td></td>
<td>Serial Julia w/ compact keys</td>
<td>284.7 (8.2%)</td>
</tr>
<tr>
<td></td>
<td>Orion Parallelized (1 thread)</td>
<td>295.128 (10.85%)</td>
</tr>
</tbody>
</table>

Table 3.3: Orion Abstraction Overhead. Comparison between a serial Julia program and the program running on Orion with a single thread. Parallelization itself took 8.7 and 9.7 respectively which including computing the histogram along two dimensions of the iteration space.

3.5.2 Orion Abstraction Overhead

Compared to a Julia serial program, executing an Orion program involves scheduling, accessing DistArrays as opposed to native Julia Arrays, and conversion between custom and Orion’s data representation. In order to understand the overhead of Orion’s abstraction, we compared the per-iteration runtime of the serial LDA program and the Orion-parallelized program executed with a single thread. The results are presented in Table 3.3. The reported time is the per-iteration average from iteration 2 through iteration 8. The first iteration is not included to exclude parallelization and JIT compilation overhead. It should be noted that the relative abstraction overhead of Orion decreases as computation per iteration increases. In the LDA application, the dataset is represented as a three-dimensional matrix. Orion represents each matrix index as a compact 64-bit integer rather than 3 integers, which is converted on the fly when each iteration is executed. Each conversion involves several floating-point division, which is a slow operation. In order to understand the runtime overhead of this trade-off, we implemented a serial program that uses the same compact keys. We found that this conversion constitutes majority of the Orion overhead.

3.5.3 Orion vs. Manual Data Parallelism

We first evaluate the importance of dependence-aware scheduling and Orion’s execution efficiency by comparing Orion’s automatic parallelization with manual data parallelism using Bösen, on two machine learning applications, SGD matrix factorization and Latent Dirichlet Allocation (LDA) with various datasets. All experiments used 12 machines. The results are presented in Fig. 3.11, Fig. 3.12, Fig. 3.13, and Fig. 3.14. We observed that in all cases Orion parallelization achieves a faster per-iteration learning progress compared to manual data parallelism due to the dependence-preserving parallelization. Manual data parallelism on Bösen achieved a higher computation throughput (e.g. 2.48 vs. 6.04 seconds per iteration on SGD MF and 4.44 vs. 4.81 on LDA with NYTimes for Bösen and Orion respectively). Orion still converges faster in time in all cases except for LDA with PubMed data.
Figure 3.11: Comparing w/ Bösen Parameter Server on SGD Matrix Factorization
(a) Convergence Over Time
(b) Convergence Over Iterations

Figure 3.12: Comparing w/ Bösen Parameter Server on LDA (dataset: NYTimes)
(a) Convergence Over Time
(b) Convergence Over Iterations

Figure 3.13: Comparing w/ Bösen Parameter Server on LDA (dataset: PubMed)
(a) Convergence Over Time
(b) Convergence Over Iterations
3.5.4 Orion vs. Manual Data Parallelism w/ Managed Communication

As discussed in Chapter 2, managed communication improves the learning progress for data parallelism by communicating updates and updated model parameters more frequently to reduce error. In essence, it automates the common practice of manual mini-batch size tuning and prioritize communication for more sensitive parameters.

Fig. 3.11, Fig. 3.12, Fig. 3.13, and Fig. 3.14 also included comparison with managed communication on top of manual data parallelism. In the case of SGD matrix factorization, the Bösen implementation uses the adaptive revision algorithm [60] for auto-tuning the learning rate to fully take advantage of managed communication. Applying adaptive revision on top of the Orion implementation requires adding about 20 lines of code, while it took the Bösen implementation about 200 lines of additional code for implementing server-side UDFs. We assigned each Bösen node a bandwidth budget of 1600Mbps and 2560Mbps respectively for SGD matrix factorization and LDA.

In both applications, communication management substantially improves learning progress for data-parallel execution, closing the gap between manual data-parallelism with Orion’s dependence-aware parallelization. This results in a similar convergence rate in time between Orion and managed communication on Bösen in most cases. But for LDA with ClueWeb25M, managed communication converges about $2 \times$ slower in time compared to Orion as the frequent communication reduced Bösen computation throughput.

In order to minimize inconsistency, communication management communicates model updates and parameter values aggressively under the given bandwidth budget. Fig. 3.15 shows the bandwidth consumed by one node of Orion and Bösen with bandwidth management for both SGD matrix factorization and LDA. 4 and 8 machines are used respectively in the two experiments and Bösen is configured with a 800Mbps and 1280Mbps bandwidth budget per machine. Bösen blindly communicates updates and parameter values with no knowledge of when those values will be needed by the receiver. Values that are not immediately used could have been buffered to allow further coalescing, reducing the bandwidth consumption. On the other hand, Orion schedules computation according to dependence analysis and communicates values only when they are needed. As we can observe from Fig. 3.15, Orion consumed much less network bandwidth compared to Bösen.
Figure 3.15: Outbound bandwidth consumption for Orion and Bösen’s managed communication

Figure 3.16: Comparing w/ STRADS Scheduler on LDA (dataset: NYTimes)

Figure 3.17: Comparing w/ STRADS Scheduler on LDA (dataset: PubMed)
3.5.5 Orion vs. Manual Model Parallelism

With the STRADS framework [51], application programmers may manually parallelize a serial training program and implement a dependence-aware scheduler themselves. Implementing a scheduler requires non-trivial programmer effort. For example, it took 4512 lines of code on STRADS but 384 lines of code on Orion to implement the same LDA application. On the other hand, a custom scheduler allows the programmer to implement more application-specific optimizations. We compared the LDA program parallelized by Orion against the LDA implementation on STRADS using various datasets and present the results in Fig. 3.16, Fig. 3.17 and Fig. 3.18.

In all experiments, Orion’s automatic parallelization achieves similar per-iteration convergence progress as manual parallelization on STRADS. Orion achieved a $2.5 \times$ to $1.5 \times$ slower computation throughput.

In order to better understand Orion’s performance bottleneck, we present a breakdown of the LDA program’s execution time in Fig. 3.4. For a small dataset (NYTimes), serialization and deserialization of the model parameters circulated among workers (see Fig. 3.9) constitutes 27% of the total runtime. The relative overhead of serialization/deserialization reduces as the data size grows and the parameter size vs. computation ratio reduces. The STRADS implementation uses shared-memory multi-threading for each worker machine. Communicating model parameters between workers in the same machine requires only communicating a pointer to shared memory and thus eliminates the serialization/deserialization and memory copy overhead. This explains the larger performance gap ($2 \times$) on NYTimes. With the PubMed dataset, 44% of a worker’s runtime was spent on waiting for its predecessor to finish and send the corresponding model parameters. This is likely due to a highly skewed data distribution for the PubMed dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Tokens</th>
<th>#Parameters</th>
<th>Seconds Per Iteration</th>
<th>(De-)Serialization Time</th>
<th>Wait Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NYTimes</td>
<td>99.5M</td>
<td>102M</td>
<td>5.93</td>
<td>1.60</td>
<td>1.2</td>
</tr>
<tr>
<td>PubMed</td>
<td>737.8M</td>
<td>1.4B</td>
<td>40.00</td>
<td>3.23</td>
<td>17.5</td>
</tr>
<tr>
<td>ClueWeb25M</td>
<td>9.93B</td>
<td>2B</td>
<td>175.28</td>
<td>27.16</td>
<td>26.07</td>
</tr>
</tbody>
</table>

Table 3.4: Orion execution runtime breakdown. All experiments are executed using 12 machines (384 hyper-threading cores in total). Time is measured in seconds.
3.5.6 Orion vs. Dataflow Systems

TensorFlow [13] has become a widely popular framework for training deep neural networks. The dataflow execution model is suitable for deep learning where most of the computation is dense matrix multiplication. Fig. 3.19 shows that TensorFlow is inefficient for applications that perform frequent and sparse updates, such as solving matrix factorization using stochastic gradient descent. The experiments were executed on one single machine and the computation uses CPU. The TensorFlow implementation of SGD MF implements the update equation for a single mini-batch of training data as a DAG and repeated executes the DAG for all mini-batches in the training dataset. The DAG consists of parallelized operators which is the major source of parallelism, which depends on the mini-batch size.

Fig. 3.19b shows TensorFlow’s per-iteration runtime when an iteration is divided into 4 (TF_mb_4) or 128 (TF_mb_128) mini-batches, compared to Orion’s per-iteration runtime. Larger mini-batch causes TensorFlow to run out of memory. Firstly, reducing mini-batch sizes increases TensorFlow’s per-iteration runtime due to reduced parallelism. Secondly, even with the largest possible mini-batch size, TensorFlow spends 2.3× more time on one iteration that Orion does. This is because TensorFlow performs dense matrix multiplication, where some of the computation is redundant for computing gradients as the data matrix contains only sparse entries.

In order to retain high computation throughput, the TensorFlow program uses mini-batches containing millions of a data items (i.e. ratings), while the Orion program uses mini-batches that contain a single data item. Therefore, the Orion program updates model parameters much more frequently than TensorFlow, leading to much faster learning progress per iteration, and thus much faster convergence rate, as shown in Fig 3.19a.

3.6 Summary and Planned Case Studies

So far I have demonstrated Orion’s effectiveness on several representative machine learning applications previously used for benchmarking learning systems. I plan to further our evaluation with case studies on applications that have traditionally required specialized systems.

Word2Vec. Word2Vec [61, 62] is a set of popular models that produce word embeddings that can be used in NLP applications. Google has open sourced an implementation in C that provides parallelism on shared-memory multi-core [12]. TensorFlow also provides an implementation on shared memory. Parallelizing Word2Vec on Orion achieves a distributed Word2Vec program and allows us to compare Orion with two different implementations.
Gradient Boosted Trees. Gradient boosted trees is a widely popular model for classification and regression. It has served as an essential building block in the winning solutions of many machine learning competitions [24]. Unlike the above ML applications, training gradient boosted trees doesn’t iteratively refine a set of parameters. Instead, it finds a sequence of splits that partition the set of data samples into a number of clusters. To compute each split, the algorithm evaluates and compares a set of candidates. There exist different methods for selecting and evaluating candidates. Existing implementations of gradient boosted trees include specialized systems [24, 48] as well as implementations on Apache Spark [14]. The special computation pattern of gradient boosted trees makes it an interesting case study for Orion.

Neural Network Training. Data parallelism is the commonly used approach for distributed neural network training. Existing frameworks [13, 25] typically have a built-in parameter server implemented in C++. One challenge with existing systems is that implementing a new optimization algorithm (e.g. adaptive revision) requires implementing the core computation in C++, especially if the algorithm requires server-side computation. Orion allows all computation to be implemented in Julia, automatically distributes the computation among servers and relies on Julia’s JIT compilation to achieve native performance.
Chapter 4

Dynamic Scheduling for Dynamic Control Flow

4.1 Distributed Machine Learning w/ Dataflow Graph

Dataflow has become a popular programming model for machine learning because of the rich library support such as TensorFlow [13]. Applications of such systems describe their computation as a dataflow graph (e.g., Fig. 4.1a), where nodes are operations and edges are operations’ input and output. By requesting certain operation’s output, an application triggers the computation of operations that the output value depends on.

In order to implement a distributed TensorFlow application, a programmer assigns a devices (CPU, GPU, etc) to each operation in the dataflow graph, thus effectively partitions the graph among distributed computing devices. Given the graph partition, TensorFlow automatically inserts `Send` and `Recv` operations on the cut edges. Note that the graph partition and device assignment is done by the application program before the graph is executed and is fixed over the lifetime of the graph. Fig. 4.1b shows a dataflow graph partitioned among two devices.

Tensorflow supports dynamic control flow within a computation graph, via a `cond` and a `while_loop` operator. Dynamic control flow allows different parts of the graph to be executed or a part of the graph to be executed for a various number of times depending on runtime values. Moreover, TensorFlow allows independent operations from different loop iterations to be executed in parallel.

Optimal device placement of graph operations is tricky. The problem becomes much harder with dynamic control flow as which operations are executed and the operations’ input sizes are no longer known statically. A statically partitioned and assigned graph can thus cause skewness and unnecessary cross-device communication compared to a dataflow graph that’s dynamically scheduled.

4.2 Conditional Computation for Outrageously Large Neural Networks

To concretely demonstrate the deficiency of static device placement in the presence of dynamic control flow, we study conditional computation in neural networks as an example. It is understood that with sufficiently large training dataset, increasing the capacity (number of parameters) of neural networks improves model performance [71]. This has been observed in various application domains, including language translation [78], image classification [42] and speech recognition [16]. Training large neural networks demands high computing power and large memory. In a typical neural network model, the entire model is activated on each data item, so the computation complexity increases along with the number of parameters.

In order to resolve the computational challenge, a number of works have proposed using conditional computation such that each data item activates only some chunks of a large neural
(a) A simple dataflow graph

(b) Partitioned across two devices

Figure 4.1: Computation represented as dataflow graph

Figure 4.2: An MoE layer embedded in a recurrent neural network. The gating network selects two expert networks to process each sample. Their outputs are modulated by the outputs of the gating network (Figure courtesy [71]).

network [19, 20, 33, 71]. To the best of my knowledge, the state-of-the-art result was achieved by Shazeer et al. [71], which introduced a neural network layer called Sparsely-Gated Mixture-of-Experts (MoE). The MoE layer consists of a learned gating network and a number of (up to hundreds of thousands [71]) learned expert networks. The gating network can be any function that outputs a sparse n-dimensional vector, where n is the number of experts, and the expert network can be any arbitrary function and an MoE layer may contain different experts as long as they accept same-sized inputs and produce same-sized outputs. Moreover, the MoE layer could even be hierarchical where each expert itself is an MoE layer. The nonzero elements in the gating network’s output vector determine which expert networks are activated by the input data item as shown in Fig. 4.2. A neural network may adopt MoE as some of its layers and allowing the neural network to have much higher number of parameters without dramatically increasing the computation complexity. With the MoE layer, Shazeer et al. [71] successfully trained neural networks with 1000× more model parameters, achieving considerably better results than previous state-of-the-art in both language modeling and machine translation at lower computational complexity. The importance of exploring coarse-grained sparsity was also reinforced by a discussion in [34].
In Shazeer et al. [71], a single MoE layer contains several billions of parameters which can’t fit in a single GPU. In order to fit the model in GPU memory, Shazeer et al. [71] statically partitions the MoE layer: the gating network is replicated for data parallel training and each GPU is statically assigned a number of expert networks. As all network parameters reside in GPU memory throughout training, a large model requires a large number of GPUs to fit. Such an architecture faces a number of challenges regarding computation efficiency.

1. **Additional inter-machine communication.** If a model fits in a single GPU, the forward and backward propagation on a mini-batch of data samples can be executed without communication between GPUs. As an MoE layer is statically partitioned and assigned to distributed GPUs, the input of an expert network, which is the output of the previous neural network layer may come from a remote GPU. Similarly, the gradients of the expert network needs to be communicated back to the previous neural network layer during the back propagation. The time spent on this additional network communication increases as the mini-batch size and number of parameters increase.

2. **Mini-batch size limitation.** A large MoE layer requires a high number of GPUs to fit. It is well known that in SGD, the mini-batch size cannot be scaled infinitely due to diminishing return of learning progress and generalization gap [44, 50]. The limited mini-batch size may cause the expensive GPUs to be severely under utilized. As reported by Shazeer et al. [71], a large hierarchical MoE layer consisting of 131072 experts required 128 Tesla K40 GPUs to fit and the computational efficiency was only 0.30 TFLOPS/GPU while a theoretical maximum of 4.29 TFLOPS/GPU was claimed by NVIDIA. Newer GPUs such as Titan X (11.29 TFLOPS), GTX 1080 Ti (11.3 TFLOPS) and Tesla V100 (14 TFLOPS) have considerably higher computational throughput with small increase in memory capacity and the computational efficiency is expected to worsen following this trend. For cost effectiveness, it is desirable to allow the training algorithm to freely choose the mini-batch size and the number of GPUs for high computation efficiency as opposed to being restricted by the model’s memory footprint.

3. **Load Imbalance.** It was observed by Shazeer et al. [71] that the gating network tends to assign examples to the same few experts. This imbalance is self-enforcing as the favored experts are trained more rapidly and thus more likely selected by the gating network. As the expert networks are statically assigned to GPUs, this imbalance causes an imbalance in both computation and memory for GPU. A large input may even cause an expert network’s GPU to run out of memory. In order to achieve a balanced load among GPUs, Shazeer et al. [71] added two terms in the model’s loss function to encourage a balanced load assignment. However, balancing load is unlikely to be aligned with the true objective of the learning task, possibly harming model performance. Moreover, the two loss terms do not necessarily guarantee a balanced load or fitting in memory.

### 4.3 Preliminary Experimental Investigation

We perform a preliminary experimental study to investigate the deficiency of existing software systems for supporting conditional computation such as Mixture of Experts. Our experiments were conducted on a cluster of 4 machines, which each is equipped with a NVIDIA Titan X GPU and are connected by 40Gbps Ethernet. We used Tensorflow 1.8 [10] and neural network models implemented in Tensor2Tensor [75]. Tensor2Tensor provides neural network models for various machine learning tasks, such as translation and language modeling. Each model has multiple variants that differ in their neural network architecture hyper-parameters such as number of hid-
Our experiments show a number of deficiencies of TensorFlow for efficiently supporting Mixture of Experts.

1. A neural network scales poorly when partitioning and placing one of its layers among a set of distributed GPUs while replicating the other layers, compared to fully replicating a similar neural network among the same set of GPUs.

2. A heavy-loaded expert network may receive $10 \times$ more data samples compared to a light-loaded expert network in one mini-batch. When training one of the model variants, one GPU ran out of memory after 1231 updates due to the imbalanced load.

3. Although it was reported by Yuan Yu et al. [81] that TensorFlow implements memory swapping to offload parameter or activation values to host memory when GPU runs out of memory, this technique did take effect in our experiments.

### 4.3.1 Limited Scalability of MoE Networks

We investigate the weak scalability of MoE networks by comparing the training throughput in terms of number of data samples processed per second on 1 machine and 4 machines while fixing the per-machine/per-GPU mini-batch size. Our experiments used two small variants of TransformerMoe and AttentionLmMoe so the network can fit in the memory of a single GPU. As mentioned above, the MoE layer is statically partitioned and the worker GPUs are assigned with different expert networks. The other layers are replicated among workers with shared model parameters. The distributed workers are synchronized at each mini-batch, which is referred to as a *global step*. 

![Figure 4.3: Distributed training (n = 4)](image1)

![Figure 4.4: Single-machine training](image2)

Figure 4.3: Distributed training (n = 4)  
Figure 4.4: Single-machine training  

Figure 4.5: Model: TransformerMoe, HParams: TransformerMoe2K, Dataset: TranslateEn-deWmt32k. X-axis is the number of global steps.
Fig. 4.5 and Fig. 4.8 compares the throughput of a single machine and 4 distributed machines for training a TransformerMoe model and an AttentionLmMoe model respectively, where the number of global steps executed per second is reported. As 4 worker machines process $4 \times$ more data samples per global step, we observed a 50% throughput decrease in terms of number of data items processed per second when scaling from 1 to 4 machines for TransformerMoe, while a $1.55 \times$ increase was observed for AttentionLmMoe.

To the best of my knowledge, it is difficult if not impossible to measure the time spent on communicating activation values and gradients between the distributed experts and their connected layers in the network, using existing profiling tools with TensorFlow. In order to understand the overhead of employing statically distributed MoE layers, we report the throughput of two similar networks while do not employ Mixture of Experts and thus are fully replicated among workers. Fig. 4.11 and Fig. 4.14 compares the throughput of a single machine and 4 distributed machines for training a 4-layer Transformer model and an AttentionLmMoe model with the MoE layer removed. We observed a $1.3 \times$ and $1.7 \times$ throughput increase respectively in terms of number of data items processed per second when scaling from 1 to 4 worker machines. We found that the networks without MoE layers achieve higher speedup from 4 machines compared to their counterparts that employ MoE layers.
4.3.2 Load Distribution among Expert Networks

We examine the load distribution among expert networks at different global time steps for three variants of the AttentionLmMoE model, the results are shown in Fig. 4.15, Fig. 4.16, and Fig. 4.17. In each figure, the horizontal axis depicts the number of global steps and the vertical axis shows workers’ load distribution in terms of number of data items. At each global step, the highest data point shows the number of data samples assigned to the heaviest expert network and the lowest data point shows the load of the lightest expert network. The other data points are percentiles at various percentage. Note that AttentionLmMoEImbalnce (Fig. 4.16) is an AttentionLmMoE model with the load-balancing loss removed. In all cases, we observed that in some global steps, the heaviest expert network received 10× more data samples than the lightest expert network.

4.3.3 Throughput Drop

We additionally report the training throughput of AttentionLmMoEImbalnce in Fig. 4.18, we observed that the training throughput dropped by 80% after approximately 6000 global steps. While the cause to this problem is unclear, I suspect it has due to the skewed load distribution.
Figure 4.16: Model: AttentionLmMoe, HParams: AttentionLmMoeImbalance, Dataset: LanguageModelLm1b32k

Figure 4.17: Model: AttentionLmMoe, HParams: AttentionLmMoeTranslation, Dataset: LanguageModelLm1b32k
and imbalanced load that gets worse as the model stabilizes.

4.4 Dynamic Scheduling for Dynamic Control Flow in Dataflow Graphs

Motivated by the problems discussed above, I propose to study dynamic scheduling for dynamic control flow in dataflow graphs, particularly targeting conditional computation, in order to reduce memory footprint and improve computation throughput.

4.4.1 Incorporating Static and Dynamic Device Placement

The goal of dynamic scheduling is to balance the computational load among GPUs under the GPUs’ memory constraints. A stretch goal of dynamic scheduling is to minimize cross-device communication. Given a computation graph, determining its optimal partition and device placement is challenging. Dynamic scheduling also requires the scheduling decisions to be made under low latency, which is more difficult with a larger graph. Moreover, it is desirable to be fully compatible with existing TensorFlow API. In order to incorporate both static and dynamic scheduling, we introduce a concept called virtual device.

I propose an infinite number of virtual devices, including one special anonymous virtual device. Scheduling maps each virtual GPU or CPU device to a physical GPU or CPU. The anonymous virtual device can be additionally partitioned to multiple virtual devices or replicated as determined by the dynamic scheduler. An application program may assign a virtual device to the operators that they desire to be scheduled dynamically and assign the rest of operators to physical devices as usual. Application programmers may thus enforce additional scheduling constraints. For example, an application may enforce some operators to be assigned to the same physical device by assigning them with the same virtual device.

4.4.2 Operator Scheduling Strategies

Finding the optimal device placement for operators assigned with the anonymous virtual device is the most challenging as the degree of freedom is the largest. Ideally, the entire computation graph may be assigned with the anonymous virtual device. An additional scheduling strategy for operators of the anonymous virtual device is replication.

An operator that receives an exceedingly large input may be replicated over $N$ devices, with
each replica receiving $1/N$ of the input. Consider the example of Mixture of Experts. The expert networks can be placed onto devices dynamically according to the gating network’s output. Moreover, given spare GPU memory, some popular expert networks can be replicated on multiple GPUs to reduce the network communication between the expert network and its connected layers.

4.4.3 Operator Profiling

Scheduling requires the operators’ runtime information including execution time, memory footprint, and output size under different input sizes. I plan to explore both strategies to collect the operators’ profile online and offline.

4.4.4 Memory Swapping and Dynamic Batching

Memory swapping has been proposed as a technique to reduce GPU memory footprint by leveraging the layer-wise computation pattern to offload parameters and activation values of some layers of the neural network to host (CPU) memory [30, 81]. Dynamic batching [55] reduces the number of GPU calls and improves GPU utilization by batching several calls to the same operator into a single call with the combined input of the individual calls. These techniques create additional opportunities in scheduling to reduce memory footprint and improve execution efficiency.
Appendix A

Orion Application Program Examples

A.1 Stochastic Gradient Descent Matrix Factorization

```julia
include("/path/to/orion/src/julia/orion.jl")
Orion.set_lib_path("/path/to/orion/lib/liborion_driver.so")

const master_ip = "10.117.1.17"
const master_port = 10000
const comm_buff_capacity = 1024
const num_executors = 64
const num_servers = 1

Orion.glog_init()
Orion.init(master_ip, master_port, comm_buff_capacity, num_executors, num_servers)

const data_path = "file:///path/to/data.csv"
const K = 1000
const num_iterations = 256
const step_size = Float32(0.01)

Orion.@accumulator err = 0
Orion.@accumulator line_cnt = 0

Orion.@share function parse_line(line::AbstractString)
  global line_cnt
  line_cnt += 1
  tokens = split(line, ',')
  @assert length(tokens) == 3
  key_tuple = (parse(Int64, String(tokens[1])),
               parse(Int64, String(tokens[2])))
  value = parse(Float32, String(tokens[3]))
  return (key_tuple, value)
end

Orion.@share function map_init_param(value::Float32)::Float32
  return value / 10
end
```
Orion.@dist_array ratings = Orion.text_file(data_path, parse_line)
Orion.materialize(ratings)
dim_x, dim_y = size(ratings)
println((dim_x, dim_y))
line_cnt = Orion.get_aggregated_value(:line_cnt, :+);
println("number of lines read = ", line_cnt)
Orion.@dist_array W = Orion.randn(K, dim_x)
Orion.@dist_array W = Orion.map(W, map_init_param, map_values = true)
Orion.materialize(W)
Orion.@dist_array H = Orion.randn(K, dim_y)
Orion.@dist_array H = Orion.map(H, map_init_param, map_values = true)
Orion.materialize(H)
error_vec = Vector{Float64}()
time_vec = Vector{Float64}()
start_time = now()
W_grad = zeros(K)
H_grad = zeros(K)
@time for iteration = 1:num_iterations
    x_idx = rating[1][1]
y_idx = rating[1][2]
rv = rating[2]

    W_row = @view W[:, x_idx]
    H_row = @view H[:, y_idx]
    pred = dot(W_row, H_row)
    diff = rv - pred
    W_grad .= -2 * diff .* H_row
    H_grad .= -2 * diff .* W_row
    W[:, x_idx] .= W_row .- step_size .* W_grad
    H[:, y_idx] .= H_row .- step_size .* H_grad
end
@time if iteration % 4 == 1 ||
    iteration == num_iterations
    println("evaluate model")
    Orion.@parallel_for for rating in ratings
    x_idx = rating[1][1]
y_idx = rating[1][2]
rv = rating[2]
W_row = @view W[:, x_idx]
H_row = @view H[:, y_idx]
pred = dot(W_row, H_row)
err += (rv - pred) ^ 2
end

err = Orion.get_aggregated_value(:err, :+)
curr_time = now()
elapsed = Int(Dates.value(curr_time - start_time)) / 1000
println("iteration = ", iteration, " elapsed = ", elapsed, " err = ", err)
Orion.reset_accumulator(:err)
push!(error_vec, err)
A.2 Sparse Logistic Regression

```julia
include("/path/to/orion/src/julia/orion.jl")
Orion.set_lib_path("/path/to/orion/lib/liborion_driver.so")

const master_ip = "127.0.0.1"
const master_port = 10000
const comm_buff_capacity = 1024
const num_executors = 16
const num_servers = 16
Orion.glog_init()
Orion.init(master_ip, master_port, comm_buff_capacity, num_executors, num_servers)

const data_path = "file:///proj/BigLearning/jinlianw/data/kdda"
const num_iterations = 64
const step_size = Float32(0.00001)
const num_features = 20216830

Orion.@accumulator err = Float32(0)
Orion.@accumulator loss = Float32(0)
Orion.@accumulator line_cnt = 0

Orion.@share function parse_line(index::Int64, line::AbstractString)
  global line_cnt += 1
  tokens = split(strip(line), ' ')
  label = parse(Int64, tokens[1])
  if label == -1
    label = 0
  end
  i = 1
  feature_vec = Vector{Tuple{Int64, Float32}}(length(tokens) - 1)
  for token in tokens[2:end]
    feature = split(token, ":")
    feature_id = parse(Int64, feature[1])
    @assert feature_id >= 1
    feature_val = parse(Float32, feature[2])
    feature_vec[i] = (feature_id, feature_val)
    i += 1
  end
  return ((index,), (label, feature_vec))
end
```

58
Orion.@dist_array samples_mat = Orion.text_file(data_path,
    parse_line,
    is_dense = true,
    with_line_number = true,
    new_keys = true,
    num_dims = 1)
Orion.materialize(samples_mat)

line_cnt = Orion.get_aggregated_value(:line_cnt, :+)
println("number of lines read = ", line_cnt)

Orion.@dist_array weights = Orion.rand(num_features)
Orion.materialize(weights)

return Float32(1.0) ./ (Float32(1.0) .+ exp(-z))
end

if abs(x) < Float32(1e-15)
    x = Float32(1e-15)
end
return log(x)
end

Orion.@dist_array weights_buf = Orion.create_sparse_dist_array_buffer((weights.dims ...
), Float32(0.0))
Orion.materialize(weights_buf)

Orion.@share function apply_buffered_update(key, weight, update)
    return weight + update
end

Orion.set_write_buffer(weights_buf, weights, apply_buffered_update)

error_vec = Vector{Float32}()
loss_vec = Vector{Float32}()
time_vec = Vector{Float64}()
start_time = now()

for iteration = 1:num_iterations
    sum = 0.0
    label = sample[2][1]
    features = sample[2][2]
    for feature in features
        fid = feature[1]
        fval = feature[2]
        sum += weights[fid] * fval
    end
    diff = sigmoid(sum) - label
    for feature in features
        fid = feature[1]
        fval = feature[2]
        weights_buf[fid] -= step_size * fval * diff
    end
end
if iteration % 1 == 0 ||
    iteration == num_iterations
    Orion.@parallel_for for sample in samples_mat
        sum = 0.0
        label = sample[2][1]
        features = sample[2][2]
        for feature in features
            fid = feature[1]
            fval = feature[2]
            sum += weights[fid] * fval
        end
        if label == 1
            loss += -safe_log(sigmoid(sum))
        else
            loss += -safe_log(1 - sigmoid(sum))
        end
        diff = sigmoid(sum) - label
        err += abs2(diff)
    end
err = Orion.get_aggregated_value(:err, :+)
loss = Orion.get_aggregated_value(:loss, :+)
curr_time = now()
elapsed = Int(Dates.value(curr_time - start_time)) / 1000
println("iteration = ", iteration, " elapsed = ", elapsed, " err = ", err, " loss = ", loss)
push!(error_vec, err)
push!(loss_vec, loss)
push!(time_vec, elapsed)
Orion.reset_accumulator(:err)
Orion.reset_accumulator(:loss)
end
end
println(error_vec)
println(loss_vec)
println(time_vec)
Orion.stop()
exit()
Bibliography


[9] Pyston 0.6.1 released, and future plans. https://blog.pyston.org/2017/01/31/pyston-0-6-1-released-and-future-plans/. 3.3.1


[38] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. volume 38, June 2013. 2.4.1


[45] Mike Innes, Stefan Karpinski, Viral Shah, David Barber, Pontus Stenetorp, Tim Besard, James Bradbury, Valentin Churavy, Simon Danisch, Alan Edelman, Jon Malmaud, Jarrett Revels, and Deniz Yuret. On machine learning and programming languages. SysML ’15, Palo Alto, California, Feb 2018. 3.3.1

[46] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Dis-


[60] H. Brendan McMahan and Matthew Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. Advances in Neural Information Processing Systems (NIPS), 2014. 2.4.2, 3.5.4


[70] A. W. Savich and M. Moussa. Resource efficient arithmetic effects on rbm neural network solution quality using mnist. In *2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2011)(RECONFIG)*, volume 00, pages 35–40, 11 2011. doi: 10.1109/ReConFig.2011.79. URL doi.ieeecomputersociety.org/10.1109/ReConFig.2011.79. 2.3.1


[78] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Mac-


