Improving ML applications in shared computing environments

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Department of Engineering

Aaron Harlap

B.S., Electrical & Computer Engineering, Northeastern University
M.S., Electrical & Computer Engineering, Carnegie Mellon University

Thesis Committee:
Gregory R. Ganger, Chair
Phillip B. Gibbons
Ameet Talwalkar
Amar Phanishayee

Carnegie Mellon University
Pittsburgh, PA

May 2019

Keywords: Distributed Machine Learning, Cloud Computing, Stragglers, Elasticity, DNNs

For my parents.

Acknowledgments

Throughout the last five years I have often thought about what I would write here when the time came, and even after five years I am not sure I have the right words to describe how much the people I have met along the way have helped, and how much they have meant to me.

It all starts with my advisors, Greg Ganger and Phil Gibbons. This whole dissertation could be about how good they are. I have often told prospective students asking about working with Greg, that getting him to be my advisor is the best decision I have made. I have never worked with a more even-keeled, helpful, supportive person than Greg. He has helped me grow as both a researcher and a person. Teaching me everything from how to go about the research process, to proper grammar (although the latter is still definitely a work in progress). Even though it was was sometimes painful during the process, I will always look back fondly on the many sleepless nights we spent preparing paper submissions. And I hope that one day Greg's ultimate dream will come true and Michigan football will finally be able to get a win or two against the Ohio State.

At the end of my first year I was also lucky enough get Phil as my advisor. Phil has this amazing ability to provide feedback in the most constructive way possible, always clearly explaining why something needs to be changed or done differently. I still remember when Phil helped me rewrite the evaluation section of my first paper submission. I have found that I learn best from examples, and there is no better example to learn from than Phil. I consider myself extremely lucky to have had the opportunity to learn from Greg and Phil.

Another person who has been very helpful to me is Amar Phanishayee from Microsoft Research. He hired me as a summer intern a couple years ago and I have been collaborating with him ever since. Amar gave me the opportunity to work on a cutting edge project, and has supported me in so many different ways, from providing valuable advice to giving me thousands of dollars of computing resources, making it possible for us to do really awesome work together. Getting to work with Amar has been one of the highlights of grad school for me and I am very happy to have him on my thesis committee.

Henggang Cui who graduated a couple years before me was an invaluable resource for me. As a younger student I used him as a role model for how to be a successful PhD student. He never minded me asking him hundreds of questions daily, always patiently explaining things to me, and showing me how to succeed in grad school. I also got to work alongside a number of other of Greg's students. From my random chats about all kinds of things with Rajat Kateja, going through the graduation process at the same time as Jun Woo Park, talking latest NBA rumors with Andrew Chung, getting to know Saurabh Kadekodi at ATC, collaborating with Alexey Tumanov, Angela Jiang having the neatest cube in our area by a mile, and watching Daniel Wong get started on his studies, I have enjoyed the time I got to spend around all of them.

There have been numerous people within the Parallel Data Lab (PDL) group that have made it infinitely easier for me to be able to just focus on my work. Karen Lindenfelser is the PDL mom, don't think I could even count high enough to keep track of the number of times I have asked her for help with all kinds of things. From bringing back pages of expense receipts in Serbian from EuroSys, to her making sure I made it through the last PDL retreat when my shoulder wasn't doing that well, she always helped. And there are many others as well. Bill Courtright who was always willing to share some of his wisdom and provide advice. Joan Digney who made awesome posters and never minded how late I sent her my drafts. Jason Boles, Chad Dougherty, and Mitch Franzos who spent countless hours helping me set up my experiments and were always willing to answer all kinds of questions. I also want to thank Ameet Talwalkar for joining my thesis committee and providing invaluable feedback, and for giving me my first job after graduation.

The CIC 2nd floor has been a great working environment, and definitely not because of the building itself. When I came to Pittsburgh five years ago, I knew very few people here and now I feel like I have made friends for a life. Janos Szurdi, Josh Tan, Mahmood Sharif, Jinliang Wei, Orsi Kovacs, Sruti Bhagavatula, Dan Smullen, and Aymeric Fromherz made the last five years feel like just a few weeks.

I also want to thank all the awesome people of PDL and my collaborators for the insightful technical discussions and the random chats over the years including Kevin Hsieh, Jin Kyu Kim, Deepak Narayanan, Majd Sakr, Garth Gibson, George Amvrosiadis, Jack Kosaian, Michael Kuchnik, Dana Van Aken, Tian Li, Timothy Zhu, Christopher Canel, Lei Cao, Andy Pavlo, Vivek Seshadri, Matei Zaharia, Nandita Vijaykumar, David Andersen, Gauri Joshi, Aurick Qiao, Lin Ma, Abutalib Aghayev, Alex Glikson, Chris Fallin, Akira Kuroda, Soo-Jin Moon, Charlene Zang, Giulio Zhou, Wei Dai, and Qing Zheng.

I also want to thank the members and companies of the PDL Consortium including Alibaba, Amazon, Datrium, Dell EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, IBM Research, Intel Corporation, Micron, Microsoft Research, NetApp, Oracle Corporation, Samsung, Seagate Technology, Two Sigma, Veritas, and Western Digital for their interest, insights, feedback, and support. This research is also supported in part by Intel as part of the Intel Science and Technology Center for Cloud Computing, NSF under awards CNS-1042537, 1409723, CCF-1533858, CNS1042543 (PRObE [25]).

I would like to thank Tom Brady, Bill Belichick, Malcolm Butler, Donta Hightower, and all the New England Patriots. Them winning three super bowls in the past five years made my time in Pittsburgh that much more enjoyable.

I want to thank Greg Ganger, Phil Gibbons, Ameet Talwalkar, and Amar Phanishayee for being on my thesis committee and supporting me throughout the entire process.

Finally I would like to thank my family, without their support none of this would have been possible.

Abstract

Machine learning (ML) has become a powerful building block for modern services, scientific endeavors and enterprise processes. The expensive computations required for training ML models often makes it desirable to run them in a distributed manner in shared computing environments (e.g., Amazon EC2, Microsoft Azure, in-house shared clusters). Shared computing environments introduce a number of challenges, including uncorrelated performance jitter, heterogeneous resources, transient resources and limited bandwidth. This dissertation demonstrates that, by structuring software frameworks and work distribution to exploit transient resources and address performance jitter and communication bandwidth limitations, we can improve the efficiency of training machine learning models.

We support this assertion with three case study systems: FlexRR, Proteus, and PipeDream. FlexRR is a distributed machine learning training system that combines a flexible synchronization model with dynamic peer-to-peer reassignment of work among workers to address stragglers caused by performance jitter. FlexRR observes near ideal run-time, mitigating the adverse effects of stragglers observed in shared computing environments. Proteus is an agile elastic machine learning training system that uses tiers of reliability and intelligent resource management to efficiently utilize transient compute resources. Evaluations on AWS EC2 show that Proteus reduces cost by 85% relative to non-transient pricing, and by 43% relative to previous approaches, while simultaneously reducing runtimes by up to 37%. PipeDream is a distributed training system for deep neural networks (DNNs) that partitions ranges of DNN layers among machines and aggressively pipelines computation and communication. By reducing the amount of communication, and overlapping communication and computation, PipeDream provides a 5x or more improvement in “time to accuracy” for training large DNN models.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Statistical machine learning (ML) has become a powerful building block for modern services, scientific endeavors and enterprise processes. Modern day machine learning is made up of different mathematical models, depending on the problem being solved, that are able to describe observed input data. Such mathematical models are able to predict outcomes for new data items based on selected characteristics (e.g., language translation, image classification), expose relationships among data items (e.g., or grouping documents into topics), and so on. In order to fit the observed data, these models contain model parameter values that are adjusted throughout the training process.

In this dissertation, we focus on the major category of ML approaches that employ iterative algorithms to determine the model parameters that best fit a given set of input data. Included in this category are convolutional neural networks, recurrent neural networks, matrix factorization, and many others. Iterative algorithms, such as stochastic gradient descent and Adam [62], make initial guesses at the appropriate model parameter values [76] and then iterate over the training data, adjusting the model parameters.

The expensive computations required for training ML applications often makes it desirable to run them in a distributed manner in shared computing environments (e.g., Amazon EC2, Microsoft Azure, in-house shared clusters). Distributed training of ML applications commonly requires the resources involved to maintain the model parameter values (solution state), evenly distribute work, synchronize progress and communicate amongst each other in order for the ML application to function effectively [49, 28, 103].

Shared computing environments introduce a number of challenges that need to be solved in order for the training process to execute efficiently. These challenges include: uncorrelated performance jitter; heterogeneous resources; transient resources; and limited communication bandwidth. In our work, we focus on improving the efficiency, reducing cost and reducing runtime of training ML applications in shared computing environments by addressing these challenges.

1.1 Thesis statement

This dissertation describes our work in addressing the challenges that arise in deploying machine learning systems in shared computing environments. In particular, we make the following thesis statement:

Improvements of 5x or more can be achieved for training ML models in shared computing environments by structuring software frameworks and work distribution to exploit transient resources and to address performance jitter and communication bandwidth limitations.

To support this thesis, we will describe three case studies of structuring software frameworks and work distribution for training ML applications in shared computing environments.

- **Addressing the straggler problem in iterative convergent ML (Chapter 3).** The frequent (e.g., per iteration) barriers used in traditional BSP-based distributed ML implementations cause every transient slowdown of any worker thread to delay all others. When training ML applications in shared computing environments, performance jitter from a variety of causes is commonplace, creating stragglers that adversely affects the training progress. We designed a system, FlexRR, that combines a more flexible synchronization model with dynamic peer-to-peer re-assignment of work among workers to address straggler threads. Experiments with real straggler behavior observed on Amazon EC2 and Microsoft Azure, as well as injected straggler behavior stress tests, confirm the significance of the problem and the effectiveness of FlexRR's solution. Using FlexRR, we consistently observe near-ideal run-times (relative to no performance jitter) across all real and injected straggler behaviors tested.
- **Agile ML elasticity through tiered reliability in dynamic resource markets (Chapter 4).** Many shared computing clusters allow users to utilize excess idle resources at lower cost or priority, with the proviso that some or all may be taken away at any time. However, exploiting such dynamic resource availability, and the often fluctuating markets for them, requires agile elasticity and effective acquisition strategies. Our system, Proteus, aggressively exploits such transient revocable resources to do machine learning cheaper and/or faster. Its parameter server framework, AgileML, efficiently adapts to bulk additions and revocations of transient machines, through a novel 3-stage active-backup approach, with minimal use of more costly non-transient resources. Its BidBrain component adaptively allocates resources from multiple Amazon EC2 spot markets to minimize average cost per work as transient resource availability and cost change over time. Our evaluations show that Proteus reduces cost by 85% relative to non-transient pricing, and by 43% relative to previous approaches, while simultaneously reducing runtimes by up to 37%.
- **Pipeline parallelism for DNN training (Chapter 5).** DNN training is extremely computationally expensive, necessitating the need for efficient multi-accelerator parallelization. Current approaches to parallelizing training primarily use intra-batch parallelization, where a single iteration of training is split over the available workers, but suffer from diminishing returns at higher worker counts. Our system, PipeDream, adds *inter-batch pipelining* to intra-batch parallelism in order to further improve par-

allel training throughput, helping to better overlap computation with communication and reduce the amount of communication when possible. PipeDream versions model parameters for backward pass correctness, schedules forward and backward passes of different minibatches to keep workers well utilized, and systematically partitions DNN layers among workers to balance work and minimize communication. Extensive experimentation with a range of DNN tasks, models, and hardware configurations shows that PipeDream reaches target accuracy up to 4.78× faster than data-parallel training.

1.2 Contributions

This dissertation makes the following key contributions.

FlexRR:

- It describes FlexRR, a new approach to straggler mitigation without the correctness problems of redundant task execution, for iterative convergent ML on efficient ML frameworks.
- It describes the concept of *helper groups*, limiting the amount of communication and data preloading required, allowing FlexRR to scale efficiently to large number of workers.
- It demonstrates that FlexRR successfully addresses stragglers for three real ML tasks in both private and publicly shared computing environments.

Proteus:

- It describes Proteus, the first parameter server ML framework designed to elastically scale with bulk additions and revocations of computing resources.
- It describes an adaptive architecture and algorithm for exploiting multiple tiers of machine reliability.
- It describes a new resource manager that aggressively exploits the EC2 spot market properties to achieve major cost savings.
- It presents results from experiments and analysis showing that aggressive multi-tier exploitation of transient computing resources is both possible and beneficial.

PipeDream:

- It introduces pipeline-parallel training, which combines efficient model-parallel and data-parallel training for DNNs.
- It identifies the key challenges in designing a pipeline-parallel training system, and details how PipeDream addresses them.
- It presents results that experimentally demonstrate that PipeDream enables efficient inter-batch parallel DNN training, even in circumstances where communication overheads cripple data-parallel training.

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 motivates our works with more background on shared computing environments, consistency models, the parameter server architecture, and some popular machine learning tasks. Chapter 3 describes FlexRR [49], our parameter server system that mitigates the effects of performance variation (stragglers) on iterative ML tasks. Chapter 4 describes Proteus [51], our parameter server design specialized for transient resources commonly found in shared computing environments. Chapter 5 describes PipeDream [50], our system for parallelized training of DNN models that are inefficient to train using data parallel approaches. Chapter 6 concludes the dissertation and discusses future research directions.

Chapter 2

Background: Data-Parallel ML and Parameter Servers

This section describes some additional background of our work, including shared computing environments, the parameter server architecture, consistency models, and example ML tasks.

2.1 Shared Computing Environments

Shared computing environments, both public and private, where multiple users time-share computing resources, have become popular for training machine learning models. This section provides background on two characteristics of modern shared computing environments that we explore in our work: transient resources and performance variation.

2.1.1 Transient Resources

Today's cluster infrastructures are increasingly dynamic, and working with transient resources on a regular basis is common. Resources may be available as temporarily-unused nodes on a revocable basis at a discount (in public pay-as-you-go clouds) or for lower-priority best effort workloads (in multi-use corporate clusters). For both public clouds and mixed-purpose corporate clusters, lower intensity periods for business critical workloads create an opportunity for extra machines to be made available to other workloads. However, those machines may need to be taken back if business-critical workload intensity increases.

Amazon AWS EC2 [1] is a public cloud that allows customers to purchase time on virtualized machine resources. The traditional EC2 model, referred to as "on demand" because machines can be requested and released by customers at any time (though billing is based on an hourly granularity), involves paying a pre-determined fixed hourly rate to have guaranteed access to rented machine resources. Amazon also has a so-called "spot market" for machines, where machines are often available at a steep discount (e.g., 70–80% lower price) with the proviso that they can be taken back at any time. Therefore, a customer who can exploit transient machines for their work can potentially save money and/or time.

2.1.2 Performance Variation

Another characteristic of shared computing environments that we address in our works is performance variation. Performance variation exists in both a persistent and transient nature. Most shared cloud providers offer a variety of hardware to users leading to persistent performance variation due to hardware heterogeneity [80, 95, 63, 1]. Examples of transient performance variation include hardware failures [14], garbage collection in high-level languages, various OS effects [17, 78] and resource contention. When using private dedicated hardware, it is possible to eliminate many of these performance variations [40, 78, 77], but in shared computing environments (especially public clouds) this is difficult and/or impossible.

2.2 Data Parallel ML

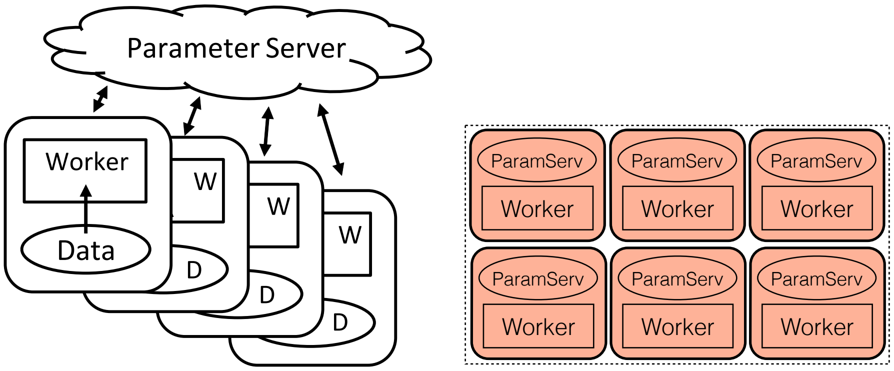
This section provides background on data parallel machine learning and the consistency models that are used to synchronize worker progress. In data-parallel machine learning training, inputs are partitioned across workers. Each worker maintains a local copy of the model weights and trains on its own partition of inputs while periodically synchronizing weights with other workers.

2.2.1 Parameter Server Architecture

The most efficient modern frameworks for data-parallel ML use a *parameter server* architecture, which allows programmers to easily build scalable ML algorithms while benefiting from such specializations [67, 54, 24, 49]. As a result, open source ML model training frameworks like TensorFlow [8], MxNet [22], Petuum [105] and many proprietary systems use variants of this architecture.

Figure 2.1 illustrates a simple parameter server system. Commonly, the only state shared among worker threads is the current parameter values, which is kept in a specialized key-value store called the *parameter server*. Worker threads process their assigned training data and use simple `read-param` and `update-param` methods to check and apply deltas to parameter values. The value type is usually application-defined, but must be serializable and have a commutative and associative aggregation function so that updates from different worker threads can be applied in any order.

To reduce cross-machine traffic, parameter server implementations include a worker-side library that caches parameter values and buffers updates. While logically a single separate server, the parameter server is usually sharded across the same machines as worker threads, enabling it to scale with the computation power and aggregate memory and bandwidth used for training. Threads associated with the worker-side cache communicate with the appropriate server shards for each given value. Updates are write-back cached and sent (asynchronously) to the appropriate parameter server shards each iteration.



(a) Logical Architecture

(b) Common Configuration

Figure 2.1: Traditional Parameter Server Architecture. The left figure illustrates the logical architecture, and the right figure illustrates that the parameter server is usually sharded across the same machines as the workers.

2.2.2 Distributed All-Reduce

In recent years, for data-parallel training of machine learning algorithms with dense updates (e.g., deep neural networks), all-reduce algorithms have grown in popularity as the means communication between workers [45, 87]. Unlike the parameter server architecture, all communication in distributed all-reduce happens directly between workers.

2.2.3 Consistency Models

In data-parallel ML, there are three commonly used consistency models: Bulk Synchronous Parallel (BSP), Stale Synchronous Parallel (SSP)[1] [54] and Asynchronous Parallel (ASP) [24] for synchronizing worker progress. In all three models, workers work on a local copy of model parameter data, receiving updated versions of the solution state from the parameter server, which sends the updated solution state once updates are received from all workers for the current clock (e.g., mini-batch). In BSP, workers block until their local copy is updated with model parameter data containing all the updates from clock. In ASP, workers never block, allowing them to work on stale data in favor of improved throughput. The SSP model is the middle ground between BSP and ASP. It generalizes BSP by allowing any worker to be up to a bounded number of iterations ahead of the slowest worker. So, for BSP, the bound would be zero. With a staleness of b, a worker at iteration t is guaranteed to see all updates from iterations 1 to $t-b-1$, and it may see (not guaranteed) the updates from iterations $t-b$ to t. Such a bound admits proofs of convergence [54, 67, 68].

[1] We refer to Stale Synchronous Parallel (SSP), but the concept has also been described as Bounded Delay consistency [67].

2.3 Example ML Tasks

This section describes six real ML tasks which are often parallelized to improve training time. In Chapter 3 and 4 we perform experiments using recommendation systems, image classification, and topic modeling tasks. In Chapter 5 we perform experiments using image classification, translation, language modeling, and video captioning tasks.

2.3.1 Recommendation Systems

Matrix Factorization is a technique (a.k.a. collaborative filtering) commonly used in recommendation systems, such as recommending movies to users on Netflix. The goal is to discover latent interactions between the two entities (e.g., users and movies). Given a partially filled matrix X (e.g., a matrix where entry (i, j) is user i's rating of movie j), MF factorizes X into factor matrices L and R, such that their product approximates X (i.e., $X \approx LR$). Like others [41, 66, 28], our *MF* implementation uses the stochastic gradient descent (SGD) algorithm. Each worker is assigned a subset of the observed entries in X; in every iteration, each worker processes every element of its assigned subset and updates the corresponding row of L and column of R based on the gradient. L and R are stored in the parameter server.

Our *MF* experiments use the *Netflix* dataset, which is a 480k-by-18k sparse matrix with 100m known elements, and factor it into two matrices with rank 1000. We also use a synthetically enlarged version of the *Netflix* dataset that is 256 times the original. It is a 7683k-by-284k sparse matrix with 4.24 billion known elements with rank 100.

2.3.2 Image Classification

Multinomial Logistic Regression is a popular model for multi-way classification, often used in the last layer of deep learning models for image classification [65] or text classification [69]. In *MLR*, the likelihood that each (d-dimensional) observation $\boldsymbol{x} \in \mathbb{R}^d$ belongs to each of the K classes is modeled by *softmax* transformation $p(\text{class}{=}k|\boldsymbol{x}) = \frac{\exp(\boldsymbol{w}_k^T \boldsymbol{x})}{\sum_j \exp(\boldsymbol{w}_j^T \boldsymbol{x})}$, where $\{w_j\}_{j=1}^K$ are the linear (d-dimensional) weights associated with each class and are considered the model parameters. The weight vectors are stored in the parameter server, and we train the *MLR* model using SGD, where each gradient updates the full model [18].

Our *MLR* experiments use the *ImageNet* dataset [82] with LLC features [102], containing 64k observations with a feature dimension of 21,504 and 1000 classes.

2.3.3 Topic Modeling

Latent Dirichlet Allocation is an unsupervised method for discovering hidden semantic structures (*topics*) in an unstructured collection of *documents*, each consisting of a bag (multi-set) of *words*. *LDA* discovers the topics via word co-occurrence. For example, “Sanders” is more likely to co-occur with “Senate” than “super-nova”, and thus “Sanders” and “Senate” are categorized to the same topic associated with political terms, and “super-nova” to another topic associated with scientific terms. Further, a document with many instances of “Sanders” would be assigned a topic distribution that peaks for the politics topics. *LDA* learns the hidden topics and the documents’ associations with those topics jointly. It is used for news categorization, visual pattern discovery in images, ancestral grouping from genetics data, community detection in social networks, and other such applications.

Our *LDA* solver implements collapsed Gibbs sampling [46]. In every iteration, each worker goes through its assigned documents and makes adjustments to the topic assignment of the documents and the words. The *LDA* experiments use the *Nytimes* dataset [3], containing 100m words in 300k documents with a vocabulary size of 100k. They are configured to classify words and documents into 1000 topics.

2.3.4 Deep Neural Networks (DNN)

Although it is a model type, rather than a task, we describe deep neural networks (DNNs) here. DNNs consist of a sequence of layers of different types (e.g., convolutional, fully connected, batch normalization) depending on the task being solved. Each layer computes a function over its inputs, using parameters (or weights) that are learned, and produces outputs. For simplicity, we refer to the first layer as the *input* layer and the last layer as the *output* layer of the DNN.

DNN training proceeds in a repeating loop composed of a forward work followed by a backward work (also known as back propagation). In *forward* work, training samples are fed to the input layer. Each layer performs its computation (e.g., convolution using its weights as filters) and propagates its results to the next layer. This proceeds until the output layer. At the end of forward work, the output layer makes a class prediction for the image. Based on the actual class of the image, the output layer then computes a loss (or error). This kicks off *backward* work, where each layer computes 1) the error for the previous layer, and 2) weight gradients for that layer, to move the DNN’s prediction towards the desired output. To efficiently utilize GPU resources, forward and backward work typically involve training on multiple samples simultaneously; this procedure is called *mini-batching*. This two-pass process iterates over all input samples in the training dataset, with samples grouped into mini-batches. A single iteration over the entire data set is referred to as an *epoch*. DNN training usually runs for multiple epochs, until the model reaches a desired level of accuracy or stops improving.

We use seven different DNN models across four different machine learning tasks:

- **Image Classification.** For image classification we use three different machine learning models. VGG-16 [89], ResNet-50 [52], and AlexNet [65]. We train VGG-16 and Resnet-50 using the the ImageNet-1K (ILSVRC12) [82] dataset. For AlexNet, we use synthetic data (otherwise, data loading is the bottleneck) and measure throughput.
- **Translation.** For translation we use the Google Neural server Translation (GNMT) with 8 LSTM layers [104], and GNMT with 16 LSTM layers. We train the translation tasks using the WMT16 English to German dataset, and then use the Newstest2014 dataset for validation.
- **Language Modeling.** For language modeling, we use the AWD Language Model

(LM) [73]. We train it using the Penn Treebank (PTB) [74] dataset.

- **Video Captioning.** For video captioning we use the S2VT [100] sequence-to-sequence model. We train it using the Microsoft Video description corpus (MSVD) [21].

Chapter 3

Addressing the Straggler Problem in Iterative Convergent ML

ML algorithms vary, and this chapter focuses on a major subset: iterative convergent algorithms solved in a data-parallel manner. As discussed in Chapter 2, such algorithms begin with a guess of the solution and proceed through multiple iterations over the input data to improve the solution. Most distributed implementations of such algorithms follow the Bulk Synchronous Parallel (BSP) computational model. Input data is divided among worker threads, each of which iterates over its subset of the input data and determines solution adjustments based on its local view of the latest parameter values. All workers execute the same iteration at the same time, enforced by barriers, and solution adjustments from one iteration are exchanged among workers before the next iteration begins. When many workers are involved, regular barrier synchronization often induces large slowdowns, due to straggler problems.

A straggler problem arises whenever worker threads experience uncorrelated performance jitter. In each iteration, under BSP, all workers must wait for the slowest worker in that iteration, so one slowed worker causes unproductive wait time for all the others. Unfortunately, even when load is balanced, transient slowdowns are common in real systems (especially in shared clouds) and have many causes, such as resource contention, garbage collection, background OS activities, and (for ML) stopping criteria calculations. Worse, the frequency of such issues rises significantly when executing on multi-tenant computing infrastructures rather than dedicated clusters (as is becoming increasingly common) and as the number of workers and machines increases.

Straggler mitigation techniques based on *redundant task execution* [14, 34, 108, 13] have been applied successfully to data processing jobs that fit map-reduce-style BSP execution (e.g., in Hadoop [15] or Spark [107]), relying on the *idempotency* of redundantly executed tasks. But, the most efficient frameworks for distributed ML do not work that way. Instead, these frameworks share state and exploit ML-specific properties to reduce coordination overheads and converge far faster [70, 67, 29, 11, 24, 22, 8]. Because the changes to shared state are *not idempotent*, new approaches to straggler mitigation are needed.

This chapter describes FlexRR, a new approach to straggler mitigation without the correctness problems of redundant task execution, for iterative convergent ML on efficient

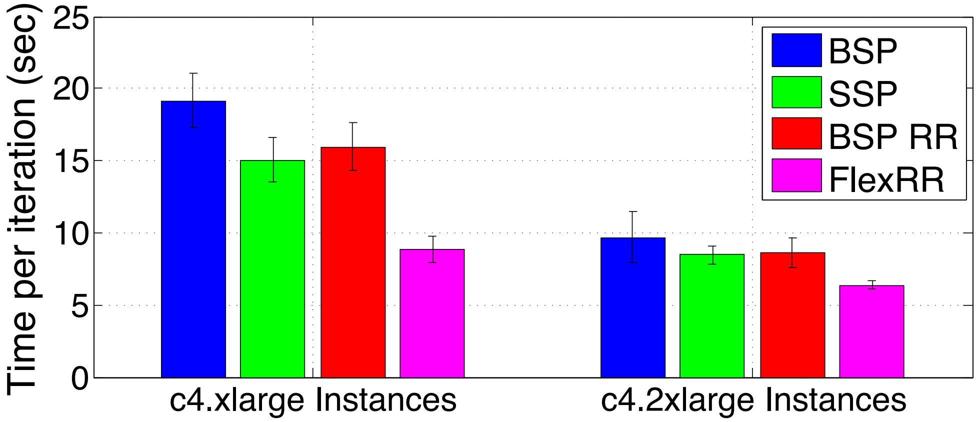


Figure 3.1: Comparison of Matrix Factorization performance on EC2. The graph shows average time-per-iteration running on 64 EC2 instances. For each of two EC2 machine classes, four approaches are compared: "BSP" and "FlexRR" represent the traditional approach and our solution, which combines flexible consistency bounds with our temporary work re-assignment technique. The "SSP" and "BSP RR" bars show use of individual ones of these two primary techniques, demonstrating that neither alone addresses the straggler problem for iterative convergent ML. FlexRR outperforms BSP and SSP by 53% and 39% (left) and 35% and 25% (right), for these two EC2 machine classes. Experimental details are in Section 3.4.

ML frameworks. FlexRR combines flexible consistency bounds with a new temporary work reassignment mechanism we call RapidReassignment. Flexible consistency bounds via SSP remove the barriers of BSP, allowing fast workers to proceed ahead of slowed workers by a bounded amount [25, 54, 67]. The flexibility improves efficiency, while the bound enables convergence to be proven [54, 68]. With RapidReassignment, a slowed worker can offload a portion of its work for an iteration to workers that are currently faster, helping the slowed worker catch up. The two techniques complement each other, and both are necessary to address the straggler problem for iterative convergent ML. Flexible consistency bounds provide FlexRR with enough slack to detect slowed workers and address them with RapidReassignment, before any worker reaches the bound and is blocked.

FlexRR's RapidReassignment is a specialized form of work shedding optimized for large-scale data-parallel iterative ML: (1) It takes advantage of the well-defined notion of progress through an iteration to identify slowed workers quickly. (2) It uses P2P communication among workers to detect slowed workers and perform work re-assignment, bounding communication overhead and avoiding a central decision-making bottleneck. (3) It uses explicit *helper groups*, limiting to which other workers any given work can be offloaded, to minimize data movement and enable input data preloading. (4) Optionally, it exploits iteration knowledge to further reduce how much data needs to be preloaded on helpers. Overall, RapidReassignment's design enables efficient and scalable temporary work reassignment for state-of-the-art efficient ML frameworks.

Extensive experiments demonstrate that FlexRR successfully addresses the straggler problem for three real ML tasks and various real and injected straggler behaviors. As illustrated in Figure 3.1, FlexRR reduces time-per-iteration by 35–53% on average, compared to the traditional BSP approach, and by 25–39% over SSP, even for relatively small ML problems on relatively expensive Amazon EC2 instances (which would be expected to have minimal resource sharing with other tenant activities). Similar results are observed in experiments on Microsoft Azure. In addition, various synthetic stragglers behaviors drawn from prior studies are used for controlled study of a wider range of scenarios; FlexRR consistently nearly matches an "ideal" lower bound in which all work is at all times perfectly balanced at no overhead, resulting in 5–10× improvement over BSP and SSP in extreme cases.

3.1 Prior Approaches Addressing Stragglers

Stragglers have long plagued parallel computing, and many techniques have been developed to mitigate them.

Eliminating performance variation. The HPC community—which frequently runs applications using the BSP model—puts significant effort into identifying and removing sources of performance jitter from the hardware and OSs of their supercomputers [40, 78]. Naiad [77] uses the same approach. While this approach can be effective at reducing performance "jitter" in specialized and dedicated machines, it does not solve the more general straggler problem. For instance, it is not applicable to programs written in garbage-collected languages, does not handle algorithms that inherently cause stragglers during some iterations, and does not work for today's multi-tenant computing infrastructures [31, 39, 53].

Blacklisting is a limited form of performance variation elimination, which attempts to mitigate stragglers by ceasing to assign work to workers that are falling behind. However, this approach is fragile. Stragglers caused by temporary slowdowns (e.g., due to resource contention with a background activity) often occur on non-blacklisted machines [32]. Worse, good workers that have such a temporary slowdown may then be blacklisted, unnecessarily reducing the computing power available.

Speculative execution and task cloning. Speculative execution is used to mitigate stragglers in data processing systems like MapReduce, Hadoop, and Spark [15, 34, 14, 108, 13]. Jobs in these systems consist of stateless, idempotent tasks like "map" and "reduce", and speculative execution runs slow tasks redundantly on multiple machines. While this consumes extra resources, it can significantly reduce job completion delays caused by stragglers, because the output from the first instance of any given task can be used without waiting for slower ones.

State-of-the-art frameworks for high-performance parallel ML use a *parameter server* architecture, which does not accommodate computation redundancy. While iterative ML can be built as a series of collections of idempotent tasks, doing so precludes many effective techniques for reducing overhead and speeding convergence [67, 28]. In parameter server systems, worker processing involves shared state and is not idempotent. Applying the same adjustments more than once can affect convergence negatively or even break algorithm invariants. FlexRR uses peer-to-peer interactions among workers to offload work when necessary, avoiding the wasted resources and potentially incorrect behavior of redundant work.

Work stealing, work shedding. Work stealing and work shedding are mirror approaches for adaptively re-balancing work queues among workers [19, 38, 9, 37]. The concept is to move work from a busy worker to an idle worker. FlexRR's temporary work reassignment mechanism is a form of work shedding, specialized to the nature of data-parallel iterative ML. There are several key differences. First, FlexRR takes advantage of the well-defined notion of progress through an iteration to identify slowed workers early on and avoid delays; work stealing, in contrast, waits for a worker to idle before looking to steal work, incurring additional delays until work is found. Second, while work stealing is computation-centric (e.g., data is moved to the thread that steals the work), FlexRR carefully avoids data movement by limiting and pre-determining reassignment patterns to avoid expensive on-demand loading of input data and parameter state. Third, because of its focus on *transient* stragglers, FlexRR's reassignments are temporary—only for the remainder of an iteration. Finally, it is designed explicitly to work in conjunction with flexible consistency bounds, as discussed below.

Using less strict progress synchronization. The strict barriers of BSP can be replaced with looser coordination models. One approach is to reduce the need for synchronization by restricting communication patterns. For example, GraphLab [70, 71] programs structure computation as a graph, where data can exist on nodes and edges. All communication occurs along the edges of this graph, so non-neighboring nodes need not synchronize. However, GraphLab requires the application programmer to know and specify the communication pattern.

Albrecht et al. [12] describe partial barriers, which allow a fraction of nodes to pass through a barrier by adapting the rate of entry and release from the barrier.

Yahoo! LDA [11] and Project Adam [24], as well as most solutions based around NoSQL databases, allow workers to run asynchronously (ASP), relying on a best-effort model for updating shared data. While such approaches can work well in some cases, they provide no guarantees of convergence for ML algorithms and indeed can readily diverge.

FlexRR uses SSP (Sec. 2.2.3). Consistent with our results, SSP has been shown to mitigate small transient straggler effects [28, 67] but not larger effects. FlexRR combines SSP with temporary work reassignment to address the straggler problem for iterative ML.

3.2 FlexRR Design & Implementation

FlexRR provides parallel execution control and shared state management for input-data-parallel iterative convergent ML algorithms. This section overviews FlexRR's API, basic execution architecture, shared state management approach, and solution to the straggler problem.

FlexRR's design relies on a few assumptions about application behavior. It assumes data-parallel processing, with worker threads processing assigned input data items independently and without order-dependence. It assumes iterations (or mini-batches, when used) are not too short to detect and react to slowed workers and that a worker's progress through an iteration can be measured, such as by the fraction of its input data items processed. Also, RapidReassignment's performance relies on being able to reassign work quickly; so, it assumes that there is either no cross-iteration data-item-specific local state or that there is a way to avoid needing to transfer it with reassigned work.[1] These characteristics are common to most data-parallel iterative ML applications, including our benchmark applications as well as K-means clustering, Sparse Coding and many others.

FlexRR is implemented as a C++ library linked by an ML application using it. During execution, FlexRR consists of one process executing on each node being used. Each FlexRR process starts a worker thread for each core on the node and a number of background threads for its internal functionality. The worker threads execute the ML application code for adjusting model parameters based on input data and possibly local state. The shared model parameters, which may be read and adjusted by all worker threads, are stored in a parameter server (Fig. 2.1) maintained by the set of FlexRR processes.

3.2.1 Workers and Execution Management

During initialization, an ML application provides FlexRR with the list of nodes/cores to be used, the input data file path, several functions called by FlexRR, and a stopping criterion. The input file contains data items in an understood format (e.g., rows that each contain one input data item in an easy-to-process format). The stopping criterion may be a number of iterations, an amount of time, or a determination of convergence. The most important function provided (`process-input`) is for processing a single input data item, taking the data item value as input and processing it to determine and apply model parameter adjustments as needed.

Each worker thread is assigned a unique ID, from zero to $N - 1$, and a disjoint subset of the input data items. The default assignment is a contiguous range of the input data, determined based on the worker ID, number of workers, and number of data items. Each worker has an outer loop for iterating until the stopping criterion is reached and an inner loop for each iteration.

3.2.2 Parameter Server for Shared State

FlexRR uses a parameter server which is derived from LazyTable [28, 29]. It exposes a simple key-value interface to the ML application code, which uses `read-param` and `update-param` functions to read or update (apply a delta to) a model parameter value specified by the key. The value type is application-defined, but must be serializable and have a commutative and associative aggregation function, such as plus, union, or multiply. With this property different worker threads can apply changes in any order without affecting the result. For the ML applications used in this chapter, the values are vectors and the aggregation function is addition.

The parameter server implementation reduces cross-node traffic by including a client-side cache for model parameter entries. While logically separate, the parameter server is part of the same FlexRR processes as the worker threads. Every process has a number of

[1] An example of the latter is our *LDA* application, which originally relied on local state. Instead of transferring local state between workers, which we found to be too inefficient, we designed an *LDA*-specific mechanism to avoid dependence on the local state.

parameter server threads and maintains a shard of the shared state. Each iteration updates are write-back cached, and asynchronously sent to the appropriate parameter server shards.

FlexRR supports both the BSP and SSP models. Each cached value is associated with an iteration number that indicates the latest iteration for which all workers' updates have been applied to it. During a read, the cached value is returned only if it reflects all updates up to the slack-bound (zero, for BSP); that is, the value is up-to-date enough if it reflects all updates from iterations more than "slack-bound" before the worker's current local iteration number. Otherwise, the read must proceed to the appropriate server shard to retrieve the value, possibly waiting there for other workers' updates.

For fault tolerance, FlexRR supports the same checkpointing mechanism as LazyTable [28]–its RapidReassignment mechanism does not affect it.

Although FlexRR is implemented using a parameter server architecture for data parallel training, we believe that it can also be implemented to work with machine learning training system that use distributed all-reduce (Sec. 2.2.2). The temporary work re-assignments in FlexRR are communicated directly worker-to-worker, independent of how parameters updates are communicated.

3.2.3 Straggler Mitigation

FlexRR combines two mechanisms, flexible consistency bounds via the SSP model and temporary work reassignments via our *RapidReassignment* protocol, to address the straggler problem for iterative ML. The SSP model allows each worker thread to be ahead of the slowest worker by up to a specified slack-bound number of iterations. This flexibility mitigates stragglers to some extent [28] (see also the SSP bars in Figure 3.1 and Section 3.4), but more importantly provides enough flexibility for RapidReassignment to be highly effective. RapidReassignment uses peer-to-peer communication to enable workers to self-identify as stragglers and temporarily offload work to workers that are ahead.

3.3 RapidReassignment Design

The goal of RapidReassignment is to detect and temporarily shift work from stragglers before they fall too far behind, so that workers never have to wait for one another. Workers exchange progress reports, in a peer-to-peer fashion, allowing workers to compare their progress to that of others. If a worker finds that it is falling behind, it can send a portion of its work to its potential helpers (a subset of other workers), which can confirm that they are indeed progressing faster and provide assistance (see Figure 3.2). Combined with SSP, RapidReassignment is highly effective in mitigating straggler delays of all intensities.

3.3.1 Worker Groups

RapidReassignment is designed for scalability, using peer-to-peer coordination among workers instead of a central arbiter. Like overlay networks [91, 81], workers exchange progress reports and offloaded work with only a few other workers, avoiding the scalability problems that would arise from all-to-all progress tracking or a centralized work reassignment approach, especially for short iterations when progress tracking and reassignment are more frequent.[2]

During initialization, each worker is assigned a group of workers that are eligible to provide assistance, referred to as its *helper group*, and a group of workers to whom the worker is eligible to provide assistance, referred to as its *helpee group*. The size of each group is set at start up and can be configured by the ML application. Each worker is assigned one helper on the same machine, and its other helpers are spread across different machines. While helper and helpee groups may overlap, they are usually not identical. For example, in a system containing 64 workers assigned round-robin to 8 machines and 4 helpers assigned to every worker, worker 14 might be eligible to assist workers (8,9,15,22) while workers (6,11,12,13) would be designated as its helpers. A waterfall effect results, whereby a worker providing a lot of assistance to some workers can in turn offload its own work to others, and so on, such that all workers make similar progress.

Worker groups also improve work reassignment efficiency. A helper needs to access the input data associated with the reassigned work. While it could fetch that data on demand, the helper's help is much more efficient if the data is fetched in advance. Indeed, based on our experiments, work reassignment is too slow to be helpful without doing so. Toward that end, each worker under RapidReassignment prefetches a copy of the input data of its helpee group members after loading its own data. The limited set of helpees bounds the cache space needed, which can be further reduced by caching only the tail of each helpee's iteration. Our experiments show that FlexRR suffers minimal performance loss from workers caching only the tail fraction of their helpees' input data (see Section 3.4.5).

3.3.2 Worker Communication

RapidReassignment uses non-blocking Message Passing Interface (MPI) for communication among workers. Workers explicitly poll for messages during each iteration, in order to compare their respective progress. The *message check frequency* parameter specifies how many times during each iteration a worker checks for incoming messages. The default setting is 100 checks per iteration, which our sensitivity experiments (Section 3.4.6) show is a good value.

To determine speed differences between workers, each worker keeps a *runtime timer* to track how long it has been running. These timers are launched during initialization, following a joint barrier. Because RapidReassignment addresses relatively large differences in progress (e.g., 20% of a multi-second iteration), with smaller differences mitigated by flexible consistency bounds, these timers are sufficiently precise and need to be resynchronized only infrequently (e.g., hourly).

[2] For small scale systems and longer iterations, our design could be readily adapted to use a central master to handle inter-worker coordination and reassignment. However, this approach would not outperform our P2P design in general. Moreover, the computation and communication of the master would compete with doing real work on that server, adding an additional straggler effect.

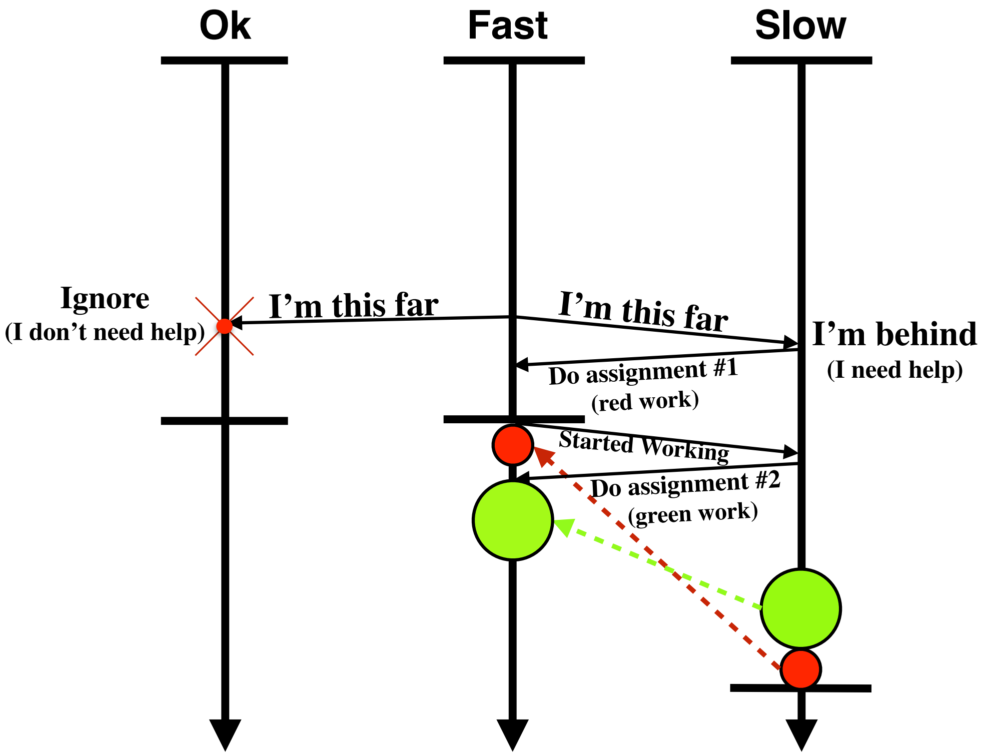


Figure 3.2: RapidReassignment example. The middle worker sends progress reports to the other two workers (its helpee group). The worker on the left is running at a similar speed, so it ignores the message. The worker on the right is running slower, so it sends a `do-this` message to re-assign an initial work assignment. Once the faster worker finishes its own work and begins helping, it sends a `begun-helping` message to the slow worker. Upon receiving this, the slow worker sends a `do-this` with a follow-up work assignment to the fast worker.

3.3.3 RapidReassignment Actions

This section describes the five primary RapidReassignment actions in FlexRR, designed to quickly offload work from slow workers to faster helpers (Figure 3.2). As will be seen, RapidReassignment is carefully tuned to the properties of data-parallel iterative convergent algorithms, such as the freedom to delay and reorder updates on the one hand, yet the need to avoid duplicating work on the other (recall that duplicating work can lead to incorrect behavior).

Identifying Stragglers. Upon reaching the *progress checkpoint* in the current iteration, which by default is set to 75% completion, a worker sends out a `progress-report` message to its helpee group, containing its current iteration number and the local time in its runtime timer. During each message check, a worker checks for `progress-report` messages from its helpers. Upon receiving such a message, the worker calculates its progress compared to the progress of the eligible helper. The logic for calculating the progress difference is shown in Algorithm 1. The result informs the worker how far ahead or behind (as a percentage of the iteration) it is relative to the `progress-report` sender.

Algorithm 1 Progress Difference Calculation

```
$completion\_diff \leftarrow$ progress in the message *minus* progress of the current worker
$current\_avg \leftarrow$ weighted average time it takes the current worker to complete an iteration
$time\_diff \leftarrow$ timer value of the current worker *minus* timer value contained in progress message
$progress\_difference \leftarrow completion\_diff + \frac{time\_diff}{current\_avg}$
```

Reassigning Work. If a worker finds that it has fallen behind the sender of a `progress-report` by more than a set threshold (the *straggler trigger threshold*, with a default of 20%), it will send an initial work assignment in a `do-this` message back to the sender. This initial work assignment is a percentage of its work for the current iteration. Section 3.4.6 shows that a default of 2.5% is a good setting for this tunable. The `do-this` message contains the current iteration number of the slow worker, beginning and end of the work assignment (a range of the input data), and a local timestamp of the message. A sample message is `do-this (iteration: 4, start: 140, end: 160, timer: 134.43)`. Note that, as shown in Figure 3.2, the slow worker reassigns ranges of work starting from the end of its current iteration. Assigning from the end is the least disruptive to the slow worker's progression through its input data items, and takes advantage of the robustness of iterative convergent algorithms to processing data items in any order despite the data dependencies (such dependencies between item processing—resulting from their `update-param` calls—would make this reordering unsafe for general code).

Algorithm 2 Helping Decision

```
$msg \leftarrow$ check for helping requests
**if** $msg.timestamp \leq last\_cancellation\_timestamp$ OR $msg.iteration > current\_iteration$ **then**
    Discard $msg$
**else if** $msg.iteration < current\_iteration$ OR finished its own work this iteration **then**
    Send begun-helping and do the help
**else**
    Save $msg$ for the end of this iteration
**end if**
```

Helping with Work. Workers check for `do-this` messages on every message check. Upon receiving a `do-this` message, the worker (the potential helper, in this case) will compare the timestamp of the message to the timestamp of the latest `cancel-help` message (see below) from the same worker. If the timestamp in the `do-this` message is greater, the potential helper will compare its current iteration to the iteration number contained in the `do-this` message. If the iteration number contained in the message is smaller than the helper's current iteration, the helper will immediately send a `begun-helping` message and begin working on the work assignment. Upon completing the work assignment, the worker will send a `help-completed` message to the original worker and check for additional `do-this` messages prior to returning to its own work.

If the iteration number in the do-this message equals the helper's current iteration (as in assignment #1 in Figure 3.2), then the helper will put aside the work assignment until the end of the current iteration. If at the end of the iteration the worker has yet to receive a cancel-help message containing a timestamp greater than the timestamp of the do-this message, the helper will send out a begun-helping message and begin working on the work assignment. Upon completing the work assignment, the helper will send a help-completed message to the original worker (the helpee) and, after checking for additional valid do-this messages, will move on to its own next iteration. Algorithm 2 shows the pseudo-code for the worker's decision about if and when to provide assistance.

Assigning Additional Work. After much experimentation, we have found that a good strategy for balancing various concerns is to first assign a relatively small amount of initial work, immediately followed by a larger amount (double) of additional work once a helper begins processing the initial assignment. To that end, after a worker sends out a do-this message, it will check for begun-helping messages during every message check in that iteration. If such a message is received, and more help is needed, the worker will send an additional do-this message to the faster worker, containing a follow-up work assignment of twice the size (see Figure 3.2). For the rest of the iteration, the worker will send another follow-up work assignment each time it receives a begun-helping message.

Cancelling Work Reassignments. After reassigning a portion of its work, a worker will continue working on its current iteration until it completes all the work it has not given away. At this point, for all pending do-this messages the worker has sent out, the worker will check for begun-helping messages. If there is a work assignment for which a begun-helping message has yet to be received, the worker will send out a cancel-help message containing the current timestamp and complete the work on its own. Upon completing all such messages, the worker will wait to receive a help-completed message for all work assignments before moving on to the next iteration. This is is done to guarantee the *slack-bound*. There is a small window in which both the helpee and a helper may begin the same work, which can be addressed by having the helpee only commit changes corresponding to reassigned work after it confirms that the helper acknowledged the cancel-help message. Again, we are relying here on the robustness of data-parallel iterative convergent algorithms to delayed and out-of-order updates.

3.4 Evaluation

This section evaluates the effectiveness of FlexRR. Results are reported for sets of Amazon EC2 and Microsoft Azure instances as well as for local clusters. The results support a number of important findings: (1) significant straggler problems occur in real cloud infrastructures, (2) when straggler problems occur, FlexRR greatly outperforms BSP and SSP, achieving near-ideal performance for all of the various straggler patterns studied; (3) to achieve ideal performance, the RapidReassignment and SSP techniques need to be combined, as is done by FlexRR, as neither alone is sufficient; (4) FlexRR is not sensitive to the choices of run-time configuration parameters, within a wide range of reasonable settings.

3.4.1 Experimental Setup

Experimental Platforms. We use a variety of clusters for our experiments. **Cluster-A** is 16 virtual machines running on a dedicated cluster of 16 physical machines, each with a 2 quad-core Intel Xeon E5430 processor running at 2.66GHz, connected via 1 Gbps Ethernet (≈700 Mbps observed). Each VM runs on one physical machine, and is configured with 8 vCPUs and 15 GB memory, running Debian Linux 7.0. **Cluster-B** is a cluster of 64 Amazon EC2 c4.2xlarge instances. Each instance has 8 vCPUs and 15 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM). **Cluster-C** is a cluster of 64 Amazon EC2 c4.xlarge instances, a lower class version of Cluster-B. Each instance has 4 vCPUs and 7.5 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM). From our testing using `iperf`, we observe a bandwidth of 1 Gbps between each pair of EC2 instances. **Cluster-D** is a cluster of 64 Microsoft Azure A4 Standard instances. Each instance has 8 vCPUs and 15 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM) on Intel Xeon E5507 processors. **Cluster-E** is a cluster of 64 Microsoft Azure A3 Standard instances. Each instance has 4 vCPUs and 7 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM) on AMD Opteron 4171 HE processors. From our testing using `iperf`, we observed a bandwidth of 1.1 Gbps between each pair of Azure instances. **Cluster-F** is a PRObE Nome [42] dedicated cluster of 128 high-end computers running Ubuntu 14.04. Each machine contains 4 quad-core AMD Opteron 8354 CPUs (16 physical cores per machine) and 32GB of RAM. The machines are connected via 1Gb Ethernet.

For experiments that control and/or instrument straggler causes, we primarily use Cluster-A. The other five clusters are used to experiment with naturally occurring stragglers in two public clouds and for a larger-scale example. We use our limited access to Cluster-F to experiment on a large problem and dataset that does not fit on the other clusters.

Naturally-occurring and Injected Straggler Patterns. Our goal is to experiment with a wide variety of straggler patterns that are likely to be encountered in practice, as well as with more extreme patterns that provide stress tests. Our experiments with EC2, Azure, and Nome provide evaluation in the presence of a variety of naturally-occurring stragglers in several real infrastructures. But, we are unable to instrument these systems to evaluate the causes or particular nature of those stragglers; we consistently observe straggler problems, but whatever happens happens when using public clouds. To evaluate a broader range of straggler effects and intensities beyond what arose during the particular times of the experiments, and to directly measure attributable delays, we also perform more controlled experiments with injected transient stragglers, using three distinct methodologies:

Slow Worker Pattern: Models transient worker slowdown by inserting *sleep* commands into worker threads. At each of 10 possible delay points within an iteration, each worker decides (independently) to be slowed, with 1% probability, for a period uniformly randomly chosen between 0–2× the duration of an iteration. Naturally, multiple (or no) workers may be slowed at any given time. We denote the *transient delay intensity %* (*delay %* for short) to be the percentage by which a worker is slowed (e.g., 100% delay means runs twice as slow). To simulate the effect of a worker being slow, we divide each iteration into 1000 parts and insert milliseconds-long *sleep* commands at each of these 1000 points. For a delay % of d within a t second iteration, each of these sleeps are $d \times t$ milliseconds long. For example, for a 50% delay within a 6 second iteration, we insert a 3ms sleep at each point.

Disrupted Machine Pattern: Models transient resource contention (e.g., due to sub-machine allocation or a background OS process) by running a disruptor process that takes away CPU resources. Every 20 seconds, each machine independently starts up a disruptor process with 20% probability. The *disruptor* launches a number of threads that each executes a tight computational loop for 20 seconds. For a transient delay intensity % of d on a p-core machine running p application threads, the disruptor launches $d \times p$ processes of its own. For example, on our 8-core machines running 8 worker threads, a 200% delay means having the disruptor launch 16 threads. Such a delay experienced by a worker for a whole iteration will cause it to run roughly 200% slower than without delays.

Power-Law Pattern: Based on a real-world straggler pattern [78], this uses a power-law distribution [79] to model the time t for a worker to complete an iteration: $p(t) \propto t^{\alpha}$, where α is the parameter that controls the "skewness" of the distribution. `sleep` commands are used, as in the *Slow Worker Pattern*, to extend an iteration as determined. Smaller α values make the distribution more "flat" and lead to more delay on average. Each experiment uses a fixed α, and the iteration time of each worker is chosen independently from this distribution. When we set the α parameter to 11, the iteration times in our emulated environment without FlexRR have the same distribution as was measured on real clusters in [78].

We also study a *persistent* straggler pattern where half the machines get 75% of the work per iteration—such uneven workloads could arise in cases where data processing skew is correlated with data placement.

Systems Compared. We compare the speed and convergence rates of four modes implemented in FlexRR: [3]

BSP	Classic BSP execution
SSP	SSP execution
BSP RR	BSP with our RapidReassignment
FlexRR	Our solution
Ideal	Best possible (computed lower bound)

We also compute a value termed "Ideal", which represents the speed that should be achieved if all work is at all times perfectly balanced with no overhead. Reporting results for BSP RR and SSP (without RapidReassignment) enables us to study the impact of RapidReassignment and flexible consistency bounds in isolation versus in combination, as in FlexRR. For SSP, we use a slack-bound of 1 in all experiments after verifying that it leads to the fastest convergence on these benchmarks.

FlexRR features several run-time configuration parameters such as *helper group size* and *work assignment sizes*. Table 3.1 lists these parameters, the range of values studied, and their default values. The default values are the best settings obtained after extensive

[3] Although we do not show results of comparisons to other systems, the base system in which we integrated FlexRR compares favorably to state-of-the-art frameworks, as noted in Section 3.1. For example, its BSP mode is faster than GraphLab [70, 43] by 10–14× for MF and 50–100% for LDA [29], which in turn has been shown to outperform Hadoop and Spark implementations [70, 44]. It also outperforms efficient single-threaded implementations of MF and LDA by 99× and 62×, respectively, when using 8 64-core machines [29].

experimentation over the range of parameters shown. Section 3.4.6 provides a sensitivity analysis on the parameter settings, showing that FlexRR performs well over a broad range of settings.

Table 3.1: FlexRR Parameter Settings

Parameter	**Range**	**Default**
Helper group size	2–16	4
Initial work assignment	1.25%–15%	2.5%
Follow-up work assignment	2.5%–30%	5%
Message checks/iteration	20–50k	100
Straggler trigger threshold	10%–40%	20%

Experimental Methodology. Every experiment was run at least thrice, and we report arithmetic means. In experiments that had injected stragglers, the first run was conducted from smallest delay injections to largest, the second in reverse order, and the third in random order.

3.4.2 Naturally-occurring Straggler Results

We performed experiments on Amazon EC2, Microsoft Azure and PRObE Nome to evaluate FlexRR in the presence of naturally-occurring straggler effects observed during the particular times of the experiments. No synthetic straggler effects are injected during these experiments.

Amazon EC2 results. Figure 3.1 and Figure 3.3 show the results for *MF* and *LDA*, respectively, Cluster-B (c4.2xlarge VMs) and Cluster-C (c4.xlarge VMs). Using c4.2xlarge VMs, FlexRR reduces time-per-iteration by 35% (25%) for *MF* and by 34% (15%) for *LDA* relative to BSP (SSP, respectively). Using c4.xlarge VMs, the reductions are 53% (39%) for *MF* and 49% (32%) for *LDA*. The improvements are larger for c4.xlarge VMs, because these less expensive VMs experience more transient straggler effects.

The improvements on EC2 come despite executing relatively short experiments on relatively expensive EC2 instances that would be expected to have minimal resource sharing with other tenant activities, highlighting the real-ness of transient stragglers in cloud infrastructures.

Microsoft Azure results. Figure 3.5 shows the results of *MF* on Cluster-D (A4 VMs) and Cluster-E (A3 VMs) on Microsoft Azure. Using the A4 VMs, FlexRR reduces time-per-iteration by 43% (32%) relative to BSP (SSP, respectively). Using the A3 VMs, FlexRR reduces time-per-iteration by 56% (38%) relative to BSP (SSP). While the A4 instances are bigger and more expensive VMs, the times-per-iteration are larger than on the A3 instances because the A3 CPUs perform better on MF's floating point computations. Nonetheless, significant straggler effects are observed for both setups, and the results are akin to those for the c4.xlarge VMs on EC2.

PRObE Nome large MF experiment. To verify FlexRR's effectiveness for a larger workload, we used the *Netflix * 256* synthetically enlarged dataset on Cluster-F. Figure 3.4 shows that FlexRR reduces time per iteration by 21% over SSP and 51% over BSP, even

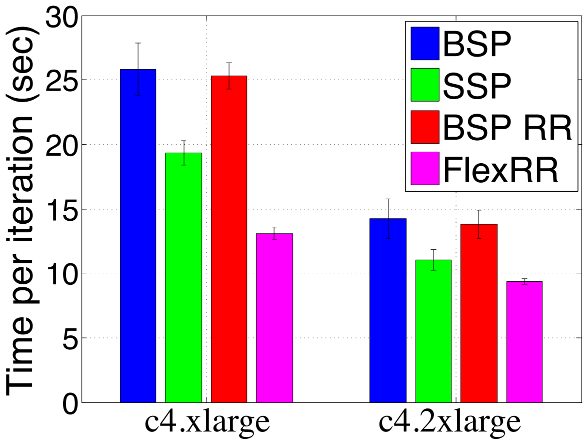


Figure 3.3: EC2, LDA, no injected delay.

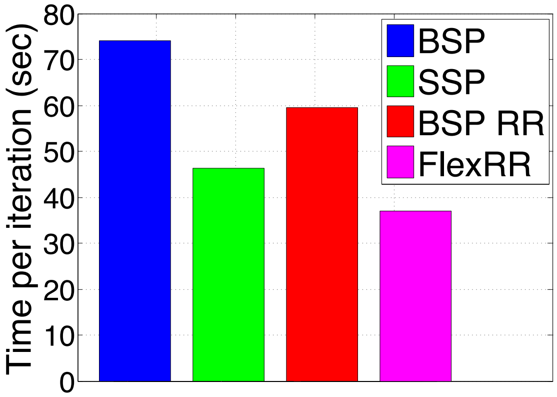


Figure 3.4: PRObE Nome, large MF, no injected delay.

on this dedicated cluster with no injected delays. As expected, more straggler effects are observed as cluster size increases, and FlexRR effectiveness is not hampered by the increased problem size.

3.4.3 Slow Worker Pattern Results

This section studies the speed and convergence rate of the ML tasks under the *Slow Worker Pattern*.

Speed Tests

For each task, we measured the time-per-iteration, $\frac{Overall\ run\ time}{Number\ of\ iterations}$, for the four modes, varying the transient delay intensity %. Each experiment ran for 20 iterations. (Running more iterations yields the same results.)

Results on Cluster-A. Figure 3.6(a) and Figure 3.6(b) show the results for the *MF* and *LDA* applications running on Cluster-A. *MLR* results looks similar to *MF* (not shown due to space constraints). BSP slows down linearly with delay intensity. By controlling straggler intensity, we see that SSP can mitigate delays below its slack-bound (e.g., see the 50% delay intensity points), but then too suffers linearly. (The natural stragglers from Section 3.4.2 were clearly too big for SSP alone.) FlexRR, on the other hand, nearly matches Ideal even up to 400% delays, which are more extreme than should be expected in practice. We measured the percentage of work that gets reassigned by FlexRR: it ranges from 8–9% of the work at 0% delay (i.e., no injected delays) to 19–22% at 400% delay. Even at 0% delay, FlexRR runs 18% faster than BSP on *MF* and *LDA* and 13% faster than BSP on *MLR*. The figures also show that our RapidReassignment technique can be used in BSP to decrease its straggler penalty, but it is FlexRR's combination of flexible consistency bounds and RapidReassignment that nearly matches Ideal.

At high delay % values, there is some divergence from Ideal for *LDA*. That is because *LDA* uses a special mechanism to handle its local state, which involves two extra model

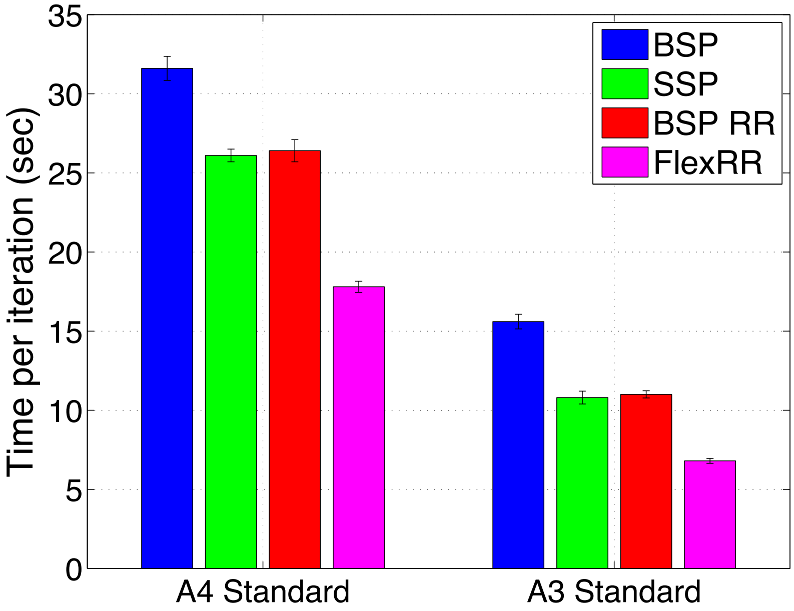


Figure 3.5: Microsoft Azure, MF, no injected delay.

parameter updates for each work re-assignment. At higher delays, more work is re-assigned, thus these extra updates begin to have an effect on the run-time, causing FlexRR to deviate from Ideal.

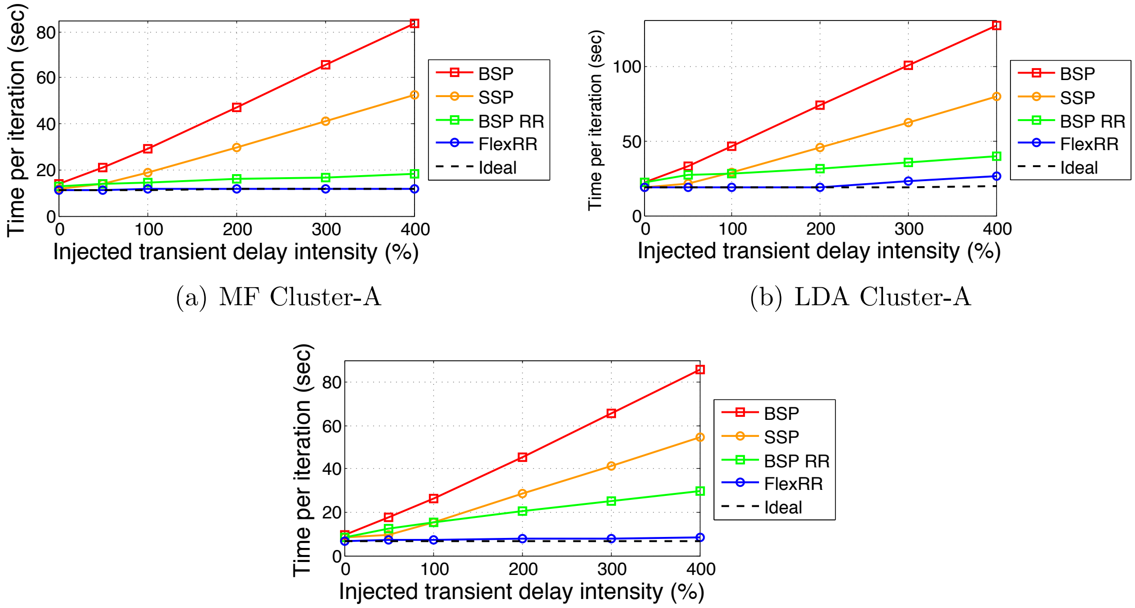
Results on Cluster-B. Figure 3.6(c) shows the results for *MF* on the larger Amazon EC2 cluster, which are qualitatively the same as on Cluster-A. As on Cluster-A, BSP slows down linearly with delay intensity, SSP can mitigate stragglers only up to its slack-bound, and FlexRR nearly matches Ideal. FlexRR reassigns 21% of the work at 0% delay and 31% at 400% delay. The main difference between the Cluster-B and Cluster-A results is that on Cluster-B there is an even larger separation between FlexRR and the next best approach (BSP with our RapidReassignment). E.g., at 400% delay, BSP RR is 10 times slower than FlexRR. At 0% delay (no injected delays, corresponding to Figure 3.1 (right)), FlexRR is 35% faster than BSP and 25% faster than SSP, because of non-injected performance jitter. The results for *LDA* on Cluster-B are qualitatively as on Cluster-A.

Convergence Tests

We also measure the time to convergence for the ML tasks running in each of the modes. We calculate Ideal by multiplying the Ideal time-per-iteration values from Section 3.4.3 by the number of iterations needed to reach convergence by the BSP experiment.[4]

Criteria for Convergence. We use the following stopping criterion, based on guidance from our ML experts: If the objective value (for *MF*) or log-likelihood (for *LDA*) of the solution changes less than 2% over the course of 10 iterations, then convergence is considered

[4] We use BSP iterations in this lower bound because the flexible consistency bounds of FlexRR and SSP can lead to a (modest) increase in the number of iterations needed [25, 54], e.g., 2-3 extra iterations in our experiments.

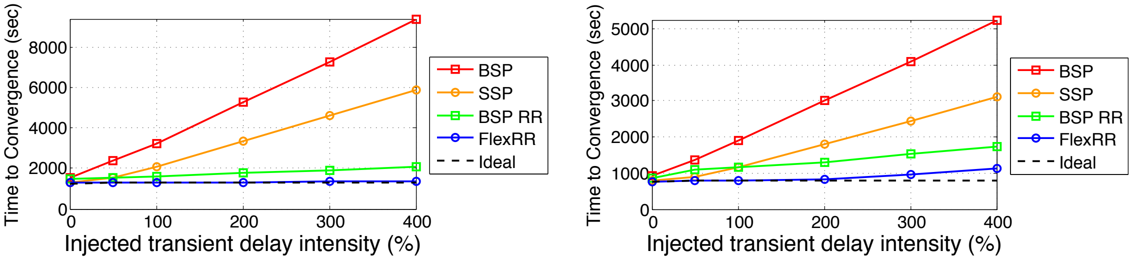


(a) MF Cluster-A

(b) LDA Cluster-A

(c) MF Cluster-B

Figure 3.6: Slow Worker Pattern Speed Tests.



(a) MF Convergence

(b) LDA Convergence

Figure 3.7: Convergence Tests.

to have been reached. We also verified that they reached the same objective value. Because the objective value calculation is relatively expensive, and we wanted to observe it frequently, we did it offline on FlexRR checkpoints.

Convergence Test Results. Figure 3.7(a) shows the results for *MF*. For all delay %, BSP (and BSP RR) required 112 iterations to reach convergence and SSP required 113 iterations. FlexRR required 114 iterations, with the exception of 400% delay, where it took 115 iterations. Even with the extra iterations required to reach convergence, FlexRR converged 10% faster than BSP at 0% delay injected. With delays injected, BSP suffered from linear increase in convergence time, while FlexRR effectively matched the Ideal convergence time even at 400% delay. As expected, adding RapidReassignment to BSP improves its convergence times, to faster than SSP but still much slower than FlexRR.

Figure 3.7(b) shows the results for *LDA*. For all delay %, BSP required 41 iterations to converge. For 0%, 50%, and 100% delays, FlexRR required 42 iterations, for 200%

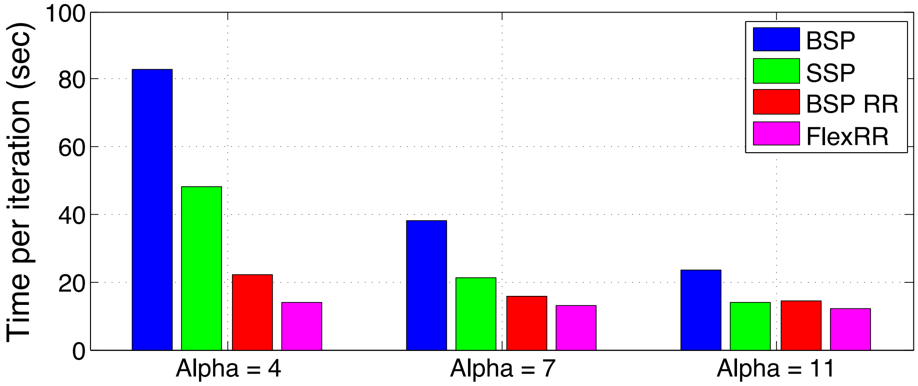


Figure 3.8: MF Speed, Power-Law Pattern.

and 300% delays it required 43 iterations, and for 400% delay it required 44 iterations. Despite the need for these extra iterations, FlexRR converges significantly faster than BSP. With no injected delays, FlexRR converged 18% faster than BSP, and maintains near-Ideal convergence time with increasing delays. BSP, on the other hand, suffers from a linear increase in convergence time when delays are injected. *LDA* deviates from Ideal at higher delays for the same local state issue discussed in Section 3.4.3.

3.4.4 Other Straggler Patterns

Disrupted Machine Pattern. We compare the average time-per-iteration (20 iterations) of FlexRR to the alternative modes for the *Disrupted Machine Pattern.* Figure 3.9 shows results for *MF* on Cluster-A—results for *LDA* and *MLR* are qualitatively similar. SSP and BSP RR individually reduce the delay experienced by BSP by up to 49% and 42%, respectively. The combination of the two techniques in FlexRR matches Ideal, reducing the run-time by up to 63%.

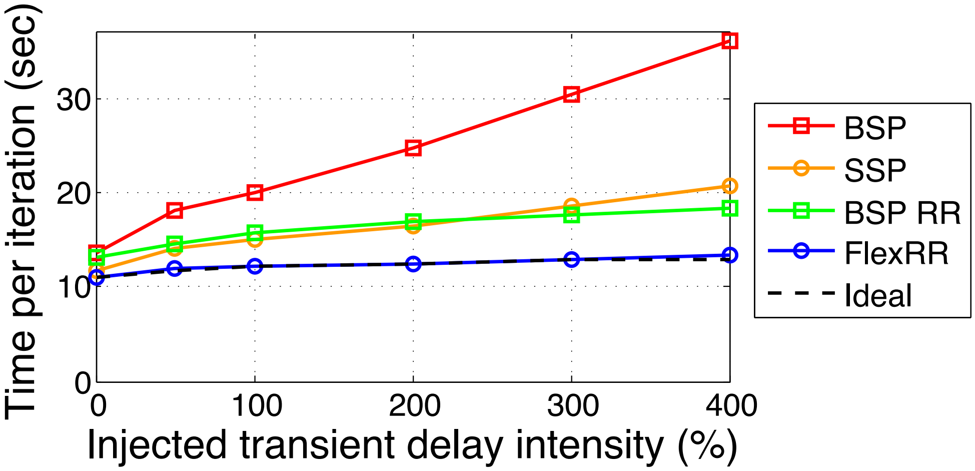


Figure 3.9: Disrupted Machine Pattern.

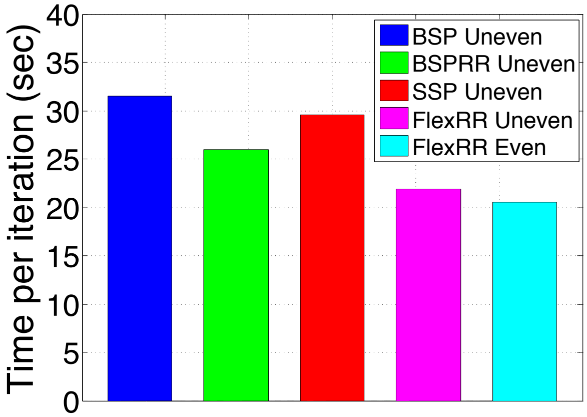


Figure 3.10: Uneven Workload

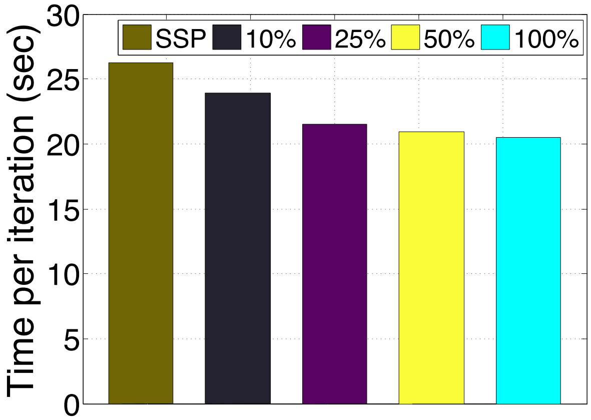


Figure 3.11: Partial Replication

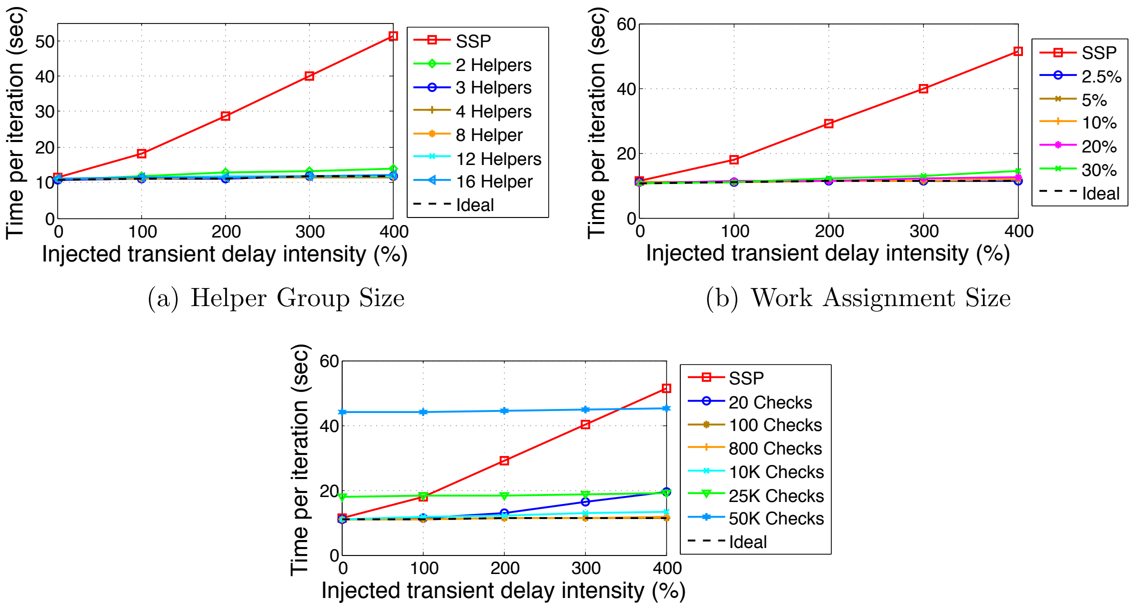
Power-Law Pattern Results. Next, We compare the average time-per-iteration (20 iterations) of FlexRR to the alternative modes for the *Power-Law Pattern*. We present results on Cluster-A for each of our applications, setting α to 4, 7, and 11. Recall that $\alpha = 11$ emulates a real cluster measured in [78], and the configurations with smaller α values yield more severe delay. Figure 3.8 shows the results for *MF*. For $\alpha = 11$, SSP and BSP RR are faster than BSP by 39% and 40%, respectively. When the two techniques are combined in FlexRR, the run-time is 48% faster than BSP. Similarly to experiments conducted in earlier sections, with increasing delays (smaller α), the other three modes experienced significant increases in run-times, while FlexRR experienced only slight increases.

The results for *MLR* and *LDA* show similar trends. For $\alpha = 11$, SSP and BSP RR were 36% and 31% respectively faster than BSP for *MLR* and 37% and 42% respectively faster than BSP for *LDA*. FlexRR was 43% and 52% faster than BSP on *MLR* and *LDA* respectively. With increasing delays (smaller α), the other three modes experienced significant increases in run-times for both *MLR* and *LDA*. FlexRR experienced only modest delays for *MLR* and somewhat larger delays for *LDA* (not shown due to space constraints). In all cases, FlexRR significantly outperforms the other three modes.

Uneven Workload Distribution. While FlexRR was originally designed to mitigate transient stragglers, it is also effective at dealing with long-term workload differences among workers. Figure 3.10 shows an experiment on Cluster-A where half of the machines are assigned 75% of the workload, and the remaining half of machines are assigned 25% of the workload. FlexRR was able to mitigate the straggler effects of the uneven workload distribution, running at close to ideal speed shown in the FlexRR Even Bar, while SSP experienced a 54% slowdown.

3.4.5 Partial Replication

In all previous experiments, workers replicated 100% of the input data belonging to the workers that they are helping. We found that FlexRR is still effective when workers replicate only a portion of the input data, and thus are only eligible to help with that portion. Figure 3.11 shows the *MF* application with no injected delays run on Cluster-A with different percentages of the input data replicated on the helper workers. FlexRR with just 25% replication is close to FlexRR with 100% replication and much better than SSP (and BSP—not shown).



(a) Helper Group Size

(b) Work Assignment Size

(c) Message Check Frequency

Figure 3.12: Sensitivity Tests.

3.4.6 Sensitivity Study

This section reports on tests used to determine good settings for FlexRR parameters. We vary each parameter across its Table 3.1 range while using the default values for other parameters. For brevity, we show sensitivity results for *MF*'s average time-per-iteration (for 20 iterations) when running on Cluster-A under the Slow Worker straggler pattern, although similar results hold for the other two applications, convergence time, and the other clusters (same default settings used in all experiments, on all clusters).

Helper Group Size Test. Recall that the *helper group* is the set of workers to whom a worker is eligible to provide assistance. Figure 3.12(a) shows the results of varying the *helper group size* from zero helpers, which is equivalent to running in SSP mode, to sixteen helpers for each worker. The results show that, once the *helper group size* is set to 3 or higher, near-Ideal performance is achieved. Closer inspection reveals that using four helpers provides the best performance. But, the difference between settings from 3 to 16 is negligible.

Work Assignment Size Test. One of the key design decisions was the amount of work to be re-assigned in `do-this` assignment messages. Work assignments occur in two different sizes, an initial work assignment size and a follow-up work assignment size. Figure 3.12(b) shows the results of varying the *follow-up work assignment size* from 0% (equivalent to SSP) to 30%. The *initial work assignment size* is always half the *follow-up work assignment size*. Near-Ideal performance is achieved across the range of non-zero sizes, although as delays increased, the larger work assignments do perform worse. This occurs because of a rare corner case where workers that run slowly can re-assign a portion of their work to a faster worker that starts to complete the extra work but then is delayed significantly. Because the current implementation of FlexRR does not look to reassign work that has already been reassigned and accepted, other workers end up waiting in this case. This corner case is not a problem for smaller work assignments, because the helper does not fall behind significantly.

Message Check Frequency Test. FlexRR depends on messages between workers to keep track of progress and re-assign work. The *message check frequency* is the number of times a worker checks for incoming messages during an iteration. If the checks are not performed often enough, the system runs the risk of not reacting fast enough, while checking too often can cause an unnecessary overhead. Figure 3.12(c) shows that any frequency between 100 and 10K performs well, but the performance suffers once the frequency is greater than 10K.

3.5 Summary

This Chapter presents FlexRR, which addresses the straggler problem for iterative convergent machine learning. FlexRR combines flexible synchronization bounds with temporary work re-assignment in order to successfully minimize synchronization time for workers. Our experiments demonstrated improvements of up to 10× compared to traditional work division combined with BSP and SSP.

Chapter 4

Agile ML Elasticity Through Tiered Reliability in Dynamic Resource Markets

Many modern compute infrastructures offer a great opportunity: transient availability of cheap but revocable resources. For example, Amazon EC2's spot market and Google Compute Engine's preemptible instances often allow customers to use machines at a 70–80% discount [4] off the regular price, but with the risk that they can be taken away at any time. Many cluster schedulers similarly allow lower-priority jobs to use resources provisioned but not currently needed to support business-critical activities, taking the resources away when those activities need them. ML model training could often be faster and/or cheaper by aggressively exploiting such revocable resources.

Unfortunately, efficient modern frameworks for parallel ML, such as TensorFlow [8], MxNet [22], and Petuum [105], are not designed to exploit transient resources. Most use a *parameter server* architecture (Sec. 2.2.1), in which parallel workers process training data independently and use a specialized key-value store for shared state, offloading communication and synchronization challenges from ML app writers [67, 54, 24]. Like MPI-based HPC applications, these frameworks generally assume that the set of machines is fixed, optimizing aggressively for the no machine failure and no machine change case (and restarting the entire computation from the last checkpoint on any failure). So, using revocable machines risks significant rollback overhead, and adding newly available machines to a running computation is often not supported.

This chapter describes Proteus—a parameter server system that combines agile elasticity with aggressive acquisition strategies to exploit transient revocable resources. Figure 4.1 illustrates the benefits for one ML example on Amazon EC2. Using three on-demand instances and up to 189 spot market instances, Proteus reduces cost by 85% when compared to using only on-demand instances, even when accounting for spot market variation and revocations, while running 24% faster. Compared to using a standard bidding strategy with a checkpointing-based approach (i.e., run on spot market machines and checkpoint regularly to retain progress if evicted [47, 88, 92]), Proteus reduces cost by ≈50% and runtimes by 32–43%, winning by avoiding checkpoint overheads, reducing restart delays, and exploiting

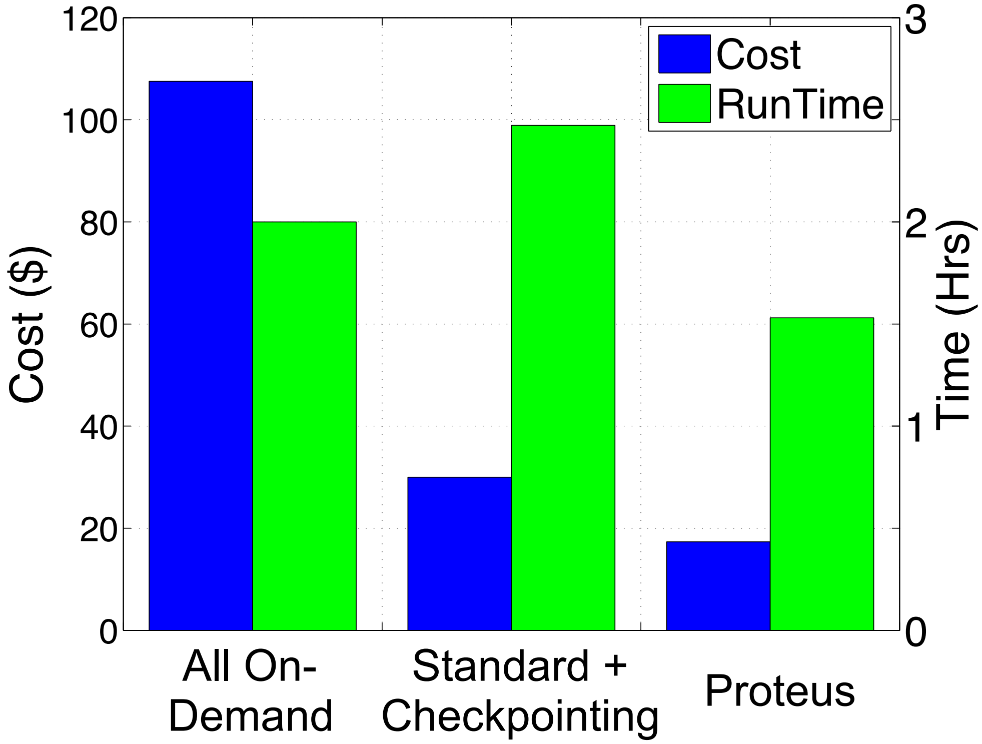


Figure 4.1: Cost and time benefits of Proteus. This graph shows average cost (left axis) and runtime (right axis) for running the *MLR* application (see Section 2.3) on the AWS EC2 US-EAST-1 Region. The three configurations shown are: 128 on-demand machines, using 128 spot market machines with checkpoint/restart for dealing with evictions and a standard strategy of bidding the on-demand price, and Proteus using 3 on-demand and up to 189 spot market machines. Proteus reduces cost by 85% relative to using all on-demand machines and by ≈50% relative to the checkpointing-based scheme. Full experimental details can be found in Section 4.5.

spot market properties.

Proteus consists of two principal components: AgileML and BidBrain. The AgileML parameter-server system achieves agile elasticity by explicitly combining multiple reliability tiers, with core functionality residing on more reliable resources (non-transient resources, like on-demand instances on EC2) and most work performed on transient resources. This allows quick and efficient scaling, including expansion when resources become available and bulk extraction of revoked transient resources without big delays for rolling back state or recovering lost work. AgileML transitions among different modes/stages as transient resources come and go. When the ratio of transient to non-transient is small (e.g., 2-to-1), it simply distributes the parameter server functionality across only the non-transient machines, instead of across all machines, as is the usual approach. For much larger ratios (e.g., 63-to-1), the one non-transient machine would be a bottleneck in that configuration. In that case, AgileML uses non-transient machine(s) as on-line backup parameter servers (BackupPSs) to active primary parameter servers (ActivePSs) that run on transient machines. Updates are coalesced and streamed from actives to backups in the background at a rate that the network bandwidth accommodates.

BidBrain is Proteus' resource allocation component that decides when to acquire and yield transient resources. BidBrain is specialized for EC2, exploiting spot market characteristics in its policies, but its general approach would apply to other environments with transient resources (e.g., private clusters or other cloud costing models). It monitors current market prices for multiple instance types, which move relatively independently, and bids on new resources when their addition to the current footprint would increase work per dollar. Similarly, resources near the end of an hour may be released if they have become less cost-effective relative to others. As part of its considerations, BidBrain estimates the probability of getting free compute due to instance revocation within the billing hour (with later in the hour being better than earlier) for different bids and spot market conditions. Simultaneously considering costs (e.g., revocation and scaling inefficiencies) and benefits (e.g., cheaper new resources), BidBrain finds a happy medium between aggressive bidding on transient resources and more conservative choices.

Experiments with three real ML tasks (recommendation systems, image classification, topic modeling) confirm that Proteus achieves significant cost and runtime reductions. AgileML's elasticity support introduces negligible performance overhead, scales well, and suffers minimal disruption during bulk addition or removal of transient machines. In breaking down the sources of benefit, we find that both the agile elasticity of AgileML and the aggressive policies of BidBrain are needed—using either one alone (e.g., BidBrain with checkpointing instead of AgileML) achieves half or less of the cost and runtime savings.

This chapter makes four primary contributions. First, it describes the first parameter server ML framework (Proteus) designed to elastically scale with bulk additions and revocations of transient machines. Second, it describes an adaptive architecture+algorithm (AgileML) for exploiting multiple tiers of machine reliability (i) to more agilely resize in the face of such changes and (ii) to balance work given different ratios of non-transient to transient resources. Third, it describes a new resource manager (BidBrain) that aggressively exploits EC2 spot market properties to achieve major cost savings. Fourth, it presents results from experiments and analyses showing that aggressive multi-tier exploitation of transient machines is both possible and beneficial, reducing costs and runtimes significantly.

4.1 Motivation and Background

This section overviews transient availability of revocable cluster/cloud resources, and what is needed for machine learning training frameworks to exploit them.

4.1.1 Dynamic Availability of Revocable Resources

As discussed in Section 2.1, today's cluster infrastructures are increasingly dynamic. This section describes how transient resources are made available in several modern infrastructures.

Amazon AWS EC2 Spot Market. Amazon AWS EC2 [1] is a public cloud that allows customers to purchase time on virtualized machine resources. In addition to "on-demand" machines, which are released at a time of the customer's choosing, Amazon also offers "spot market" for machines, where machines are often available at a steep discount (e.g., 70–80% lower price) with the proviso that they can be taken back at any time.

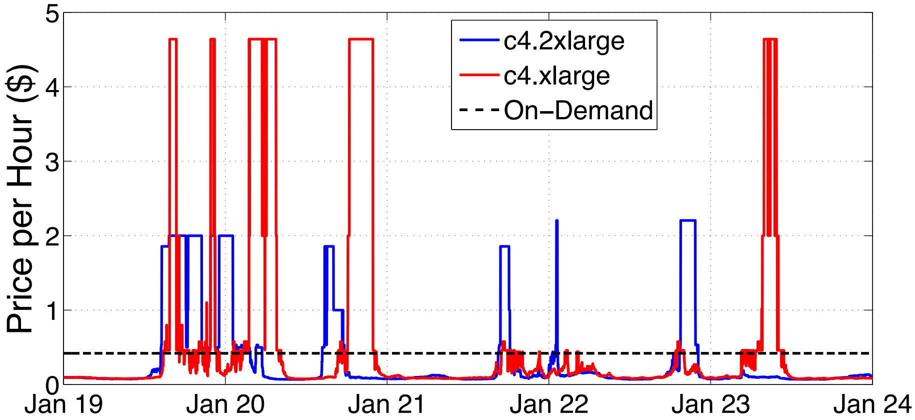


Figure 4.2: AWS spot prices over time. Spot prices for two classes of machines are shown for 6 days in January 2016. The unchanging on-demand price for c4.2xlarge is shown, and the values shown for c4.xlarge are doubled so that all three lines show the price for the same number of cores; c4.2xlarge machines have 8 cores and c4.xlarge machines have 4 cores.

The EC2 spot market design has interesting properties that affect customer savings and the likelihood of eviction. First, it is not a free market [10]. Customers specify their *bid prices* for a given machine class, but generally do not pay that amount. Instead, a customer is billed according to the current EC2-determined *spot price* for that machine class. Fig. 4.2 shows one example of spot price variability over a week, for two machine classes in an EC2 zone. Second, if a customer receives machine resources in response to their bid price, they will retain those resources until either they release them or the spot price rises above the customer's bid price. If the latter occurs, the customer is not billed for the current hour, but the resources are taken back from the customer. Third, EC2 does not guarantee any warning when resources are going to be revoked, but since 2015 EC2 has provided a two-minute warning prior to eviction in most cases. Fourth, once a customer submits a bid and receives a resource, the bid price cannot be changed. The bid can be canceled, if not yet granted, and a new bid price submitted. But, once the resource is granted, the bid price cannot be changed until the resource is terminated.

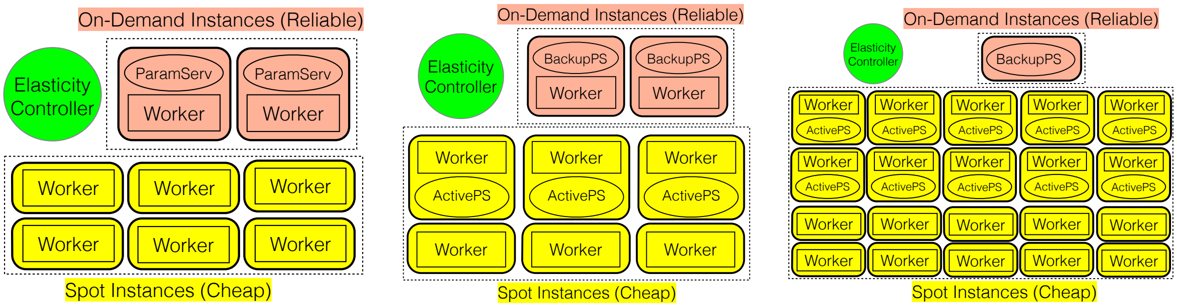
Google Preemptible Instances. Google Compute Engine (GCE) [2] offers revocable machine resources, called "preemptible instances", akin to those provided by the EC2 spot market. Google preemptible instances can be revoked at any time, as the name suggests, but differ from EC2's spot market in several ways. First, Google charges a fixed price of 70% less than the "on-demand" (non-revocable instance) price for the requested machine type. There is no price variability. Second, GCE offers a 30-second warning, rather than a 2-minute warning. Third, GCE limits preemptible instances to 24 hours.

Dynamic Resource Offers in Mixed-Function Corporate Clusters. Many corporate clusters serve a mix of online services, business critical batch analytics jobs (often with deadlines), and ad hoc jobs (often called "best effort") for application development, exploratory data analyses, etc. Business critical activities are usually given priority, but extra resources are often available for ad hoc jobs. Moreover, modern schedulers for such clusters, such as YARN [99] and Mesos [53], have mechanisms to offer recently-freed resources to already running jobs' "application master" components, allowing some of them (e.g., large map-reduce jobs) to elastically grow to higher performance levels by spreading work over more machines. But, these resources may subsequently be revoked if higher priority workloads intensify or additional jobs arrive [31, 99, 101].

4.1.2 Exploiting Transient Resources for ML

To exploit transient resources, ML frameworks need to couple agile elasticity with good resource allocation strategies. The elasticity must accommodate efficient bulk extraction of revoked transient resources with little-to-no warning, which is akin to correlated failures. Existing solutions that address ML framework elasticity include checkpointing [47, 88, 92] and Spark RDDs. Spark RDDs in particular allow fine-granularity Spark application checkpointing and rollback to handle resource revocations. Elasticity to transient resources can, thus, be achieved by relying on the fault tolerance mechanisms of RDDs. There are two problems with this, first, RDDs work well only for deterministic computation. Second, for highly correlated bulk revocations, the amount of recovery work approaches that of the checkpointing mechanisms (see Sec. 4.7).

Section 4.2 describes Proteus, which is a parameter server system that exploits resource reliability tiers (e.g., stable EC2 on-demand instances and transient spot market instances) to achieve agile elasticity efficiently. Section 4.3 describes BidBrain, which uses an aggressive strategy of bidding on multiple spot markets to minimize the cost of training ML models. Section 4.4 describes how these two components are combined in Proteus.



(a) Stage 1

(b) Stage 2

(c) Stage 3

Figure 4.3: Three stages of AgileML architecture. Stage 1: ParamServs only on reliable machine. Stage 2: ActivePSs on transient and BackupPSs on reliable. Stage 3: No Workers on Reliable Machines.

4.2 AgileML Design

This section describes AgileML, Proteus' elastic ML training component. AgileML introduces the concept of "tiers of reliability" to organize resources into tiers based on their expected reliability (and associated cost) and deploy different functional components of the ML framework to different tiers. AgileML, which is implemented as a C++ library linked by an ML application using it, is built upon the parameter server architecture described in Sec. 2.2.1. Functional components include workers, parameter servers (ParamServs), and newly introduced Active and Backup parameter servers (Tab. 4.1). These solution state servers hold ML model parameter state and have different fault tolerance expectations. AgileML uses different combinations of these components to allow safe and agile exploitation of different quantities of transient resources.

4.2.1 Workers and Execution Management

During initialization, an ML application provides AgileML with an initial list of reliable and transient nodes to be used, the input data file path, several functions called by AgileML, and a stopping criterion.[1] During execution, AgileML consists of one process executing on each node. Each process then starts one worker thread per core. The worker threads execute the ML application code for model training—adjusting model parameters as a function of input (training) data and current solution state. Each worker thread operates on a disjoint subset of input data items. By default, input data is partitioned evenly amongst workers. Workers iterate on the data until reaching the stopping criteria.[2]

[1] The stopping criterion may be a number of iterations, an amount of time, or a determination of convergence.

[2] This is an over-simplification. For greater flexibility, AgileML actually provides a notion of a *clock of work* that gets executed on each iteration. It may be some number of data items (a "mini-batch" of an iteration) or some number of iterations.

ParamServs	Serve solution state for workers and always run on reliable resources
BackupPSs	Serve as a hot backup for solution state served by ActivePSs and always runs on reliable resources
ActivePSs	Serve solution state for workers, periodically pushing aggregated updates to BackupPSs, and run on transient resources

Table 4.1: Types of solution state servers used by AgileML

4.2.2 Architecture

This section describes how AgileML uses reliability tiers and the mechanism of moving between its different stages of execution. At a high level, AgileML enables ML applications to run on a dynamic mix of reliable and transient machines, maintaining the state required for continued operation on reliable machines, while taking advantage of transient machine availability. AgileML uses three stages of system functionality partitioning in order to avoid bottlenecking the reliable nodes, as the ratio of transient to reliable resources grows..

Stage 1: Parameter Servers Only on Reliable Machines. For most ML applications including K-means, DNN, Logistic Regression, Sparse Coding, as well as MF, MLR, and LDA (Sec. 2.3), the workers are stateless, while the ParamServs contain the current solution state. AgileML's first stage spreads the parameter server across reliable machines only, using transient nodes only for workers, thereby taking advantage of these two primary levels of machine reliability. This has the effect of removing all solution state from transient machines. Fig. 4.3(a) illustrates a running example of eight AWS EC2 machines with six spot instances (transient) and two on-demand instances (reliable).

Pros: By removing parameter state from transient resources, AgileML is able to utilize them without losing progress when transient resources are revoked (or fail). Unlike a traditional parameter-server architecture, no checkpointing is required to assist with using transient resources.[3] **Cons:** While stage 1 successfully removes state from transient resources, it causes a network bottleneck (Sec. 4.5.3) when the ratio of transient to reliable resources grows too large. With 60 transient and 4 reliable machines, for example, the network bottleneck to the ParamServs slows the *MF* application by over 85%. Limiting this ratio is undesirable, as it caps achievable savings/benefits from transient resources.

Stage 2: ActivePSs on Transient Machines and BackupPSs on Reliable Machines. For higher transient to reliable node ratio, AgileML switches to stage 2 (Fig. 4.3(b)). Stage 2 uses a primary-backup model for parameter servers, using transient nodes for an active server (ActivePS) and reliable nodes for the hot standby (BackupPS). This shifts the heavy network load from the few reliable resources to the many transient resources. Solution state is sharded across the set of ActivePS instances. Workers send all updates and read requests to the ActivePSs, which update their state and push updates in bulk to the

[3] In AgileML, there is benefit in checkpointing the reliable resources in case they fail, however as we show later in this section, this checkpointing has no overhead on system performance in stages 2 and 3.

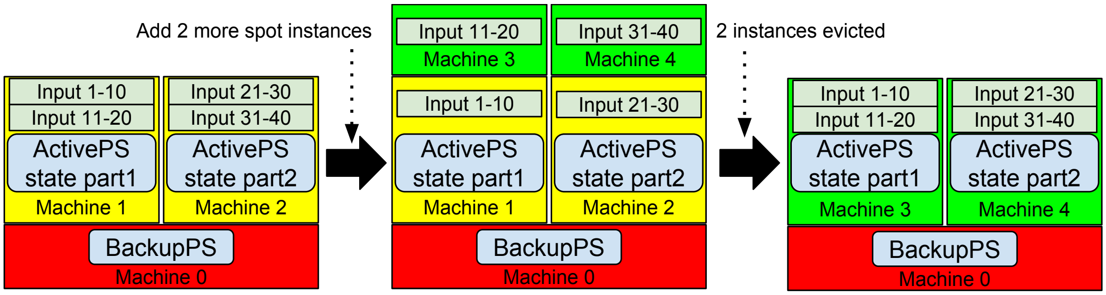


Figure 4.4: AgileML component and data transitions as resources are added and evicted. In this toy example, there are 40 pieces of input data. Initially, one on-demand Machine 0 runs BackupPS, and 2 spot instances (Machine 1,2) are processing $\frac{1}{2}$ of the input data each. 2 new spot instances (Machine 3,4) are added, at the same time, price, of the same type, and shown in the same color (we refer to these atomic sets as *allocations*, described in Sec. 4.3). Each new instance $\in \{3,4\}$ loads $\frac{1}{2}$ of the input data, but works only on $\frac{1}{4}$ of it. An eviction of the 2 yellow spot instances triggers the second transition. The remaining spot instances assume ownership of the evicted input data with minimal delay.

BackupPSs. Solution state affected by transient node failures or evictions is recovered from BackupPSs. Stage 2 improves on stage 1 for higher transient-to-reliable ratios (Sec. 4.5.3) but loses to an all-reliable baseline by 2x for ratios exceeding 63:1.

Stage 3: No Workers on Reliable Machines. Workers collocated with BackupPSs on reliable machines were found to cause straggler effects at transient-to-reliable ratios beyond 15:1, causing Proteus' performance drop relative to the PS baseline. Stage 3 simply removes these workers (Fig. 4.3(c)), allowing AgileML to match all-reliable performance levels (Sec. 4.5.3). The optimal ratio threshold to switch to stage 3 depends on the network bandwidth, transient-to-reliable ratio and the size of the model.

Transitioning Between Stages. AgileML dynamically transitions between these stages to match the number of transient nodes available. Transitioning between stages 1 and 2 involves switching between a set of ParamServs and the active/backup PS pair. This process is described in Sec. 4.2.3. When scaling up, workers are directed to send their requests to ActivePSs started in the background. When scaling down, ActivePSs push their updates to BackupPSs, which become ParamServs. The worker requests are then redirected to the ParamServs. This transition is done with minimal overhead in the background. Transitioning between stages 2 and 3 boils down to re-assigning work between reliable and transient resources. Scaling up, work is offloaded from workers on reliable nodes to workers on transient nodes, followed by shutting down the former workers. Transitioning back to stage 2 requires reassigning input data to reliable workers. This change of assignment incurs zero run-time overhead, as it involves just a single worker notification message.[4]

Elasticity Controller: This component of AgileML makes decisions to switch between stages based on the transient-to-reliable ratio and the network bandwidth. It is responsible for (a) tracking which resources are participating in ongoing computations, (b) assigning

[4] Input data assigned to workers on reliable resources is preloaded by workers on transient resources, simplifying the transition from stage 2 to 3.

a subset of input data to each worker, and (c) starting new ActivePSs. On eviction, it reshards the solution state and shuts down ActivePSs using policies discussed next.

4.2.3 Handling Elasticity: Policy and Mechanism

The toy example in 4.4 illustrates how AgileML handles adding and removing resources from an ongoing computation. We evaluate AgileML's effectiveness at handling such elasticity in 4.5.5.

Scaling Up. *Workers.* Once a node becomes available, and the appropriate software has been initialized, it contacts the elasticity controller responsible for the job and receives its input data assignment (see transition to phase 2 in 4.4). It loads the data (from S3 storage for AWS EC2) and signals the elasticity controller that it's ready. The elasticity controller then signals corresponding workers to update their working sets. *ActivePS.* AgileML achieves best performance when running ActivePSs on half of the resources (Sec. 4.5.3). This ratio is thus maintained when scaling the resource footprint. When the resource footprint increases, AgileML starts new ActivePSs on the longest running transient resources that do not yet have an ActivePS. It notifies the resource to host the ActivePS and serve a given partition assignment. A partition is a unique subset of the parameter state. During start-up, AgileML divides the parameter state into N partitions, where N is the maximum number of ActivePSs that can exist at any one point. We found that setting N equal to half of the maximum number of resources that could be used by AgileML at any point to be effective. Each partition is assigned to a ParamServ. In stage 2 and 3 each partition is also assigned to an ActivePS, which is responsible for forwarding updates applied to the partition to the BackupPS that owns it. By using partitions in this way, AgileML avoids the need to re-shard the parameter state when adding or removing servers, instead re-assigning partitions as needed.

The resource that starts the new ActivePS contacts the previous partition owner for a copy of the partition. The original owner points all workers to the new partition owner. During ownership propagation, all partition messages are forwarded to the new ActivePS. Workers and ActivePS additions happen in the background with negligible impact on system performance (Sec. 4.5.5).

Scaling Down. AgileML differentiates between evictions and failures based on whether it received a warning, and it handles them differently. When resources are removed from an ongoing computation after some warning, such as the two-minute warning offered by AWS or the 30-second warning offered by GCE, we call this an *eviction*. When resources are removed without warning or with a warning detected with insufficient lead time, we call this a *failure* or an *effective failure*, respectively.

Evictions. AgileML's elasticity controller checks for eviction warnings every 5s. These warnings consist of a set of instances marked for eviction, if any. When this set includes all transient resources, the elasticity controller signals all ActivePSs to push their most recent consistent state to the BackupPSs and cease operation. A special end-of-life flag is appended to these updates to signal the last message from ActivePS to BackupPS. When the BackupPSs receive end-of-life messages from ActivePSs, they signal any workers on reliable machines (including those getting turned on by the elasticity controller, as discussed at the end of this section) to address read and update requests to them. Note that the AgileML design makes this scenario simple and fast.

When an eviction is about to take back only some of the transient resources, the elasticity controller signals the ActivePSs that are being evicted to either (i) move their partitions to the ActivePSs that will survive the eviction, or (ii) move them to transient resources that are going to survive the eviction and do not yet have an ActivePS running on them (see transition to phase 3 in Fig. 4.4). The process for relocating partitions mirrors the process of adding ActivePSs above, which includes pointing all surviving workers to the new partition owner.

Failures. In the case of failures, which are detected via heartbeat messages, or effective failures, when the eviction warning is not early enough for all evicted ActivePSs to send their end-of-life messages to BackupPSs, AgileML performs a form of on-line roll-back recovery. This roll-back recovery depends on how many resources have failed.

When all or most of the transient resources fail (usually due to an effective failure), the BackupPSs will use the last consistent state[5] from the ActivePSs as the new solution state, and the workers will re-do the work lost in the roll-back recovery. The ActivePSs send the workers the iteration number of the last iteration included in the new solution state, and all workers will restart from what is essentially an online checkpoint. When a single or few resources running ActivePSs fail, the elasticity controller reassigns partition ownership from those ActivePSs to other transient resources. This is done by the BackupPSs sending their solution states to the new owners of the ActivePSs. The surviving ActivePSs then roll-back to a state consistent with the current state of the BackupPSs. This roll-back to a consistent state is straightforward, because the ActivePSs already store the aggregate of the delta applied to their local state since the last time they applied their state to the BackupPSs.

Reacting to the eviction and failures of workers is orchestrated by the elasticity controller. When a worker is removed from a computation, the previous owner of the worker's input data takes ownership for it. A previous owner always exists when input data is assigned to a transient node. Thus, there will be no need to stop and load the input data from storage. To account for the infrequent failure of reliable resources, checkpointing of reliable resources can be used. In stage 3 of AgileML, checkpointing of reliable resources has no overhead on ML training speed because there are no worker threads running on these resources.

Stage Transitions. AgileML uses the ratio of transient to reliable resources to determine which stage to use. For ratios greater than 1:1, AgileML uses stage 2, and for ratios greater than 15:1, it uses stage 3 (Sec. 4.5.3). While transitioning between stages is important for AgileML performance, as the ratio of transient to reliable resources changes, we find that perfect threshold settings are not required. For our work, appropriate thresholds for different compute clusters were determined by measuring and comparing system performance for the three stages at different ratios (Sec. 4.5.3), resulting in the 1:1 and 15:1

[5] Recall that parameter server systems often allow flexibility in progress synchronization among workers and shared state consistency. Often, workers see parameter values that do not yet reflect recent updates from all other workers, but a bound on the staleness is often enforced [28, 67]. In such systems, the consistent state corresponds to the latest common iteration and reflects all updates up to that iteration and no updates afterwards. Achieving a consistent state requires either synchronization of worker progress or (usually) some extra buffer memory.

thresholds as well as the observation regarding low sensitivity. We believe that future work can automate the threshold selection process for any given cluster.

Hyperparameters. AgileML uses strong scaling and applies updates inside parameter server shards. By using strong scaling, where the global batch size is kept consistent, AgileML makes it unnecessary to tune the learning rate when workers are added or removed. It is also possible to perform weak scaling and tune the learning rate appropriately [45].

4.3 BidBrain Design

BidBrain is Proteus' resource allocation component. It keeps track of current and historical market prices for different types of resources (e.g., Amazon EC2 instance types) and makes allocation decisions of the form $\vec{x}$, where x_i corresponds to the set of instances allocated of type i. Fig. 4.5 illustrates how this allocation vector changes over time as the allocation footprint managed by BidBrain changes (due to BidBrain decisions or to evictions).

BidBrain decisions consider several parameters characterizing the application (4.2), including its ability to scale with more instances (ϕ), cost of modifying its resource footprint (σ), and the cost of evictions (λ). BidBrain's primary objective is to minimize cost per unit work. The current implementation of BidBrain focuses on Amazon EC2, but we believe that its mathematical framework and mechanisms can also be applied in other cloud provider settings.

Resource Allocation. BidBrain interfaces with AWS to acquire new resources. To do so, BidBrain supplies a (instance type, count, bid price) tuple. We call this an allocation request. An *allocation* is defined as a set of instances of the same type acquired at the same time and price. BidBrain's *total footprint* $\vec{x}$ is a set of such allocations that are the elements of $\vec{x}$. We use different colors in 4.4 and 4.5 to signify different atomic allocations. Resource allocation decisions are made periodically as well as a few minutes before the termination of each billing hour.

4.3.1 Formulation

At each allocation decision, BidBrain calculates the total expected cost and the total expected work by considering available instance types and their current spot market prices. BidBrain works with the following free variables: (a) instance type i to choose, (b) *bid delta* to bid over the spot price.

Expected Cost. Given a set of *allocations*, the total cost for a given footprint $\vec{x}$ is calculated as the sum over individual allocation costs $C_A\left[x_i\right]$, where each allocation's cost is calculated as the product of its instance count k_i and instance price P_i:

$$C_A = \sum_{i=0}^{n} C_A[x_i] = \sum_{i=0}^{n} k_i * P_i$$

One of BidBrain's features is to reason about the probability of free compute it can get if its instances are evicted before the billing hour expires. If the allocation is evicted, AWS

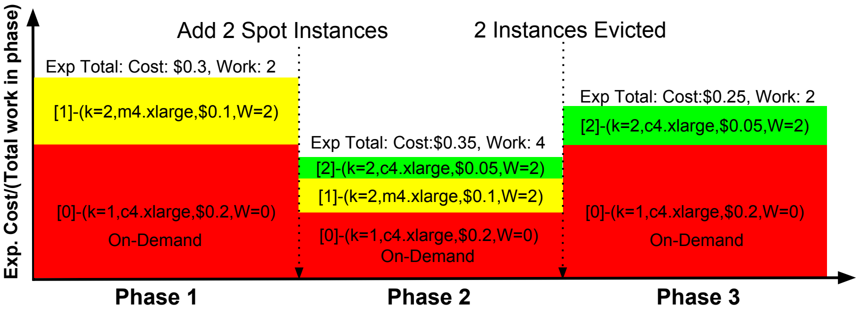


Figure 4.5: Expected cost per unit work for the toy example transitions in 4.4. Each block represents an *allocation* (Sec. 4.3), described by how many instances are in the allocation (k), instance type, the expected cost of the allocation, and the expected work produced by this allocation. Each block's height equates to that allocation's relative contribution to the cost of the total work done in its phase. Combining the blocks' heights in each phase equates to the total expected cost per unit work for that phase. In phase 1, BidBrain has an expensive, required on-demand allocation (red) that produces no work and a spot allocation (yellow). The on-demand instance type is pre-determined to be c4.xlarge and is never terminated by BidBrain, even if it negatively affects cost-per-work. In phase 2, BidBrain further amortizes the cost of the red allocation by adding a second spot allocation [2] (green), which lowers the total expected cost-per-work. This transition increases its actual cost at that moment, but reduces the final cost by decreasing the amount of time for which the on-demand allocation is needed.

refunds the amount charged at the beginning of the current billing hour. To capture this, BidBrain calculates the *expected* cost of an allocation by considering the probability of eviction β_i for a given instance type i at a given *bid delta*. There are only two possibilities: an allocation can either be evicted (with probability β_i) or it will reach the end of its billing hour in the remaining t_r minutes (with probability $1-\beta_i$). The expected cost can, therefore, be written down by the definition of expectation (Eq. 4.1).

$$C_A[x_i] = (1-\beta_i) * P_i * k_i * t_r + \beta_i * 0 * k_i * t_r \tag{4.1}$$

Estimating Evictions. To estimate β_i in Eq. 4.1, BidBrain uses historical AWS spot market price data. This historical data is gathered individually for each instance type in each availability zone and indicates the price at each instant in time. Combining such data with knowledge that spot instances are evicted when the price rises above the bid, BidBrain computes the historical probability of being evicted within the hour and the median time to eviction for a given bid delta. The *bid delta* is the difference between the bid price and the market price. Using the AWS spot market trace from March to June of 2016, we ran simulations with a wide range of *bid deltas* [$0.0001,$0.4] and recorded the probability of getting evicted within the hour, β, and the median time to eviction. Using this information, BidBrain estimates the probability of eviction for any allocation.

Expected Work. BidBrain explicitly reasons about the expected amount of work each allocation is expected to produce. To capture this, BidBrain computes the *expected* useful

β	Probability that allocation is evicted (0-1)
ϕ	How efficiently application scales (0-1)
σ	Overhead of adding/removing resources (min)
λ	Overhead of evicting resource (min)
ν	Work produced by instance type
ω_i	Max compute time remaining in allocation i
C_A	Expected cost of a set of allocations (\$)
W_A	Expected work of a set of allocations
E_A	Expected cost per work of a set of allocations

Table 4.2: Summary of parameters used by BidBrain

compute time Δt_i for each allocation by considering factors such as eviction overhead, overhead for resource addition, and scalability of the application.

The maximum useful compute time of any allocation is the time remaining in the current billing hour ω_i. If BidBrain expects the allocation to be evicted prior to the end of the billing hour, it reduces ω_i accordingly. The eviction of any allocation reduces the useful compute of each allocation x_i by the eviction overhead λ of the application. The probability of an eviction for a set of allocations is computed as: $1 - \prod_{j=0}^{n}(1 - \beta_j)$, where β_j is the probability of eviction for allocation j. When considering removing or adding resources, BidBrain subtracts this overhead σ from the expected compute time for each allocation (Eq. 4.2).

The expected work for an allocation is the product of its resources k_i, expected useful compute time Δt_i and the work produced per time by that instance type ν.[6] BidBrain expresses the expected work for a set of allocations as the sum of each allocation's expected work reduced by the scalability overhead ϕ of the application (Eq. 4.3).

$$\Delta t_i = \omega_i - (1 - \prod_{j=0}^{n}(1 - \beta_j)) * \lambda - \sigma \tag{4.2}$$

$$W_A = (\sum_{i=0}^{n} k_i * \Delta t_i * \nu) * \phi \tag{4.3}$$

4.2 summarizes the parameters used by BidBrain. In future work, we plan to automate the process of determining ϕ, σ, λ and ν. Currently, we set ϕ, σ, λ empirically (see experiment description in 4.5.4 and 4.5.5). ν is set to equal the number of virtual cores in the instance and is a proxy for how much work an application is expected to achieve on that instance per unit time. ϕ measures the first order Taylor series expansion coefficient of the application's scalability curve as a function of instance count of a given type. σ and λ measure for how long the application does not make progress after a change in the resource footprint.

[6] For ML workloads performed by AgileML, work produced is proportional to the number of cores on an instance. For example, ν of a c4.2xlarge instance (8 cores) is equivalent to 2 * ν of a c4.xlarge instance (4 cores).

4.3.2 Resource Acquisition

BidBrain acquires resources x_i only if they lower the footprint's expected cost per work. For a set of allocations in $\vec{x}$, it is approximated [86] as the expected cost divided by the expected work produced (Eq. 4.4).

$$E_A = C_A/W_A \tag{4.4}$$

During every "decision point", BidBrain builds a list of possible allocations that it can make. This set of possible allocations is constructed by pairing different bid prices with different instance types. The range of possible bid prices includes $[\$.0001, \$.4]$ over the current spot market price. Once BidBrain constructs the set of possible allocations, it computes the cost per work for the current allocations and the cost per work for current allocations plus each of the possible allocations. If the cost per work for the best possible allocation is smaller than for the current allocations, BidBrain will send this allocation request to AWS. As described earlier, each allocation is made for the duration of the billing hour. This means that briefly before the end of an allocation's billing hour, BidBrain compares the cost per work if the allocation is renewed or terminated. If the cost per work is lower when the allocation is not renewed, BidBrain will terminate all the instances in the allocation prior to them reaching the next billing hour. In addition to spot resources, BidBrain acquires the required amount of on-demand resources (reliable instances in Fig 4.3). It does not consider terminating these resources even if they negatively affect cost-per-work.

4.3.3 Application Compatibility

BidBrain's design is compatible with applications beyond AgileML. It should work well for batch computations, where optimizing cost per unit work makes sense, that are able to efficiently add and remove large portions of their resource footprint quickly and efficiently. In future work, we plan to explore other optimization metrics to fit other elastic application types.

4.4 Proteus Implementation

This section describes how Proteus incorporates BidBrain and AgileML and how it connects to AWS. Figure 4.6 shows a high level overview. As described in Section 4.2, the user links an ML application to Proteus and specifies the location of the training data-set. The user is also responsible for providing the security credentials necessary to connect to AWS.

Upon start-up, Proteus connects AgileML to BidBrain via a ZMQ message that specifies the application characteristics (Sec. 4.3). Proteus then connects to AWS, gathers the current spot market price via *boto.ec2* API calls and loads the historic spot market data, both of which are directed into BidBrain. Using the information about the AWS spot market in combination with information about the characteristics of the ML application, BidBrain builds allocation requests (Sec. 4.3), which Proteus sends to AWS via the *boto.ec2* API. Upon receiving the allocation requests, AWS returns a set of spot request ids, which are translated by BidBrain to the assigned AWS EC2 instances. Once these instances

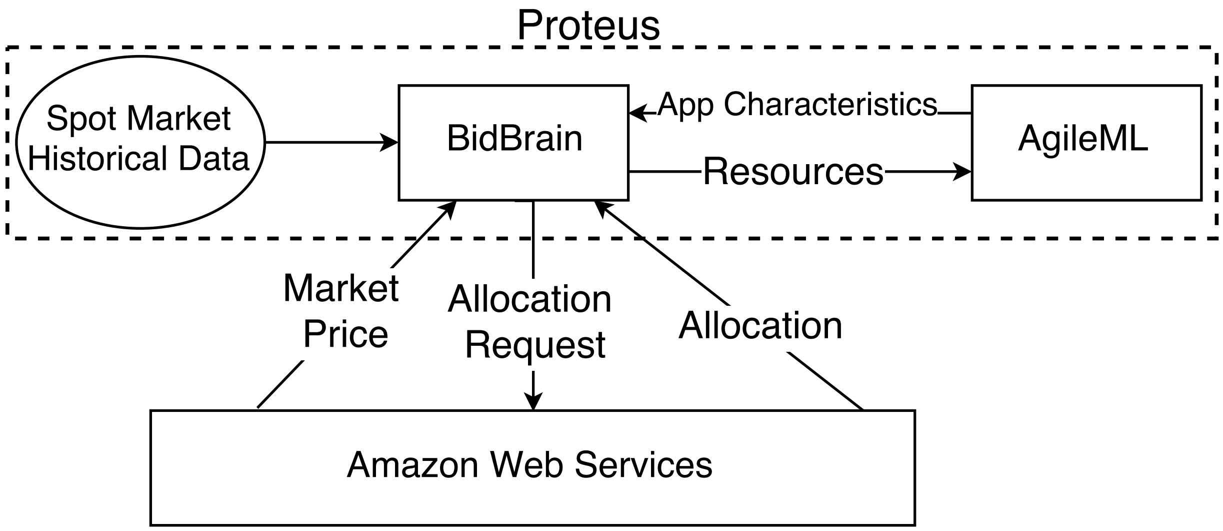


Figure 4.6: The Proteus architecture consists of the resource allocation component, BidBrain, and the elastic ML framework, AgileML.

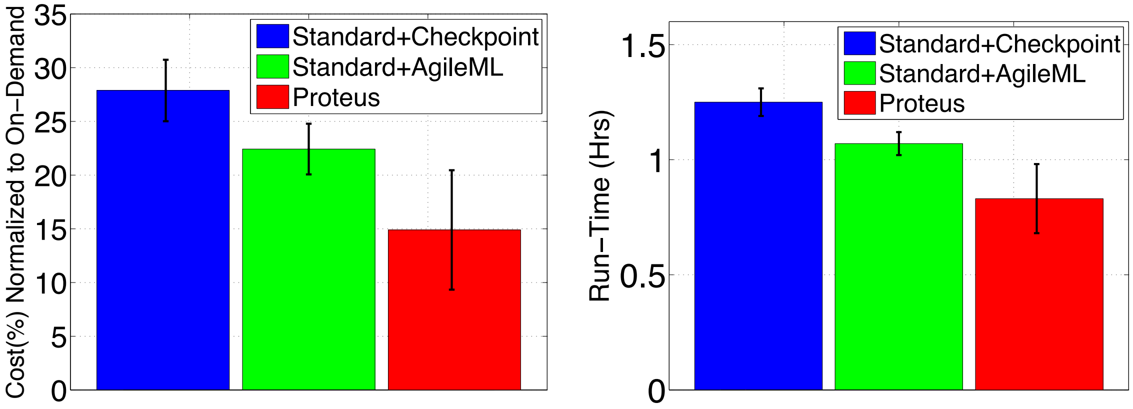
become reachable via SSH, BidBrain sends a ZMQ message to AgileML's elasticity controller containing the list of IP addresses and sizes of the instances in the new allocation.

BidBrain considers making new allocation requests to AWS every two minutes, briefly before the end of a billing hour of any current allocations, and immediately following an eviction. BidBrain monitors AWS for eviction notifications. Upon receiving an eviction notification, BidBrain translates it to the ids of the resources that are affected and notifies AgileML's elasticity controller. Proteus assumes that multiple ML applications are executed in sequence. Upon completing the final job in the queue, Proteus immediately terminates the on-demand resources. It then waits until the end of current billing hours to terminate the spot allocations, in hope that they are evicted by AWS prior to the end of the billing hour, lowering the overall cost.

In the current design, there is no redundancy for BidBrain or the elasticity controller. If either components fails, Proteus is still able to continue making progress. Either component can be restarted, if it fails, and synchronized with the ongoing computation.

4.5 Evaluation

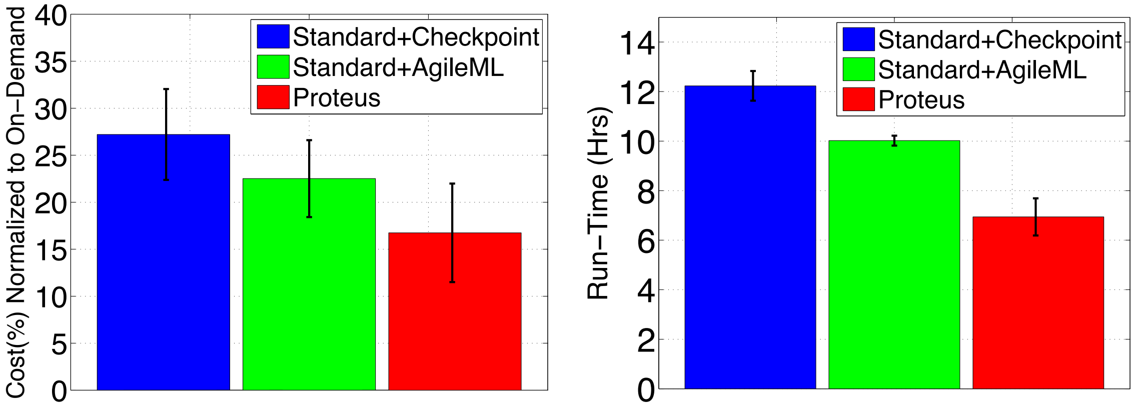
This section evaluates Proteus' effectiveness. The results support a number of findings, including: 1) In the context of AWS, Proteus' exploitation of spot market resources significantly reduces cost (e.g., by ≈85% compared to on-demand only) and outperforms standard bidding policy combined with a checkpointing-based elasticity in terms of both cost (by 42%–47%) and runtime (by 32%–43%); 2) Proteus' elasticity support introduces minimal overhead to a traditional non-elastic parameter-server configuration; 3) Proteus enacts bulk machine additions and revocations with minimal disruption, performing most



(a) Cost Savings

(b) Run-time

Figure 4.7: 2hr Job Duration.



(a) Cost Savings

(b) Run-time

Figure 4.8: 20hr Job Duration.

setup actions in the background.

4.5.1 Experimental Setup

Experimental Platforms. We report results for experiments on two virtual cluster configurations on AWS. **Cluster-A** is a cluster of 64 Amazon EC2 c4.2xlarge instances. Each instance has 8 vCPUs and 15 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM). **Cluster-B** is a cluster of 128 Amazon EC2 c4.xlarge instances. Each instance has 4 vCPUs and 7.5 GB memory, running 64-bit Ubuntu Server 14.04 LTS (HVM). From our testing using `iperf`, we observe a bandwidth of 1 Gbps between each pair of EC2 instances.

4.5.2 Cost Savings with Proteus

Proteus enables significant cost reductions on infrastructures that offer inexpensive transient machines. Fig. 4.1 summarizes the cost and time savings using BidBrain and AgileML for the *MLR* application. This section drills down further by evaluating Proteus' ability to reduce cost on EC2, relative to using only reliable on-demand machines, by analyzing the AWS Spot Market Traces from June 11, 2016 to August 14, 2016 for the US-EAST-1 region (all 4 zones).[7] We also compare the cost savings achieved by Proteus to those from existing approaches (see Section 4.7), which combine a checkpointing-based scheme for exploiting spot market machines with a standard spot market bidding strategy.

We perform cost savings analysis with long-term AWS traces, rather than experiments on EC2 for several reasons. Simulations on long-term AWS traces let us experiment with different approaches applied to the same market data, allowing for fair comparisons. Using AWS traces also allows us to gather data points across many months to get a full picture of expected behavior than our budget-limited experiments could otherwise provide. For each scheme and bidding model considered, we present the average cost (relative to full on-demand price) across 1000 randomly chosen day/time starting points in each zone. We perform experiments on durations with length of 2 and 20 hours, which is representative of long-running ML experiments (e.g., 4 hours for MLR) as well as the common practice of performing sequences of ML jobs for hyperparameter explorations.

We present cost results as average cost per job, so for accounting purposes we do not charge a given job for any minutes that remained in a job's final billing hours (e.g., if 20 minutes left, the job is charged for only 2/3 of the cost of the hour). We did this because the left over time is used by the following job in a sequence. This accounting was done the same way for all experiments, providing no benefit to Proteus.

Checkpointing-Based Scheme. As a comparison point for AgileML, we consider a scheme that tries to run entirely on spot market machines, using checkpointing to recover from evictions. We assume an MTTF-based checkpointing frequency, like that used in Flint [88]. We observe a resulting average checkpointing overhead of 17% for *MF* on both Cluster-A and Cluster-B (Sec. 4.5.2) when bidding the on-demand price, from the combined overheads of producing a consistent checkpoint state (recall that bounded staleness is allowed during ML application execution) and storing it. These overhead values are consistent with those reported by others [88].

Standard Bidding Strategy. As a comparison point for BidBrain, we consider an oft-used bidding strategy that selects the resource type with the lowest current market price and bids the on-demand price. It uses these resources until they are evicted, at which point it again selects the resources with the lowest current market price and bids the on-demand price. This is the default bidding policy used by Spot Fleet EC2, a service provided by AWS for users to acquire allocations of spot resources.

Cost Savings Results. Figure 4.7 and 4.8 show the cost savings and run-time for three different configurations for jobs of 2 hours and 20 hours, respectively, relative to running on 64 on-demand machines from Cluster-A: (1) the standard bidding strategy

[7] We used AWS Spot Market Traces from March 14, 2016 to Jun 10, 2016 to train the β parameter used in BidBrain.

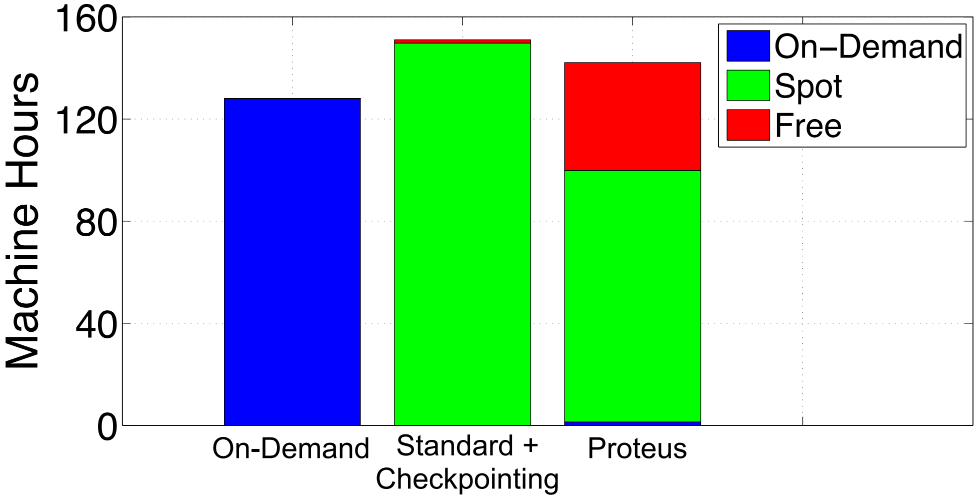


Figure 4.9: Breakdown of machine hours (for 2-hour jobs) among on-demand resources, spot resources (not evicted), and free resources (spot resources evicted prior to end of billing hour).

combined with the checkpointing-based scheme (blue). (2) the standard bidding strategy combined with AgileML, allowing evaluation of the incremental benefit of AgileML over the checkpointing-based scheme (green). (3) Proteus which combines BidBrain and AgileML (red). Comparing Proteus to the second configuration allows evaluation of the additional benefit of BidBrain over the standard bidding strategy.

The results demonstrate that Proteus significantly reduces both cost and run-times. On average, Proteus reduces cost by 83%–85% compared to traditional execution on on-demand machines and by 42%–47% compared to the state-of-the-art approach (Standard+Checkpoint). In addition to significantly lowering costs, Proteus also reduces run-times by 32%–43%. The results also show that each of BidBrain and AgileML contribute significantly to Proteus' overall cost and runtime improvements.

Attribution of Benefits. Proteus' superior performance arises from several factors. AgileML's ability to perform agile elasticity, ability to efficiently handle evictions, and lack of run-time overhead reduces the cost by 18%–20% compared to the checkpointing-based scheme (see blue and green bars in Figure 4.7 and 4.8). Similar benefits are seen when evaluating AgileML vs. the checkpointing-based scheme combined with BidBrain. The remaining improvements come from BidBrain's ability to effectively exploit the spot market. BidBrain reduces cost and run-time by providing opportunities for *free computing* and projecting how resource allocations impact work throughput.

Free computing occurs when an allocation produces useful work but is evicted by AWS before the end of a billing hour. The user receives a refund for the last partial hour, which means that any work produced by the allocation during the current billing hour has no cost to the user. Users increase their likelihood of getting *free computing* by bidding closer to the current spot market price, which increases the likelihood of evictions. When executing applications with significant eviction overheads, regularly bidding just above the current

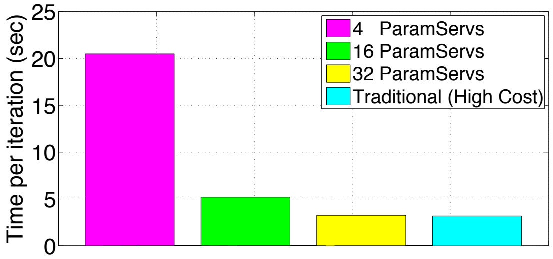


Figure 4.10: AgileML stage 1 with 4–32 reliable machines out of 64 total compared to traditional (all 64 reliable; cyan), for MF.

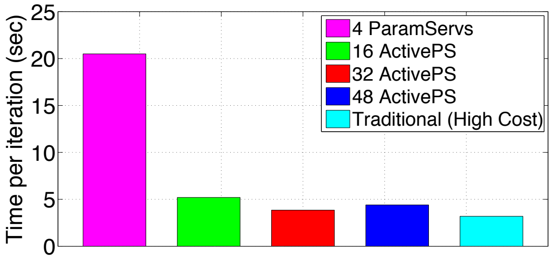


Figure 4.11: AgileML stage 2 with 4 reliable and 60 transient compared to stage 1 (same ratio; magenta) and traditional (64 reliable).

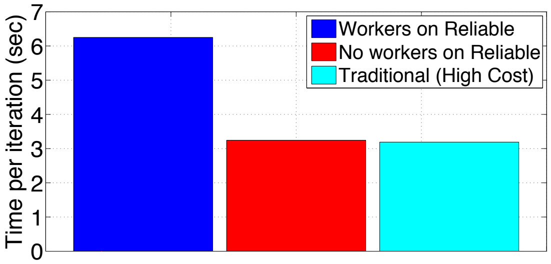


Figure 4.12: AgileML stage 3 (red) with 1 reliable and 63 transient compared to stage 2 (same ratio; blue) and traditional.

market price hoping to gain *free computing* is not an effective strategy. BidBrain accounts for eviction overheads in making decisions about how much above the market price to bid. We experimented with always bidding just above the market price to acquire *free computing* as often as possible, but it increased the run-time of applications (3-4x) and resulted higher cost due to suffering too many evictions after too short a period of time. BidBrain's predictions of eviction likelihood and times are effective enough to find a happy medium. On average, 32% of Proteus' computing is *free computing*, as shown in 4.9.

Experiments in the Wild. In addition to simulations, we ran a number of Proteus jobs on AWS. Although the results cover a much smaller sample size than our simulations, the observed behavior is consistent with the simulation results.

4.5.3 Efficiency with AgileML Tiering

AgileML enables execution on a mix of reliable and transient machines, and efficient scale-up and scale-down, while always maintaining state required for continued operation on reliable machines. To avoid the reliable machines becoming a bottleneck, AgileML uses three stages of functionality partitioning (see Section 4.2.2), decreasing reliance on reliable machines as the ratio of transient to reliable increases. (Of course, higher ratios are better from a cost standpoint, because transient machines are often 70–80% cheaper.) This section evaluates AgileML's performance relative to the traditional parameter-server architecture run entirely on high-cost reliable machines, in which all functionality (worker and parameter server) is partitioned among all machines, showing that AgileML avoids performance loss at least to a ratio of 63 transient machines to 1 reliable machine. All results in this section are for the *MF* application with the *Netflix* data set on Cluster-A, but results for the other applications and Cluster-B are consistent and omitted only due to space constraints.

Stage 1: Parameter Servers only on Reliable Machines. The first stage spreads the parameter server across the reliable machines, rather than all machines, using transient machines only for worker processes.

Figure 4.10 shows the time-per-iteration for different numbers of machines running parameter server shards (ParamServs) in a 64-machine Cluster-A, representing different ratios of transient to reliable machines under the stage 1 configuration. All 64 machines run workers. The 64 ParamServ case, which is labeled "Traditional" in the graph, represents the traditional parameter server architecture in which all machines are reliable and run both worker and parameter server processes. The results show that stage 1 has negligible slowdown for a small ratio (e.g., 1:1, represented by "32 ParamServ") of transient to reliable machines, but introduces significant slowdown as the ratio increases. The slowdown is caused by network bottlenecks caused by many workers communicating with a relatively smaller number of ParamServs.

Stage 2: ActivePSs on Transient Machines and BackupPSs on Reliable Machines. To avoid the network bottleneck for higher ratios, stage 2 switches to a tiered primary-backup model, using reliable machines for continuity but not requiring them to serve as active parameter servers for a much larger number of workers.

Figure 4.11 shows the time-per-iteration for different configurations in a 64-machine Cluster-A that consists of 4 reliable machines and 60 transient machines. The "4 ParamServs" and "Traditional" bars described above for Figure 4.10 are included as well, for comparison. The other three bars represent running ActivePSs on different numbers of transient machines, together with BackupPSs on the 4 reliable machines. In each case all 64 machines run worker processes. The results show that the ActivePS-based architecture with 32 ActivePSs introduces ≈18% slowdown compared to the traditional parameter-server architecture, when using a 15:1 ratio of transient to reliable machines. This slowdown does not occur at 7:1 and represents the beginning of the straggler problem addressed by stage 3.

Stage 3: No Workers on Reliable Machines. When the ratio of transient to reliable machines increases beyond 15:1, we observe even larger slowdowns for AgileML stage 2 relative to the traditional parameter-server architecture. This slowdown is caused by the workers running on reliable machines becoming stragglers; the network load of running BackupPSs for a much larger number of ActivePSs interferes with worker communication. To solve this problem, stage 3 does not run workers on the reliable machines when the ratio is very high. While this reduces aggregate worker computation power, stage 3 is only used when the reduction is small because the fraction of reliable machines is low.

4.12 shows time-per-iteration with and without workers on the one reliable machine in a 64-machine Cluster-A that consists of 1 reliable machine and 63 transient machines. The one reliable machine runs only a BackupPS. The "Traditional" bar is again shown

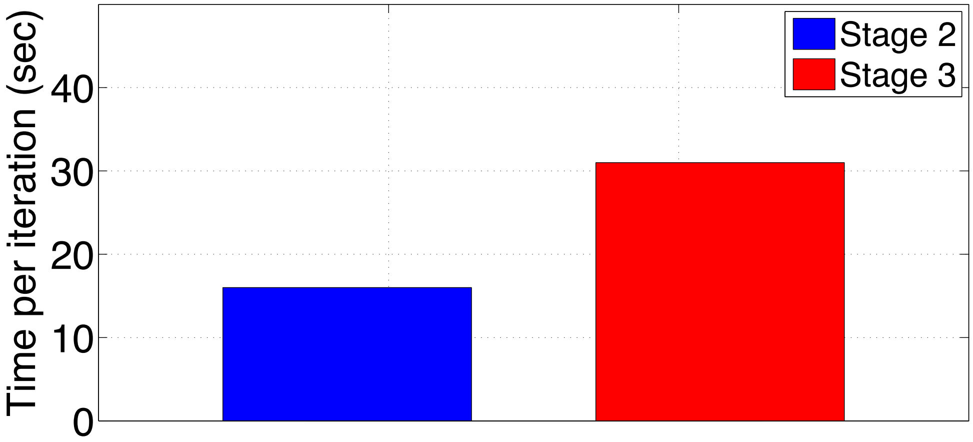


Figure 4.13: AgileML running on 8 reliable and 8 transient machines in stage 2 and stage 3 mode. Stage 2 is better for lower transient-to-reliable ratios.

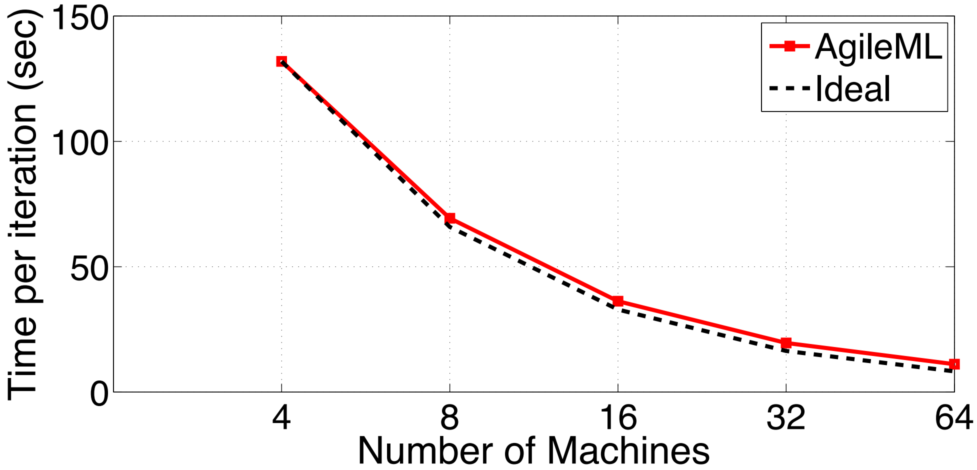


Figure 4.14: AgileML scalability for LDA. Showing time-per-iteration when using 4 to 64 machines.

for comparison. The results show that, by shutting down reliable machine workers once they become stragglers, AgileML is able to match the performance of the traditional parameter-server architecture at a 63:1 ratio of transient to reliable machines.

Stage 3 provides the best performance only as the ratio of transient vs. reliable machines increases. Thus, all three stages are needed for AgileML to achieve high performance across a range of possible ratios. To illustrate, 4.13 compares time-per-iteration attained with the same footprint (8 reliable + 8 transient machines), but in two different modalities: stage 2 and stage 3. Stage 2 is clearly best for this 1:1 ratio, unlike 4.12, where the ratio was 63:1.

4.5.4 AgileML Scalability

This section confirms that AgileML scales well as machines are added, like the traditional parameter-server architecture has been shown to do. Figure 4.14 shows time-per-iteration

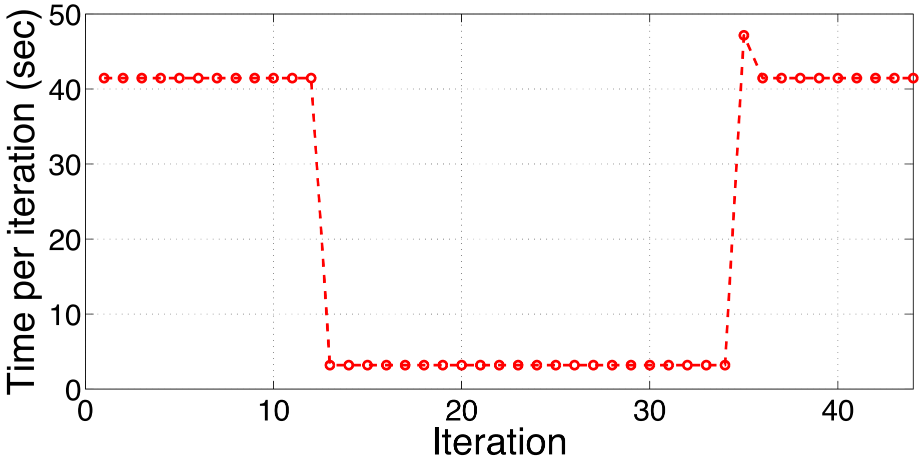


Figure 4.15: AgileML starts with 4 reliable resources, adds 60 transient resources at iteration 11, evicting 35 transient resources at iteration 35.

for the *LDA* application as a function of the number of Cluster-A machines used. (We observe the same scaling behavior for the other ML applications tested.) So, strong scaling is evaluated, and the curve labeled "Ideal" corresponds to perfect scaling of the 4-machine case. The 4-machine case uses the traditional parameter-server architecture to provide a baseline. The 8-machine case uses the stage 1 configuration for 4 reliable and 4 transient machines. The 16-, 32-, and 64-machine cases use the stage 3 configuration for 1 reliable machine and the remainder transient. These results show that AgileML scales effectively, exploiting available transient machines to speed up applications.

4.5.5 Efficiency of AgileML Elasticity

This section confirms that AgileML's mechanisms for bulk incorporation and eviction of machines induce minimal disruption of the ongoing ML application. Figure 4.15 shows time-per-iteration for each of 45 *MF* iterations on Cluster-A machines. The first 10 iterations execute on 4 reliable machines. 60 transient machines are incorporated during iteration 11, resulting in immediate speedup consistent with Figure 4.14. Adding the 60 machines causes no disruption because they are started, initialized, and prepared in the background, signaling the elasticity controller for final incorporation when ready. The opposite change is made in iteration 35, evicting the 60 transient machines from the computation, as though in reaction to an eviction notice. A 13% blip in performance is seen during the iteration in which the eviction is done, after which the time-per-iteration stabilizes, returning to its full 4-machine value. The blip occurs because of network overhead in aggressively bringing up-to-date the BackupPSs and transitioning them to being active ParamServs.

4.6 Discussion and Limitations

The popularity of systems like Proteus may increase spot market prices. We believe, however, that infrastructure clouds will continue to offer cheaper transient resources in order to monetize resources that would otherwise be idle due to varying demand on resources provisioned for contract customers. We also believe that AWS will eventually transition to a free market, where users are charged the *bid price* instead of the *market price.* This transition will render the commonly used strategy of bidding far above the market price obsolete, further motivating the need for intelligent bidding strategies, such as the one we instantiate with BidBrain.

BidBrain's allocation policies use AWS market price information to estimate resource reliability. However, BidBrain's allocation policies could be retargeted to be applicable beyond the AWS spot market. While this would eliminate the benefit received from free compute, only a portion of BidBrain's wins comes from such AWS specifics (Sec. 4.5.2). Instead of basing resource reliability predictions on historical AWS spot market data, BidBrain may perform reliability calculations by observing available resource capacity, its dynamics over time, and the activity of higher-priority jobs sharing the cluster. For example, in a private cluster setting, purchase cost (P_i) may be the same constant value for any best-effort allocation, but the expected work (4.3) still varies based on expected time to eviction (4.2).

Worker State. Proteus is designed to work with stateless AgileML workers, which reduces system complexity and obviates the need for I/O intensive state-management operations, such as checkpointing. Indeed, the elasticity controller only needs to coordinate easily reassigned state in ActivePSs and ParamServs. This design choice is further motivated by the fact that most ML training algorithms can be implemented with stateless workers.

Distributed All-Reduce. AgileML is designed to work with parameter server architecture. But, we believe it could be modified to work with distributed all-reduce (Sec. 2.2.2). In order to maintain high resources efficiency, this modification would require AgileML to use only stage three of its architecture (Sec. 4.2.2). In stage three, distributed all-reduce could be used to update model parameters on all transient machines, with the latest parameters then pushed to reliable machines in the background.

Communication Overhead. In a traditional parameter server configuration (Sec. 2.2.1), each machine co-locates ML workers and a parameter server shard responsible for $1/N$ of the model parameters (N is the number of machines). If every model parameter is read and updated by every machine during every iteration, each machine will communicate $4 * W * (N - 1)/N$, where W is the number of model parameters. In stage three of AgileML's architecture, half of the machines contain both ML workers and ActivePSs (responsible for $2/N$ of the model parameters), and the other half machine contain only workers. Machines with ActivePSs, communicate $4 * W * (N - 2)/N$, and machines with only workers communicate $2 * W * N$ per iteration. This means that for $N > 2$, AgileML reduces the amount of communication performed by each machine.

4.7 Related Work on Exploiting Transient Resources

Previous work exploits transient resources using checkpointing, DHTs, RDDs, and heuristic bidding strategies.

Checkpointing. Checkpointing can be used to preserve state when using transient resources [47, 88, 92]. For example, a non-elastic computation can be started on EC2 spot market machines and checkpointed regularly. If the machines are revoked, the computation can be restarted on another set of machines from the last completed checkpoint. Gupta et al. [47] propose this approach for scientific computations. Parameter server architectures such as TensorFlow [8], MxNet [22], Petuum [105], LazyTables [28], and IterStore [29] provide no explicit mechanism for exploiting transient resources, and hence would likewise rely on checkpointing. A single machine failure causes most of these systems to restart an ongoing computation from the most recent checkpoint.[8] Although this is reasonable in small-to-medium clusters under traditional failure models, it can incur high overheads in elastic settings due to the frequency of revocations (e.g., all the spikes in Figure 4.2). Moreover, dynamically adding machines to running ML applications is not supported by these frameworks. To do so would seem to require stopping the computation in a consistent state, adding the resources, adjusting the mapping of computation tasks to machines and copying any needed state accordingly, and then restarting. (Section 4.2.3 describes AgileML's alternative, efficient approach.) We hope this work will motivate other ML frameworks to become agile elastic, and when they do, we believe they will integrate well with BidBrain and provide a great comparison for AgileML. In our experimental study, we compare Proteus' explicit elasticity support to this checkpointing-based approach.

Distributed Hash Tables (DHTs). The parameter server system described by Li et al. [67] includes support for adding and removing machines during execution. To realize this feature, the system uses a direct-mapped DHT design based on consistent hashing, wherein each parameter server process is responsible for a particular key range, and parameter value replication. Protocols for adding and removing machines are described. While DHTs are effective for adding or removing resources one-at-a-time, we believe that Proteus' approach to elasticity is better suited to the *bulk* addition and removal of nodes that characterize the transient resource availability discussed above. Li et al. did not evaluate the speed of node set changes, but we expect that it would be insufficient to address revocation of a sizable subset of cheap machines within the limited warning period provided. The replication mechanism also would not solve this issue, because bulk revocation is akin to *correlated* failure of many nodes, while the mechanism is designed for independent failures.

Spark and RDDs. Spark achieves fault tolerance with RDDs, storing deterministic transformations for subsequent replay on recovery from checkpoint. Flint [88], concurrently with our work, proposed a system for running Spark applications on transient machines. Unlike our tiered approach leveraging a mix of transient and non-transient machines simultaneously, Flint runs ML workloads entirely on transient nodes,[9] like the checkpointing approach above. RDDs reduce the cost of checkpointing/recovery for Spark applications

[8] Tensorflow has a mechanism for handling single machine failures via its straggler mitigation mechanism.

[9] or entirely on non-transient nodes in the rare cases when they are cheaper

by selectively choosing the set of RDDs needed. Whereas Flint relies heavily on the Spark's computing model in exploiting transient machines, AgileML enables exploitation of such resources for parameter server systems, which are different and much more efficient for iterative convergent ML. Furthermore, Proteus' aggressive allocation strategy on the spot market provides significant savings, including 32% *free computing* on average. In contrast, Flint only considers switching when current resources are revoked and only bids the on-demand price, corresponding to Standard+AgileML in our graphs.

Bidding Strategies. Bidding strategies for EC2 spot instances have been studied [10, 110, 93]. Agmon et al. [10] show minimal correlation between near term AWS spot prices and spot instance availability. Marathe et al. [72] propose using redundancy *across* AWS zones for HPC computations on EC2. For interactive workloads, Flint [88] seeks to diversify across zones and machine classes to minimize revocation probability. Spot Fleet EC2, an AWS service, allows users to specify a resource capacity target and automatically maintain that target, replacing evicted instances. It is application agnostic, however, and does not take into account any application-level concerns, such as maximizing performance per unit cost. By default, Spot Fleet follows the same configured strategy as the standard bidding policy, bidding the on-demand price on the currently cheapest available resource (Sec. 4.5.2). We show significant improvements over such a bidding strategy. Tributary [48] proposes using many diversified sets on unreliable resources to achieve reliability for jobs requiring high reliability (e.g., jobs with SLOs).

4.8 Summary

This chapter presents Proteus, which efficiently utilizes transient resources for machine learning training through tiers of reliability. Proteus combines agile elastic machine learning training system with an intelligent resource manager. Our experiments show that both components are required to achieve maximum improvements of ≈85% compared to using only on-demand machines.

Chapter 5

Generalized Pipeline Parallelism for DNN Training

5.1 Introduction

Deep Neural Networks (DNNs) are being applied to great effect across a wide range of applications, including image classification [65, 89, 52], translation [104], language modeling [73] and video captioning [100]. As DNNs have become more widely developed and used, they have also become more computationally expensive to train, thus requiring parallel execution across multiple accelerators (e.g., GPUs).

DNN training proceeds in iterations of forward and backward pass computations, with each iteration processing a *minibatch* of input data at a time, and the model parameters updated at the end of each iteration. Current approaches focus on parallelizing each iteration of the optimization algorithm, one at a time, across a set of workers. For example, data parallelism partitions the input data across workers [65], model parallelism partitions operators across workers [33, 24], and hybrid schemes partition both [60, 61, 64]. Unfortunately, these *intra-batch* parallelization schemes suffer from poor scaling due to high communication costs and low resource utilization. For example, Figure 5.1 shows the communication overhead for data parallelism across five different popular DNN models on state-of-the-art `p3.16xlarge` instances on AWS, each with eight V100 GPUs connected over `NVLink`. Over 16 GPUs, the communication overhead for some models, computed as the percentage of total time spent on communication, is as much as 90% due to expensive `all_reduce` communication. Moreover, rapid increases in GPU compute capacity over time will further shift the bottleneck of training towards communication for *all* models.

In this chapter, we propose a new form of parallelism, *pipeline parallelism*, which enables faster DNN training by combining intra-batch parallelism with *inter-batch* parallelization. Pipeline parallelism divides the model among available workers, assigning a group of adjacent layers (*stages*) to each of them, and then overlaps the computation and communication of *different* minibatches in a pipelined fashion. Inter-worker communication is greatly reduced because it can be limited to layer inputs and outputs (activations in the forward pass and gradients in the backward pass) solely across adjacent layers assigned to *different* workers.

While pipelining is a simple idea, DNN training poses an important requirement not present in traditional pipelining: DNN training is bi-directional—the forward pass is followed by a backward pass through the same layers in *reverse* order, using state and intermediate results from the forward pass. To keep the pipeline full and thus achieve high hardware efficiency, a simple scheduling mechanism is injecting *all* minibatches into the pipeline, first completing forward passes for all input minibatches followed by backward passes. But, such an approach has low statistical efficiency [27], increasing the number of iterations needed to produce a high-quality model, or even preventing the model from reaching the desired target accuracy, since gradients are averaged over all training samples [20]. To improve statistical efficiency, one could inject only a subset of m minibatches into the pipeline, and apply weight updates every m minibatches. This would, however, result in reduced hardware efficiency due to frequent pipeline flushes; traditional model parallel training corresponds to an extreme case of this ($m = 1$). Thus, a more nuanced approach to pipelining is needed – one that achieves both high hardware efficiency *and* statistical efficiency.

Given a pipeline of stages, our system, PipeDream, proposes a scheduling algorithm called 1F1B to keep hardware well utilized while achieving high statistical efficiency in a bi-directional pipeline. In 1F1B's steady state, each worker strictly alternates between forward and backward passes for its stage, ensuring high resource utilization (negligible pipeline stalls) even in the common case where the backward pass takes longer than the forward pass, and uses *different* versions of model weights for good statistical efficiency. Each backward pass at a stage results in weight updates; the next forward pass uses the *latest* version of weights available, and "stashes" a copy of these weights to use during the corresponding backward pass. Although the forward pass will not see updates from incomplete in-flight mini-batches, learning is effective because model weights change relatively slowly and bounded staleness has been shown to not affect statistical efficiency [28]. However, for the backward pass to compute *numerically correct* gradients, the same weight version used during the forward pass must be used. PipeDream limits the number of "in-pipeline" minibatches to the minimum needed to keep the pipeline full, keeping the number of stashed weights to a minimum.

Operating the pipeline at peak throughput also requires that it be balanced. Because the throughput of a pipeline is bottlenecked by the slowest stage, PipeDream automatically determines how to partition the layers of the DNN based on a short profiling run (on a single GPU), balancing computational load among the different stages while minimizing communication for the target platform. PipeDream effectively load balances even in the presence of model diversity (computation and communication) and platform diversity (interconnect topologies and hierarchical bandwidths). As DNNs do not always divide evenly among available workers, PipeDream may decide to use data parallelism for some stages—multiple workers can be assigned to a given stage, processing *different* minibatches in parallel. PipeDream extends 1F1B to incorporate round-robin scheduling across data-parallel stages, while making sure that gradients in a backward pass are routed to the corresponding worker from the forward pass since the same weight version and intermediate outputs need to be used for a correct gradient computation. The combined scheduling algorithm, 1F1B-RR produces a static schedule of operators that each worker runs repeatedly, keeping utilization high across all workers. Thus, pipeline-parallel training can be thought

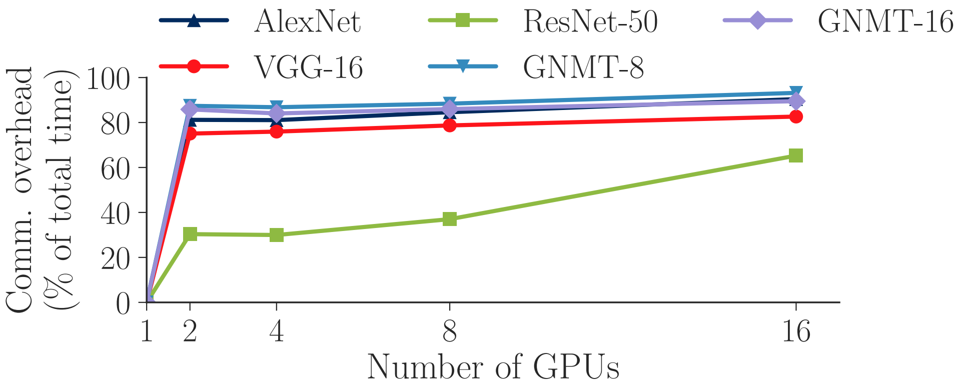


Figure 5.1: Communication overhead of data-parallel training (on one or two `p3.16xlarge` AWS instances each with eight `NVLink`-connected V100 GPUs) for popular image classification and machine translation DNN models, using PyTorch 1.0 and Gloo.

of as a principled combination of inter-batch pipelining with intra-batch parallelism.

Our evaluation with many combinations of DNN models, datasets, and hardware configurations confirms the training time benefits of PipeDream's pipeline parallelism. Compared to data-parallel training, PipeDream reaches target accuracy on multi-GPU machines up to 4.78× faster for image classification tasks, up to 3.45× faster for machine translation tasks, 4.25× faster for language modeling tasks, and 3× faster for video captioning models. PipeDream is also 2.5× – 18× faster than model parallelism, up to 1.9× faster than hybrid parallelism, and 1.7× faster than simpler approaches to pipelining.

Contributions. PipeDream is the first system to implement pipelining efficiently for DNN training. This chapter makes three primary contributions. First, it introduces the use of inter-batch parallelism via pipeline-parallel training. Second, it identifies the key challenges in realizing the potential of this idea and details how PipeDream addresses each, through its load balancing optimizer, 1F1B-RR scheduler, and weight stashing mechanism. Third, it experimentally demonstrates that PipeDream enables fast parallel DNN training even in circumstances where communication overheads cripple traditional intra-batch parallelization approaches.

5.2 Background and Related Work

This section covers two broad classes of parallel DNN training: intra- and inter-batch. It also highlights parallelization challenges posed by DNN model and hardware diversity.

5.2.1 Intra-batch Parallelism

The most common way to train DNN models today is intra-batch parallelization, where a single iteration of training is split across available workers.

Data Parallelism. In data parallelism, inputs are partitioned across workers. Each worker maintains a local copy of the model weights and trains on its own partition of inputs while periodically synchronizing weights with other workers, using either collective communication primitives like `all_reduce` [45] or parameter servers [67]. The amount of data communicated is proportional to the number of model weights and the number of workers participating in training.

The frequency of weight synchronization affects both statistical efficiency (number of iterations needed to reach a particular target accuracy) and hardware efficiency (time needed per iteration). Synchronizing at the end of every minibatch, referred to as *bulk synchronous parallel* or BSP [97][1] reduces the staleness of weights used to compute gradients, ensuring good statistical efficiency. However, BSP requires each worker to *wait* for gradients from other workers, lowering hardware efficiency. Despite optimizations such as Wait-free Backpropagation [109], where weight gradients are sent as soon as they are available (common in modern frameworks), communication stalls are inevitable due to the structure of the DNN computation, and because communication time often dominates computation time.

Figure 5.1 quantitatively shows the fraction of training time spent in communication stalls with data parallelism for different classes of DNNs on 2, 4, 8, and 16 NVIDIA V100 GPUs using 8-GPU instances with `NVLink` interconnects within servers and 25Gbps interconnects across servers (one instance for 2,4,8 and two instances for 16).

We focus on three takeaways. First, the communication overhead for many of these models is high despite using multi-GPU servers and state-of-the-art communication libraries like Gloo. Data parallelism comparatively scales well for models like ResNet-50, which have a large number of convolutional layers with compact weight representations, but scales less well for other models with LSTM or fully-connected layers, which have more dense weight representations. Second, applications distributed across multi-GPU servers are bottlenecked by slower inter-server links, even though faster intra-server links help. Data parallelism for such hierarchical networks is a poor fit, since the same number of bytes are sent over both high- and low- bandwidth channels. Third, as the number of data-parallel workers increases, communication overheads increase for all models, even if training is performed on a multi-GPU instance with `NVLink`. Coleman et al. [26] show similar results.

Other DP Optimizations. Asynchronous parallel training (ASP) allows each worker to proceed with the next input minibatch before receiving the gradients from the previous minibatch. This approach improves hardware efficiency over BSP by overlapping computation with communication, but also introduces staleness and reduces the statistical efficiency of DNN training. Our experimental results corroborate recent findings that show that these techniques do not reduce end-to-end DNN training time [30, 8].

Seide et al. [84, 83] looked at quantizing gradients to decrease the amount of data needed to be communicated over the network. This approximation strategy lacks generality and is effective for limited scenarios; it does not hurt convergence for some speech models [85], but hurts statistical performance due to noisy gradients in many others [30, 8]. Goyal et al. [45]

[1] In this paper, we use DP to refer to data-parallelism with BSP.

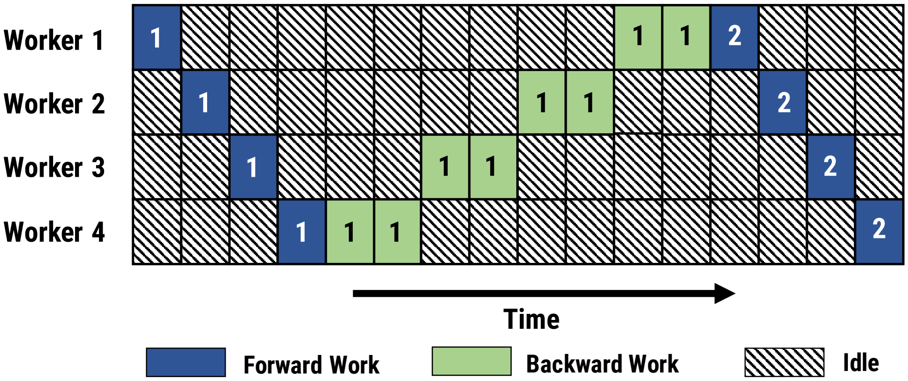


Figure 5.2: Model parallel training with 4 workers. Numbers indicate minibatch ID, and backward work takes twice as long as forwards work. For simplicity, here we assume that communicating activations/gradients across workers has no overhead.

used more efficient implementations of `all_reduce`, like the *recursive halving-and-doubling algorithm* and the *bucket algorithm* [94] to reduce the amount of data being sent over the network. Others have explored techniques from the HPC literature to reduce the overhead of communication [16, 96]. All these use synchronous *collection* communication patterns with no inter-batch pipelining, and hence only slightly alleviate communication bottlenecks.

Recent work has demonstrated that using large minibatches is effective for training ResNet-50, especially when combined with Layer-wise Adaptive Rate Scaling (LARS) [45, 106, 58]. Large minibatches reduce the communication overhead by exchanging parameters less frequently. But, our experiments show that such techniques lack generality beyond ResNet-50 and pipeline parallelism is still faster than the fastest data-parallel option with LARS.

Model Parallelism. Model parallelism is an intra-batch parallelism approach where the operators in a DNN model are partitioned across the available workers. That is, each worker evaluates and performs updates for only a subset of the model's parameters for all inputs. The amount of data communicated is the size of intermediate outputs (and corresponding gradients) that need to be sent across workers.

Although model parallelism enables training of very large models, traditional model parallelism is *rarely* used to accelerate DNN training because it suffers from two major limitations. First, model-parallel training results in under-utilization of compute resources, as illustrated in Figure 5.2. Each worker is responsible for a group of adjacent layers; in this regime, the intermediate outputs (activations and gradients) between these groups are the only data that need to be communicated across workers.[2] For each input, only a single stage is active at any instant of time. Pipelining multiple minibatches would improve utilization, but is traditionally not done because 1) the bi-directionality of DNNs makes pipelining challenging (the forward pass is followed by a backward pass through the same layers in

[2] While other partitioning schemes are possible, this is the most common, and the one we will use in this paper.

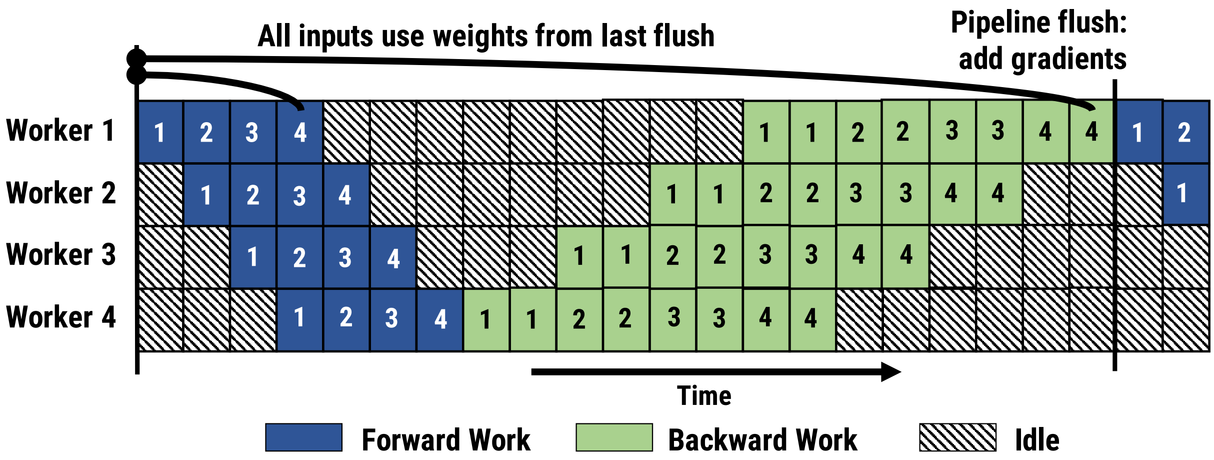

All inputs use weights from last flush
Pipeline flush: add gradients
Worker 1
Worker 2
Worker 3
Worker 4
Time
Forward Work
Backward Work
Idle

Figure 5.3: GPipe's inter-batch parallelism approach. Frequent "pipeline flushes" lead to increased idle time.

reverse order), and 2) naive pipelining performs weight update computations with *stale* weights, leading to the final model achieving a lower accuracy compared to data-parallel training.

The second limitation for model-parallel training is that the burden of partitioning a model across multiple GPUs is left to the programmer [64], resulting in point solutions. Recent work explores the use of reinforcement learning to automatically determine device placement for model parallelism [75]. Unfortunately, such techniques are time- and resource-intensive. They also don't seamlessly combine intra- and inter-batch parallelism, thus limiting their effectiveness.

Hybrid Intra-batch Parallelism. Recent work has proposed splitting a single iteration of the optimization algorithm among *multiple* dimensions. OWT [64] splits the then-popular AlexNet model by hand, using data parallelism for convolutional layers that have a small number of weight parameters but large outputs, while choosing to not replicate fully connected layers that have a large number of weight parameters but small outputs. OWT does not use pipelining. FlexFlow [60] proposed splitting a single iteration along samples, operators, attributes, and parameters, and describes an algorithm to determine how to perform this splitting in an automated way. However, FlexFlow does not perform pipelining, and we show in our experiments (Sec. 5.5.3) that this leaves as much as 90% of performance on the table.

5.2.2 Inter-batch Parallelism

Chen et al. [23] briefly explored the potential benefits of pipelining minibatches in model-parallel training, but do not address the conditions for good statistical efficiency, scale, and generality as applicable to large real-world models. Huo et al. [56] explored parallelizing the backwards pass during training. Our proposed solution parallelizes both the forward and backward pass.

GPipe [55], concurrent work with ours [50], uses pipelining in the context of simple model-parallel training for very large models. GPipe performs forward passes followed by backward passes for every m minibatches, aggregating weight gradients along the way; it also does not store intermediate state generated during the forward pass needed for the backward pass, instead opting to re-compute them (see Figure 5.3, where $m = 4$). As a result, it suffers from reduced hardware efficiency due to re-computation overheads and frequent pipeline flushes every m minibatches (Sec. 5.5.4).

In comparison, PipeDream addresses key issues ignored in prior work, offering a general solution that combines pipelining with intra-batch parallelism in a principled way, and automates partitioning the model across workers.

5.2.3 DNN Model and Hardware Diversity

DNN models are increasingly diverse, with convolutional layers, LSTMs [104], attention layers [98], and fully-connected layers commonly used. These different types of models exhibit vastly different characteristics; this makes the optimal parallelization strategy highly model-dependent.

Picking an optimal parallelization scheme is made harder by the fact that the efficacy of such a scheme depends on the characteristics of the target deployment hardware as well. GPUs, ASICs, and FPGAs have very different compute capabilities. Moreover, interconnects linking these accelerators have different topologies and capacities; cloud servers are linked by tens to 100Gbps networks, accelerators within servers might be connected over shared PCIe trees (10 to 15GBps), and specialized expensive servers, such as the DGX-1 [36], use `NVLink` with point-to-point 30GBps bandwidth capabilities. This diversity in models and deployments makes it extremely hard to *manually* come up with an optimal parallelization strategy. PipeDream automates this process.

5.3 Pipeline Parallelism

PipeDream uses *pipeline parallelism* (PP), a new parallelization strategy that combines intra-batch parallelism with inter-batch parallelism. Pipeline-parallel computation involves partitioning the layers of a DNN model into multiple *stages*, where each stage consists of a *consecutive* set of layers in the model. Each stage is mapped to a separate GPU that performs the forward pass (and backward pass) for all layers in that stage.[3]

In the simplest case, only one minibatch is active in the system, as in traditional model-parallel training (Figure 5.2); in this setup, at most one GPU is active at a time. Ideally, we would like all GPUs to be active. With this in mind, we inject multiple minibatches into the pipeline one after the other. On completing its forward pass for a minibatch, each stage asynchronously sends the output activations to the next stage, while simultaneously

[3] We use GPUs as a concrete instance of accelerators and use the terms “GPU” and “worker” interchangeably.

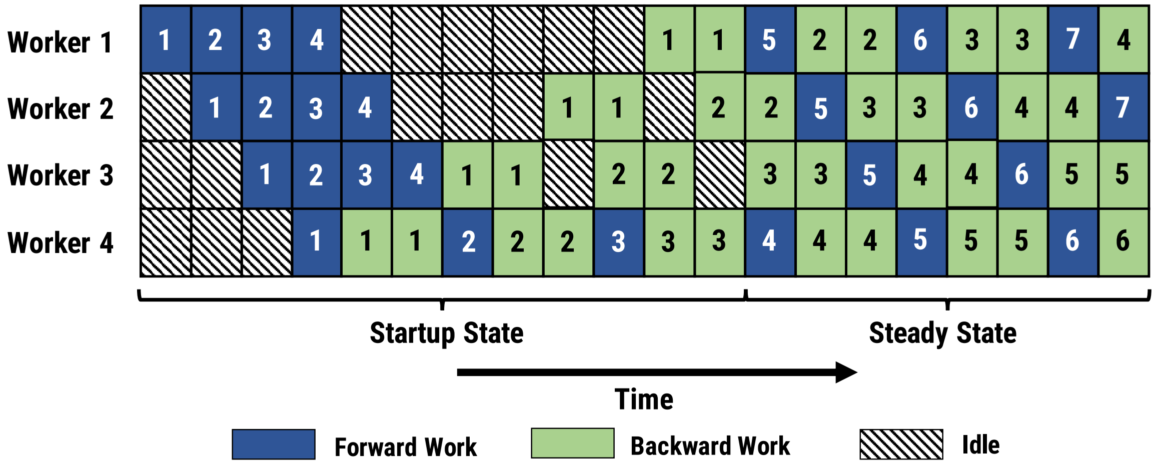


Figure 5.4: An example PipeDream pipeline with 4 workers, showing startup and steady states. In this example, the backward pass for a minibatch takes twice as long as the forward pass.

starting to process another minibatch. When the last stage completes its forward pass, it starts the backward pass on the same minibatch. On completing its backward pass, each stage asynchronously sends the gradient to the previous stage while starting computation for the next minibatch (Fig. 5.4).

Pipeline parallelism can outperform intra-batch parallelism methods for two reasons.

Pipelining communicates less. PP often can communicate far less than DP. Instead of having to aggregate the updates for all the parameters and send the result to all the workers, as is done in data-parallel approaches (using either collective communication or a parameter server), each worker in a PP execution has to communicate only subsets of the updates and output activations, often to only a single other worker. This can result in large reductions in communication (e.g., >90% reduction for VGG-16).

Pipelining overlaps computation and communication. Asynchronous communication of forward activations and backward gradients across stages results in significant overlap of communication with the computation of a subsequent minibatch as shown in Figure 5.5.

However, to realize the opportunity of PP, PipeDream must overcome three challenges. In discussing PipeDream's solutions to the challenges, we will refer to Figure 5.6, which shows PipeDream's high-level workflow.

5.3.1 Challenge 1: Work Partitioning

PipeDream treats model training as a computation pipeline, with each worker executing a subset of the model as a stage. Like with any pipeline, the steady state throughput of the resulting pipeline is the throughput of the slowest stage. Having each stage process minibatches at vastly differently throughputs can lead to bubbles in the pipeline, starving

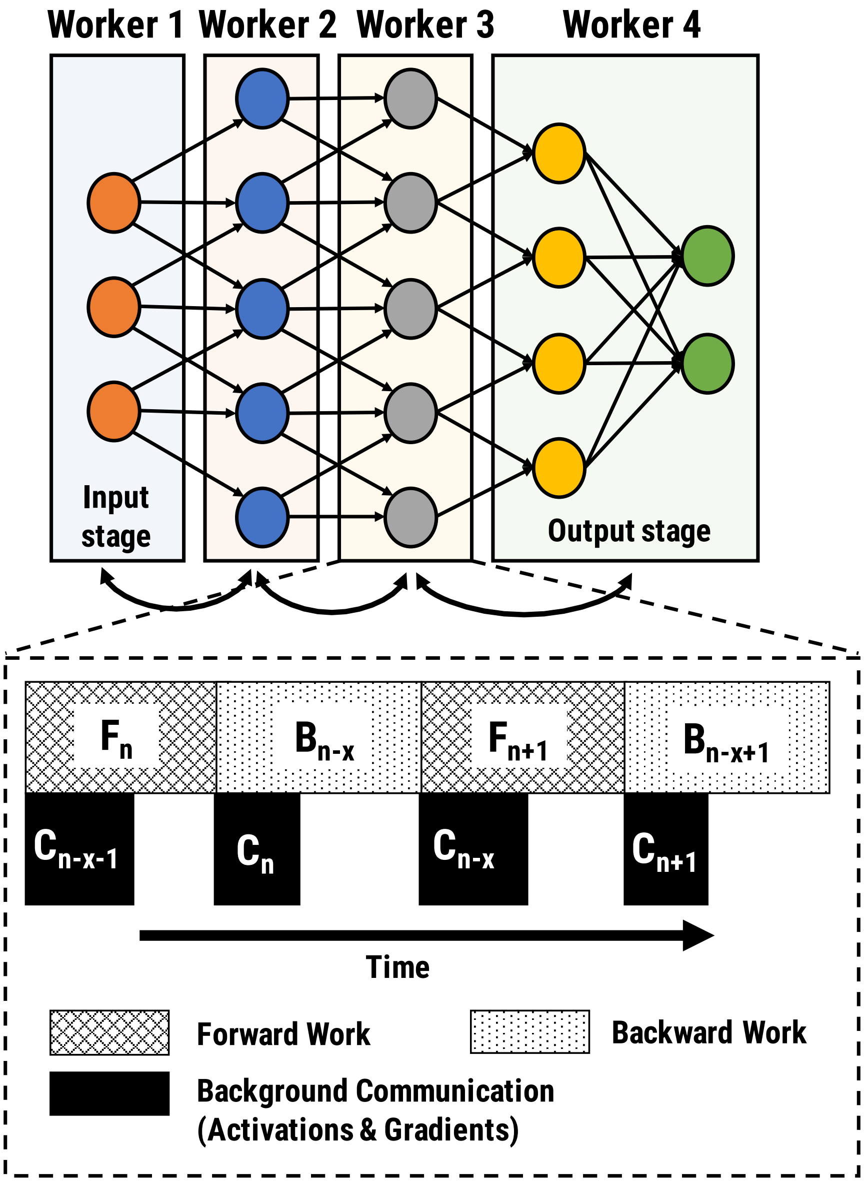


Figure 5.5: An example pipeline-parallel assignment with four GPUs and an example timeline at one of the GPUs (worker 3), highlighting the temporal overlap of computation and activation / gradient communication.

faster stages of minibatches to work on, leading to resource under-utilization. Excessive communication between workers can also lower the throughput of the training pipeline. Moreover, the allocation of stages to workers needs to be model- and hardware-aware to be effective, and there may be cases for which no simple partitioning across the GPUs achieves both limited communication and load balance.

Solution: PipeDream's optimizer determines a balanced pipeline. Its algorithm partitions DNN layers into stages, such that each stage completes at roughly the same rate, while trying to minimize communication across workers in a topology-aware way (for example, large outputs should be sent over higher bandwidth links if possible). To further improve load balancing, PipeDream goes beyond straight pipelines, allowing a stage to be replicated (i.e., data parallelism is used on the stage). This partitioning problem is equivalent to minimizing the time taken by the slowest stage of the pipeline, and has the *optimal sub-problem property*: a pipeline that maximizes throughput given a worker count is composed of sub-pipelines that maximize throughput for smaller worker counts. Consequently, we use dynamic programming to find the optimal solution.

PipeDream exploits the fact that DNN training shows little variance in computation and communication time across inputs, especially epochs. PipeDream records the computation time taken by the forward and backward pass, the size of the layer outputs, and the size of the associated parameters for each layer as part of an initial profiling step; this profile is used as the input to the optimizer's partitioning algorithm (Figure 5.6). The partitioning algorithm also takes into account other constraints such as hardware topology and bandwidth, number of workers, and memory capacity of the compute devices.

Profiler. PipeDream records three quantities for each layer l, using a short (few minutes) profiling run of 1000 minibatches on a single GPU: 1) T_l, the total computation time across the forward and backward pass for the layer on the target GPU, 2) a_l, the size of the output activations of the layer (and the size of input gradients in the backward pass) in bytes, and

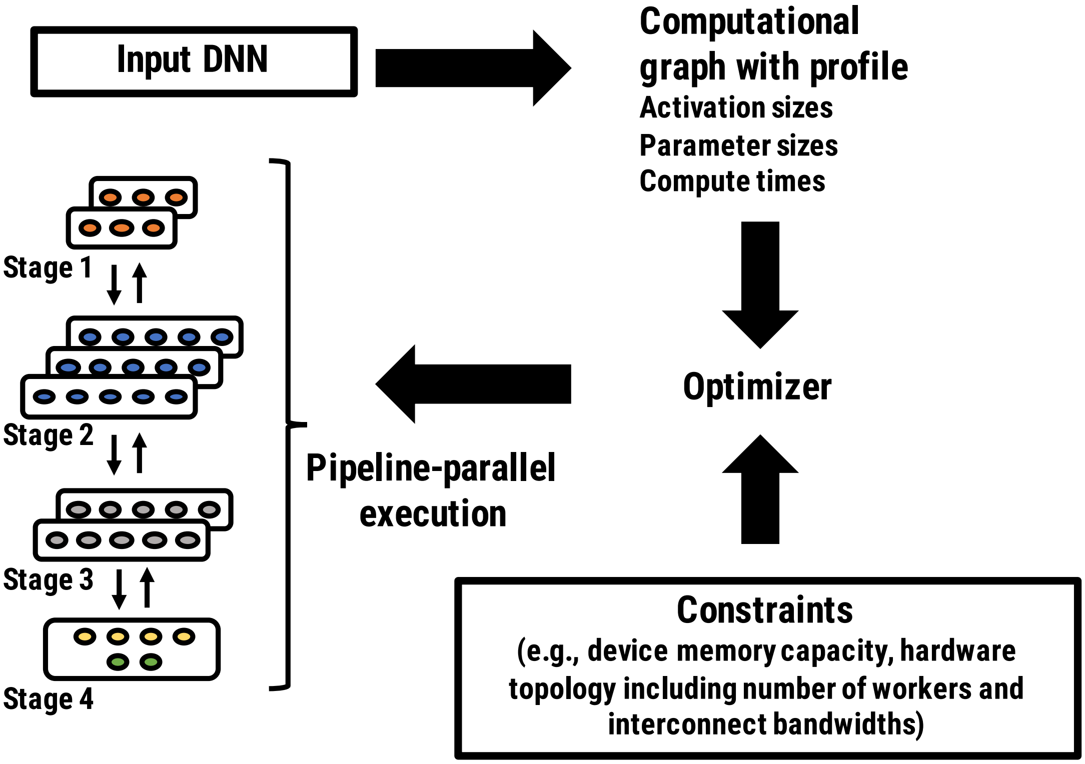

Input DNN
Computational graph with profile
Activation sizes
Parameter sizes
Compute times
Optimizer
Constraints
(e.g., device memory capacity, hardware topology including number of workers and interconnect bandwidths)
Pipeline-parallel execution
Stage 1
Stage 2
Stage 3
Stage 4

Figure 5.6: PipeDream Overview: profiler, optimizer, and runtime.

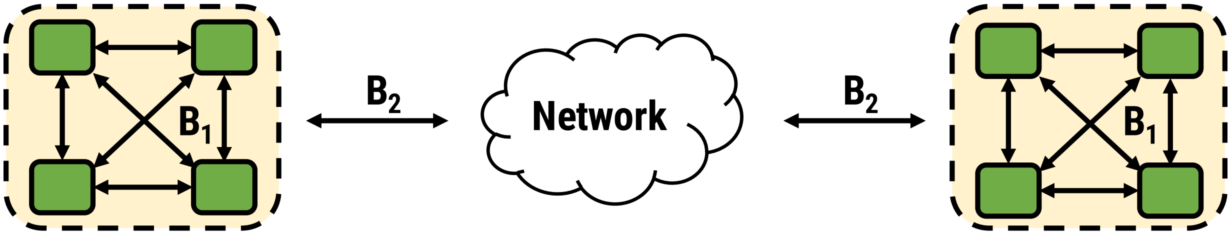


Figure 5.7: An example 2-level hardware topology. Green boxes represent GPUs. Each server (yellow boxes) has 4 GPUs connected internally by links of bandwidth B_1; each server is connected by links of bandwidth B_2. In real systems, $B_1 > B_2$.

3) w_l, the size of weight parameters for layer l in bytes.

PipeDream estimates the communication time by dividing the amount of data that needs to be transferred by the network bandwidth of the communication link. The amount of data communicated in data-parallel configurations with m workers is $2 \cdot (m-1) \cdot |w_l|$ (assuming collective communication); this is used to estimate the time for weight synchronization for layer l when using data parallelism with m workers.

Partitioning Algorithm. Our partitioning algorithm takes the output of the profiling step, and computes: 1) a partitioning of layers into stages, 2) the replication factor (number of workers) for each stage, and 3) optimal number of in-flight minibatches to keep the training pipeline busy.

PipeDream's optimizer assumes that the machine topology can be organized into levels, as shown in Figure 5.7. Bandwidths within a level are the same, while bandwidths across levels are different. We assume that level k is comprised of m_k level $(k-1)$ components, connected by links of bandwidth B_k. In Figure 5.7, $m_2 = 2$ and $m_1 = 4$. In addition, we define m_0 to be 1; m_0 represents the number of compute devices within the first level (green boxes in Figure 5.7).

PipeDream's optimizer solves dynamic programming problems progressively from the lowest to the highest level. Intuitively, this can be thought of as trying to find the optimal partitioning within a server, and then using these partitions to split a model optimally across servers.

Notation. Let $A^k(i, j, m)$ denote the time taken by the slowest stage in the optimal pipeline between layers i and j using m workers at level k. The goal of our algorithm is to find $A^L(0, N, m_L)$, and the corresponding partitioning, where L is the highest level and N is the total number of layers in the model.

Let $T^k(i \to j, m)$ denote the time taken by a single stage spanning layers i through j, replicated over m workers using bandwidth B_k.

Formulation. For all k from 1 to L,

$$T^k(i \to j, m) = \frac{1}{m} \max \left(A^{k-1}(i, j, m_{k-1}), 2(m-1) \sum_{l=i}^{j} \frac{|w_l|}{B_k} \right)$$

where the left term inside the max is the total computation time for all the layers in the stage using level $k-1$ as the computation substrate, and the right term is the time for data-parallel communication among all layers in the stage.

The optimal pipeline can be broken into an optimal sub-pipeline consisting of layers from 1 through s with $m-m'$ workers followed by a single stage with layers $s+1$ through j replicated over m' workers. Then, using the optimal sub-problem property, we have:

$$A^k(i,j,m) = \min_{i \le s < j} \min_{1 \le m' < m} \max \begin{cases} A^k(i,s,m-m') \\ 2a_s/B_k \\ T^k(s+1 \to j, m') \end{cases}$$

where the first term inside the max is the time taken by the slowest stage of the optimal sub-pipeline between layers i and s with $m-m'$ workers, the second term is the time taken to communicate the activations and gradients of size a_s between layers s and $s+1$, and the third term is the time taken by the single stage containing layers $s+1$ to j in a data-parallel configuration of m' workers.

Initialization. Level 0 uses the profiled computation times: $A^0(i,j,m_0) = \sum_{l=i}^{j} T_l$. For $k > 0$, optimal compute times with all compute devices in the previous level are used: $A^k(i,j,1) = A^{k-1}(i,j,m_{k-1})$.

Runtime Analysis. For a given level k, the total number of sub-problems is $O(N^2 m_k)$. Time complexity per sub-problem is $O(N m_k)$, leading to a total time complexity of $O(N^3 m_k^2)$ for level k. Total time complexity is $\sum_{k=1}^{L} O(N^3 m_k^2)$. In our experiments, the running time is under 8 seconds.

5.3.2 Challenge 2: Work Scheduling

Unlike traditional uni-directional pipelines, training in PipeDream involves a bi-directional pipeline, where a minibatch proceeds through the computation pipeline first forward and then backward. Consequently, each active minibatch in the pipeline may be in a different stage, either in the forward pass or backward pass. As a result, each worker in the system needs to determine whether it should i) perform its stage's forward pass for a minibatch, pushing the minibatch to downstream workers, or ii) perform its stage's backward pass for a different minibatch, pushing the minibatch to upstream workers. In addition, how should minibatches be routed with replicated stages?

Solution: In the startup phase, the input stage admits enough minibatches to keep the pipeline full in steady state (e.g., see Figure 5.4). Based on the partitioning generated by our algorithm, the optimal number of minibatches admitted *per input stage replica* to keep the pipeline full in steady state is given by:

NUM_OPT_ACTIVE_MINIBATCHES ($NOAM$) =

$$\lceil \text{(\# workers) / (\# of replicas of the input stage)} \rceil.$$

Once in steady state, each stage *alternates* between performing its forward pass for a minibatch and its backward pass for an earlier minibatch. We call this the *one-forward-one-backward* (1F1B) schedule. 1F1B ensures that every GPU is occupied with a minibatch in a balanced pipeline, because each stage is producing outputs in aggregate at roughly the same rate, and that backward passes from inputs are applied at regular intervals of time (Fig. 5.4).

When a stage is run in a data-parallel configuration (replicated across multiple GPUs), we use deterministic round-robin load balancing based on a minibatch identifier to spread work across the replicas. Such deterministic load-balancing ensures that each minibatch is routed to the same worker for both the forward and backward passes of the stage, which is important since parameters and intermediate outputs from the forward pass are needed for the backward pass. This mechanism, which we call *one-forward-one-backward-round-robin* (1F1B-RR), is a *static* policy that is executed without expensive distributed coordination.

Figure 5.4 shows the corresponding compute timeline for a pipeline with 4 stages. The NOAM for this configuration is 4. In the startup phase, the input stage admits exactly four minibatches that propagate their way to the output stage. As soon as the output stage completes its forward pass for the first minibatch, it performs its backward pass for the same minibatch, and then starts alternating between forward and backward passes for subsequent minibatches. As the backward pass starts propagating to earlier stages in the pipeline, every stage starts alternating between its forward and backward passes for different minibatches. As shown in the figure, in steady state, every worker is busy doing either its forward or backward pass for some minibatch. For 1F1B-RR to be effective, it is *not necessary* for the forward pass to take as long as the backward pass. In fact, we observe that in practice, the backward pass is always larger than the forward pass, and 1F1B-RR remains an effective scheduling mechanism, as highlighted in Figure 5.4.[4]

5.3.3 Challenge 3: Effective Learning

In a naively pipelined system, each stage's forward pass for a minibatch is performed using one version of parameters and its backward pass is performed using a different version of parameters. Figure 5.4 illustrates this using a partitioning with four workers and no stage replication. In stage 1, the forward pass for minibatch 5 is performed after the updates from minibatch 1 are applied, whereas the backward pass for minibatch 5 is performed after updates from minibatches 2, 3, and 4 are applied. As a result, in the backward pass for minibatch 5 on stage 1, the gradient is computed using a different set of weights than the ones used in the corresponding forward pass; this discrepancy in weight versions results in invalid gradients, and prevents model convergence. Experimental results show that naive pipelining is *unable* to achieve the same accuracy as data-parallel training.

Solution: To avoid a fundamental mismatch between the version of weights used in the forward and backward pass, PipeDream uses a technique we call *weight stashing* which maintains multiple versions of the weights, one for each active minibatch. When performing its forward pass, each stage processes a minibatch using the latest version of weights available. After completing the forward pass, PipeDream stores the weights used for that minibatch. When performing the minibatch's backward pass, the *same version* of the weights is used to compute the weight update and upstream weight gradient.

[4] 1F1B-RR produces a full steady state pipeline even for cases where the ratio of backward- to forward-pass time is not an integer (e.g., 3 to 2).

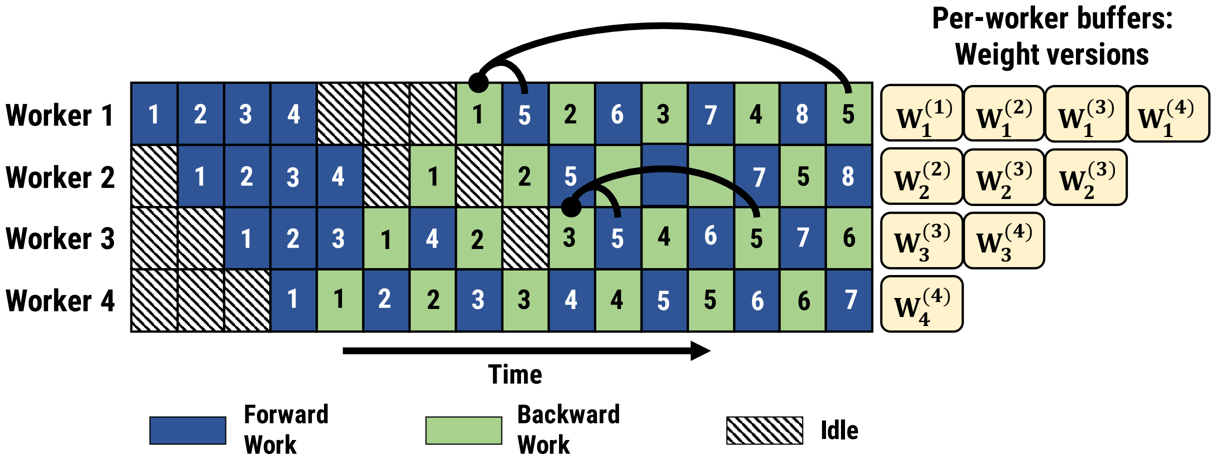


Figure 5.8: Weight stashing as minibatch 5 flows across stages. Arrows point to weight versions used for forward and backward passes for minibatch 5 at the first and third stages.

Weight stashing ensures that *within a stage*, the same version of model parameters are used for the forward and backward pass of a given minibatch. For example, in Figure 5.8, minibatch 5 uses parameter updates from minibatch 1 on machine 1 and from 2 on machine 2. Weight stashing says nothing about the consistency of parameter versions used for a given minibatch *across* stages.

Vertical Sync. Vertical Sync is an optional technique in PipeDream that eliminates the potential inconsistency *across stages*. For example, in Figure 5.4, using vertical sync, minibatch 5 uses parameters updated by minibatch 1 on all workers for both its forward and backward passes. Each minibatch (b_i) that enters the pipeline is associated with the latest weight version ($w^{(i-x)}$) seen at the input stage. This information is propagated along with the activations and gradients as the minibatch b_i flows through the pipeline in the forward direction. Across all stages, the forward pass for b_i uses the stashed weights $w^{(i-x)}$, as opposed to the latest weight update. After performing the backward pass for b_i (using stashed weights $w^{(i-x)}$), each stage independently applies weight updates to create the latest weights ($w^{(i)}$), and can then delete $w^{(i-x)}$. This coordination across stages is asynchronous.

Staleness. We can now formalize the degree of staleness of weight updates for each of these techniques. For this discussion, we assume a straight pipeline (i.e., no stage replication) with the model split into n stages; the weights in each stage are represented as w_1, w_2, and so on. In addition, we denote $w_l^{(t)}$ as the weights w_l after t minibatches.

Now, after every minibatch, we compute the gradient $\nabla f(w_1, w_2, \ldots, w_n)$ averaged over all samples in the minibatch. Vanilla minibatch SGD (f is the loss function, ν is the learning rate) has the following gradient update:

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f(w_1^{(t)}, w_2^{(t)}, \ldots, w_n^{(t)})$$

With weight stashing, gradients in stage 1 are computed with weights that are n steps delayed, gradients for stage 2 are computed with weights that are $n-1$ steps delayed, etc. Mathematically, this means the weight update looks like:

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f(w_1^{(t-n+1)}, w_2^{(t-n+2)}, \ldots, w_n^{(t)})$$

Without weight stashing, the weight update is not a valid gradient of the loss function f for any vector $w_1, \ldots, w_n$.

Adding vertical sync alters the weight update to:

$$w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f(w_1^{(t-n+1)}, w_2^{(t-n+1)}, \ldots, w_n^{(t-n+1)})$$

This is semantically similar to data parallelism with BSP synchronization on n workers (with the same per-worker minibatch size), with the same staleness (but gradients averaged over a minibatch size B instead of nB).

Memory Overhead. Pipelining does not significantly increase per-worker memory usage relative to data-parallel, even with weight stashing. Consider a straight pipeline (no data-parallel stages) where a model is divided across n workers, with each worker holding $1/n$ of the weights. With non-pipelined model-parallel training, each worker would need $1/n$ of the memory compared to data parallel training. Admitting n inputs into the pipeline, as PipeDream does, increases this by at most a factor of n, because a version of the (weights, activations) is needed for each in-flight minibatch. Thus, PipeDream's per-worker memory usage is on par with data parallelism. PipeDream's memory footprint can be further reduced by efficient memory management of intermediate data [57].

Weight stashing is critical for meaningful learning.[5] PipeDream's default semantics (weight stashing but no vertical sync) are between regular minibatched SGD on a single worker, and data parallelism with BSP synchronization [28, 54]. Our evaluation demonstrates its effectiveness across models, datasets, and hardware configurations.

5.4 Implementation

Figure 5.6 shows PipeDream's high-level workflow. PipeDream first profiles the model on a single GPU with a subset of inputs from the training dataset. It then runs the optimization algorithm described in Section 5.3.1 to partition the DNN model into stages, with some stages possibly replicated. The PipeDream runtime then assigns each stage (including replicas for replicated stages) to a single worker, according to its 1F1B-RR schedule.

The interface to PipeDream is implemented as a Python library that manages device memory, schedules work, and handles communication. The current implementation is integrated with PyTorch [6]. However, PipeDream is extensible and can work with other ML frameworks such as Tensorflow [8], MXNet [22], and CNTK [83]. As a proof of concept, we also integrated PipeDream with Caffe [59].

[5] In our experiments, we find that the impact of *vertical sync* is negligible. PipeDream's default semantics exclude vertical sync as it requires more metadata to be stored at every stage in the pipeline.

Parameter State. For each stage, PipeDream maintains all parameters associated with the layers assigned to the stage directly in GPU memory. If the stage is not replicated, PipeDream applies the updates to the most recent parameter version when the weight update becomes available. If the stage is replicated, the weight updates are synchronized across replicas prior to being applied. When a newer version of the parameters becomes available, the prior version is *not* immediately discarded. Parameters are discarded only once a backward pass that uses fresher parameters is performed.

Intermediate State. Each stage's input and output data is assigned a unique blob ID. Upon receiving intermediate data from the prior stage (or from disk in the case of the input stage), PipeDream copies the intermediate data to GPU memory and places a pointer to the associated buffer in a work queue. Intermediate data from the forward pass is not discarded until the associated minibatch completes that stage's backward pass. Intermediate data from the backward pass is freed as soon as the worker finishes using it, and if necessary, after it is sent to the next stage.

Stage Replication. PipeDream uses PyTorch's `Distributed-DataParallel` library [7] to synchronize parameters for layers of data-parallel stages. Using wait-free back propagation, weight gradients are communicated to servers as soon as they are computed, rather than waiting for computation to finish for all layers. Because we support replication of individual stages, data-parallel training can be thought of as a special case in our framework – we represent this as a single stage that contains all the layers of the DNN model, and replicate the stage across all available GPUs.

All inter-GPU communication in PipeDream, both in data-parallel and pipeline-parallel settings, uses Gloo [5].

Checkpointing. PipeDream supports periodic checkpointing of model parameters for fault-tolerance, with default checkpointing across stages at the end of every epoch. Checkpoints don't require expensive global coordination; each stage dumps its model parameters locally when it performs the backward pass for the last minibatch in an epoch. Restarting a failed training run due to a failure entails starting from the last epoch successfully checkpointed by all the stages.

5.5 Evaluation

This section evaluates the effectiveness of PipeDream for seven different DNNs on three different clusters. The results of our experiments support a number of important findings: 1) PipeDream achieves significant speed-ups in time-to-target-accuracy across a wide range of different learning tasks on different hardware deployments, 2) PipeDream is more efficient than other recently proposed inter-batch approaches, 3) PipeDream greatly reduces overheads of communication and does not significantly increase memory footprint compared to data-parallel training, and 4) combining pipelining, model parallelism, and

Task	Model	# Servers × # GPUs per server (Cluster)	PipeDream Config	PipeDream speedup over DP
Image Classification	VGG-16 [89]	1x4 (A)	3-1	2.14×
		4x4 (A)	12-1	4.78×
		2x8 (B)	15-1	4.33×
	ResNet-50 [52]	4x4 (A)	8	1×
		2x8 (B)	8	1×
	AlexNet [65]	1x4 (A)	3-1	3.4×
		4x4 (A)	12-1	4.62×
		2x8 (B)	15-1	2.70×
Translation	GNMT-16 [104]	1x4 (A)	Straight	3.45×
		4x4 (A)	Straight	2.91×
		2x8 (B)	Straight	2.62×
	GNMT-8 [104]	1x4 (A)	Straight	3.26×
		4x4 (A)	Straight	2.67×
		2x8 (B)	Straight	2.14×
Language Modeling	AWD LM [73]	1x4 (A)	Straight	4.25×
Video Captioning	S2VT [100]	4x1 (C)	2-1-1	3.01×

Table 5.1: Summary of results comparing PipeDream with data parallelism (DP) when training models to advertised final accuracy. A PipeDream config of "2-1-1" means the model is split into three stages with the first stage replicated across 2 workers, and a "straight" configuration is a pipeline with no replicated stages—e.g., "1-1-1-1" on 4 workers. Batch sizes used to train these models are reported in Sec. 5.5.1.

Cluster name	Server SKU	GPUs per server	Interconnects Intra-, Inter-server
Cluster-A	Azure NC24 v3	4x V100	PCIe, 10 Gbps
Cluster-B	AWS p3.16xlarge	8x V100	NVLink, 25 Gbps
Cluster-C	Private Cluster	1 Titan X	N/A, 40 Gbps

Table 5.2: Characteristics of servers used for evaluation.

data parallelism performs significantly better than using model-, data-, or hybrid-parallelism in isolation.

5.5.1 Experimental Setup

Clusters. We use three different clusters in our experiments, summarized in Table 5.2. *Cluster-A* has servers with 4 NVIDIA V100 GPUs each (Microsoft Azure NCv3 instances), with 16 GB of GPU device memory, and a 10 Gbps Ethernet interface. *Cluster-B* has servers with 8 V100s each (AWS EC2 p3.16xlarge instances), with 16 GB of GPU device memory, and a 25 Gbps Ethernet interface. GPUs within servers are connected via a shared PCIe interconnect on Cluster-A, and via point-to-point `NVLink` on Cluster-B. All servers run 64-bit Ubuntu 16.04 with CUDA toolkit 10.0 and cuDNN v7.4. *Cluster-C* has servers with 1 NVIDIA Titan X GPU and 12 GB of GPU device memory, connected via 40 Gbps Ethernet. Unless otherwise stated, all our experiments are run on multi-GPU servers (Cluster-A and Cluster-B).

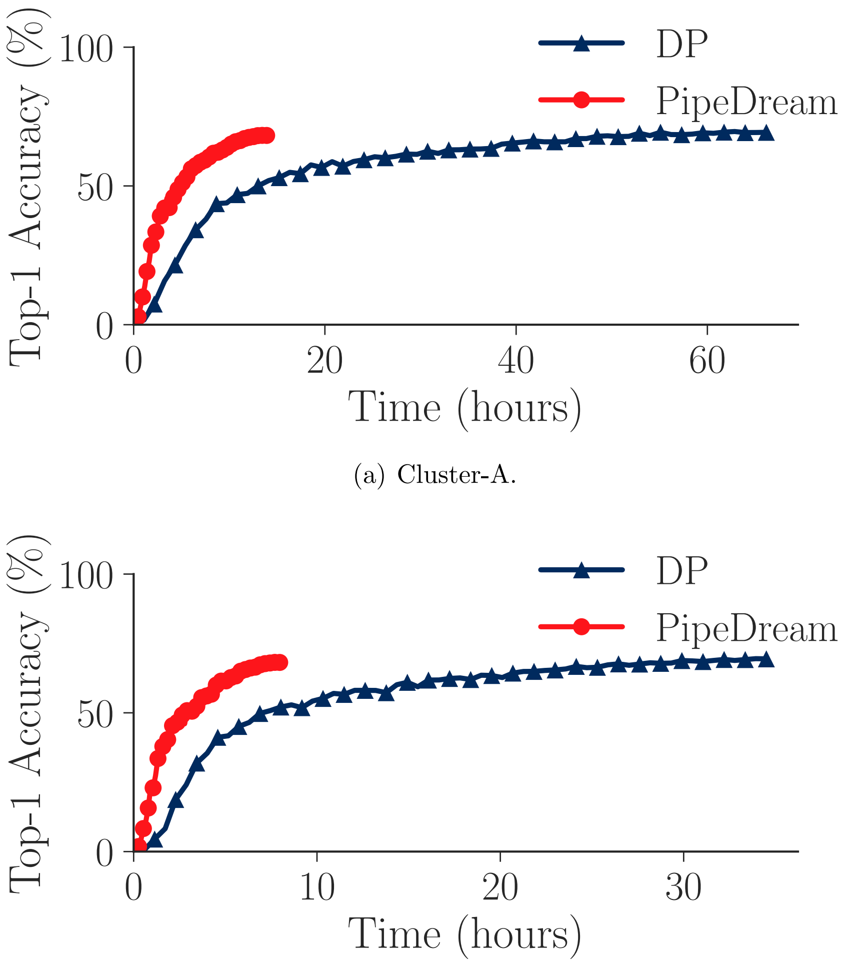
Batch Sizes and Training Methodology. We use the largest per-GPU minibatch that fits in one GPU's memory – anything larger yields out-of-memory exceptions. This ensures that we hit peak achievable FLOPs on a single device. Unless otherwise stated, we report per-GPU minibatch sizes (G); for data-parallel runs with n workers, the global minibatch size (BS) is $n \times G$. The global minibatch sizes we use are consistent with those used by the ML community and reported in the literature for these models. We use a per-GPU minibatch size of 64 per GPU for VGG-16, 256 for AlexNet, 128 for ResNet-50 (e.g., $BS = 1024$ for 8 GPUs), 64 for GNMT, 80 for S2VT, and batch size of 80 for LM. We train the VGG-16, ResNet-50, Language Modeling, and S2VT models using SGD with an initial learning rate of 0.01, 0.1, 30.0, and 0.01 respectively. For GNMT, we use the Adam optimizer [62] with an initial learning rate of 0.0003.

For all experiments (other than AlexNet), we measure the time taken to train to a target accuracy: top-1 accuracy of 68% for VGG-16, top-1 accuracy of 75.9% for ResNet-50, BLEU score of 21.8 for GNMT, a validation perplexity of 98 for LM, and a METEOR [35] score of 0.294 for S2VT. Guided by prior work, we adjust the learning rate during training to converge to the desired result faster [90, 62] and adjust initial learning rate and utilize learning rate warm-up for large global batch sizes [45]. For AlexNet, we use synthetic data (otherwise, data loading is the bottleneck) and measure throughput.

5.5.2 Comparison to Data Parallelism

Table 5.1 summarizes results comparing PipeDream with data-parallel training (DP). The table shows PipeDream's auto-generated configurations and their speedups in training time-to-accuracy over corresponding data-parallel training configurations.

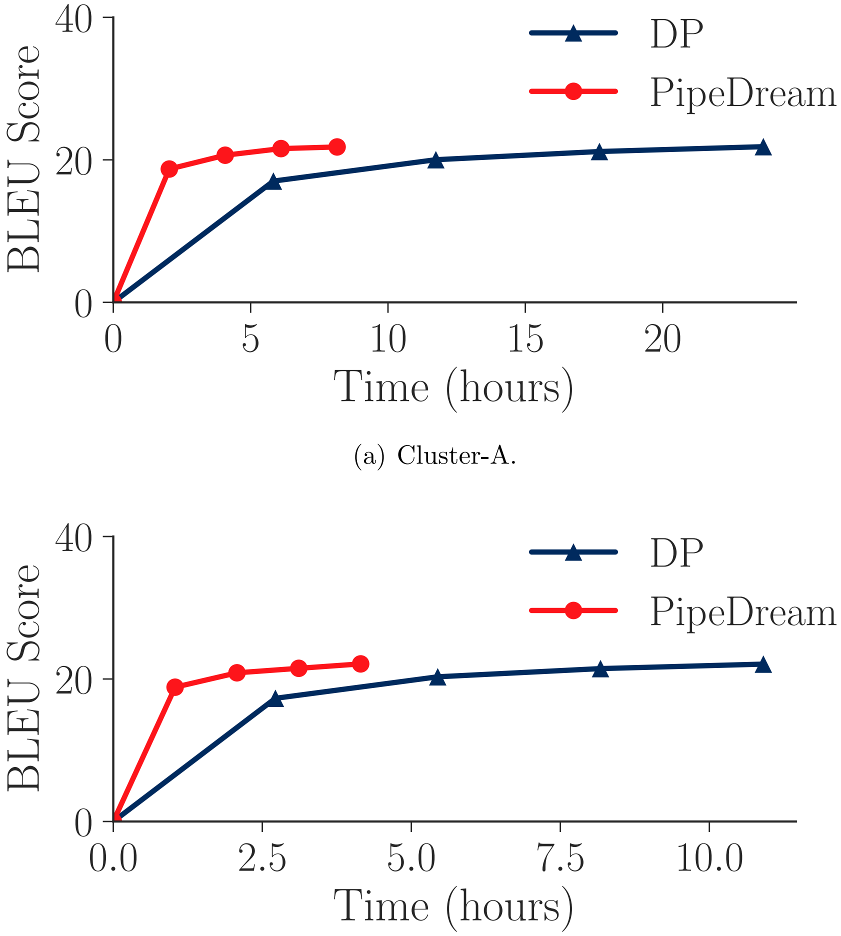
PipeDream Configurations. As described in Section 5.3.1, given a DNN model and a set of servers, PipeDream's optimizer automatically chooses to partition the model into stages, while also deciding the optimal replication factor for each stage. Although most prior research has focused on improving data-parallel training, our results indicate that the best configurations for many models is not data parallelism despite the use of many important optimizations such as wait-free back propagation. In all but one of our experiments, the best PipeDream configuration combines model parallelism, pipelining, and sometimes data parallelism; each of these configurations significantly outperform data-parallel training, thus highlighting the importance of combining inter-batch pipelining with intra-batch parallelism. PipeDream's optimizer recommends data parallelism for ResNet-50 because its weight representations are small and its layer outputs are large.



(a) Cluster-A.

(b) Cluster-B (Note x-axis difference from Cluster-A).

Figure 5.9: Accuracy vs. time for VGG-16 using 16 GPUs. Each circle or triangle represents two epochs of training.



(a) Cluster-A.

(b) Cluster-B (Note x-axis difference from Cluster-A).

Figure 5.10: Accuracy vs. time for GNMT-16 using 16 GPUs. Each circle or triangle represents an epoch of training.

Image Classification. We compare PipeDream and DP time-to-accuracy for VGG-16 using 1 and 4 servers in Cluster-A (1x4 (A) and 4x4 (A) in Table 5.1). PipeDream reaches target accuracy 2.14× faster than DP on a single server. PipeDream reduces communication by 95% compared to DP. Inter-server communication is slower than GPU-to-GPU intra-server communication, thus the reduction in communication overhead leads to an even larger speedup for PipeDream compared to DP as we scale to 4 servers (16 GPUs) – PipeDream reaches target accuracy 4.78× faster than DP. Figure 5.9 (a) shows this comparison as the DNN is trained over time. In the 4-server configuration, PipeDream's optimizer (Sec. 5.3.1) recommends a 12-1 configuration, achieving the above speedup with 3 fewer workers than DP! PipeDream makes this choice because the layers within the non-replicated stage contain a large number of parameters, making a 12-4 configuration ineffective.

Compared to Cluster-A, which has 4 GPUs per server connected via PCIe, Cluster-B has 8 GPUs per server connected over faster peer-to-peer NVLink interconnects. On 2

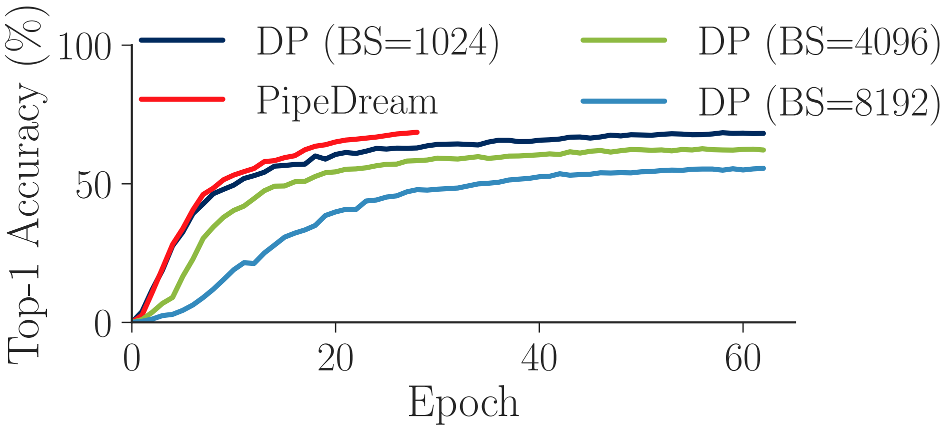


Figure 5.11: Statistical efficiency (accuracy vs. epoch) using LARS (VGG-16, 8 GPUs).

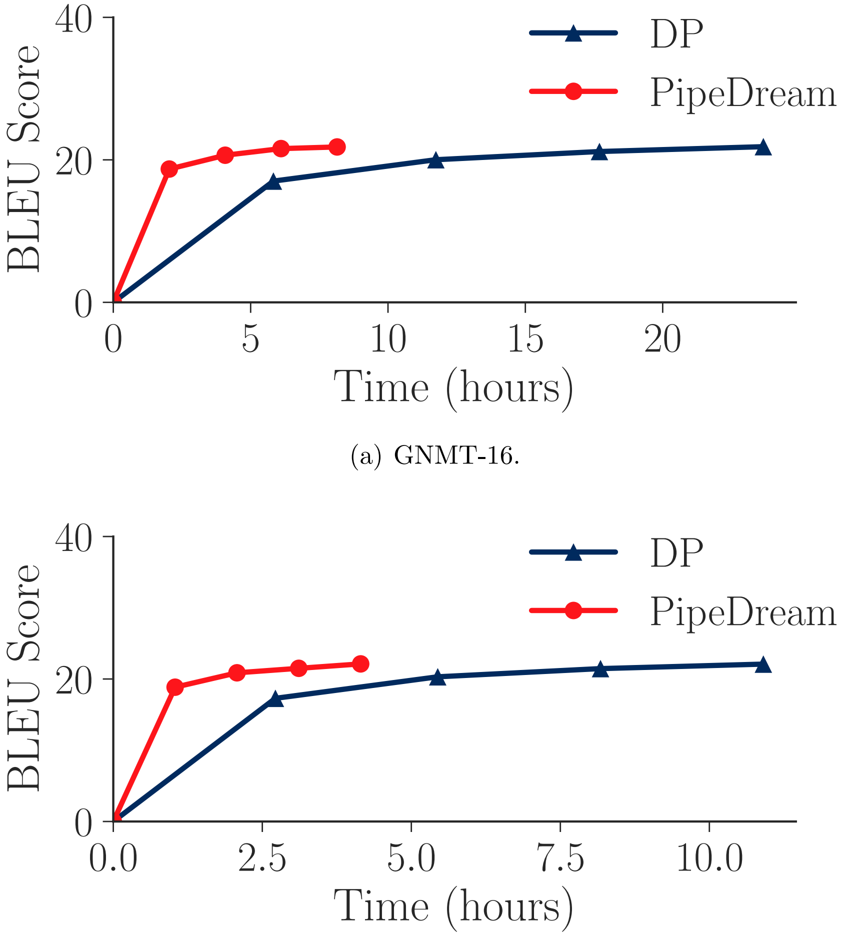
servers on Cluster-B (16 GPUs total), PipeDream reaches target accuracy 4.33× faster than DP when training VGG-16. Because of the change in hardware topology, PipeDream chooses to use all 16 GPUs unlike the 12-1 configuration on Cluster-A. Due to the faster interconnects on Cluster-B, both PipeDream and DP reach target accuracy faster than on Cluster-A (see Figure 5.9).

For training ResNet-50 on Cluster-A, PipeDream's partitioning algorithm recommends data parallel as the optimal configuration (no pipelining or model parallelism). Later, in Sec. 5.5.5, we show the reason for this recommendation: non data-parallel configurations incur higher communication overheads than DP for ResNet-50. For AlexNet, we compare throughput of PipeDream on Cluster-A and Cluster-B. On Cluster-A, PipeDream achieves a time-per-epoch speedup of 3.4× and 4.62× for 1 and 4 servers, respectively. On Cluster-B, PipeDream achieves a speedup of 2.7× when using 16 GPUs.

Translation. We show results for the GNMT model with 8 LSTM layers (GNMT-8) and 16 LSTM layers (GNMT-16). Using 1 server on Cluster-A, PipeDream reaches target accuracy 3.26× and 3.45× faster than DP for GNMT-8 and GNMT-16, respectively. When using 4 servers (16 GPUs) on Cluster-A, PipeDream reaches target accuracy 2.67× (GNMT-8) and 2.91× (GNMT-16) faster than DP. We show in Sec. 5.5.5 that PipeDream significantly reduces communication compared to DP, thus reducing its time to target accuracy.

On 2 servers (16 GPUs) of Cluster-B, PipeDream reaches target accuracy 2.14× and 2.62× faster than DP. For GNMT, PipeDream chooses a "straight" configuration (no stage replication). Since PipeDream uses the same configuration on both Cluster-A and Cluster-B (in contrast to VGG-16), and because DP has less communication overhead when training on Cluster-B (see faster time to target accuracy in Figure 5.10), the relative speedup for PipeDream over DP on Cluster-B is lower.

Language Modeling. This model is made up of six LSTM layers that contain a large number of model parameters (0.41GB), making data-parallel training inefficient. Using



(a) GNMT-16.

(b) VGG-16.

Figure 5.12: Accuracy vs. epoch using 16 GPUs on Cluster-B.

a single server on Cluster-A, PipeDream reaches target accuracy 4.25× faster than DP. PipeDream chooses a "straight" configuration that reduces communication by 88% compared to DP.

Video Captioning. PipeDream chooses to use a 2-1-1 configuration for the S2VT on Cluster-C, reducing communication by 85% compared to DP, which in turn allows it to reach target accuracy 3.01× faster than DP.

Comparison to DP with large minibatches. Recent work has demonstrated that using large minibatches is effective for training ResNet-50 and AlexNet models, especially when combined with Layer-wise Adaptive Rate Scaling (LARS). [45, 106, 58]. LARS uses different learning rates for each layer based on the ratio of the weight norm to the gradient norm. Large minibatches decrease the frequency of communication, reducing the communication overhead for data parallelism. Figure 5.11 shows 8-server results for data-parallel training of VGG-16 using LARS and large minibatches on Cluster-C. Minibatches of 1024 had the fastest time-to-target-accuracy, while minibatches of 4096 and 8192 failed to reach target accuracy highlighting the lack of generality of such approaches. PipeDream still reaches target accuracy over 2.4× faster than the fastest data-parallel option (1024 with LARS).

Comparison to Asynchronous Parallelism (ASP). ASP can reduce communication overhead in data-parallel training. Unlike BSP, which synchronizes parameters after every minibatch, ASP has no synchronization overheads, and workers use the most recent parameter data *available*. The result is often poor statistical efficiency. For example, when training VGG-16 on 4 Cluster-B servers, ASP data-parallel takes 7.4× longer than PipeDream to reach a 48% accuracy (when we terminate ASP for taking too long to converge), even though ASP has minimal communication delays.

Evaluating quality of work. Figure 5.12 shows accuracy vs. epoch for VGG-16 and GNMT-16 on Cluster-B. We do not show statistical efficiency for other experiments due to space constraints, but we consistently observe that PipeDream reaches target accuracy in the same number of epochs as DP.

5.5.3 Comparison to Other Intra-batch Parallelism

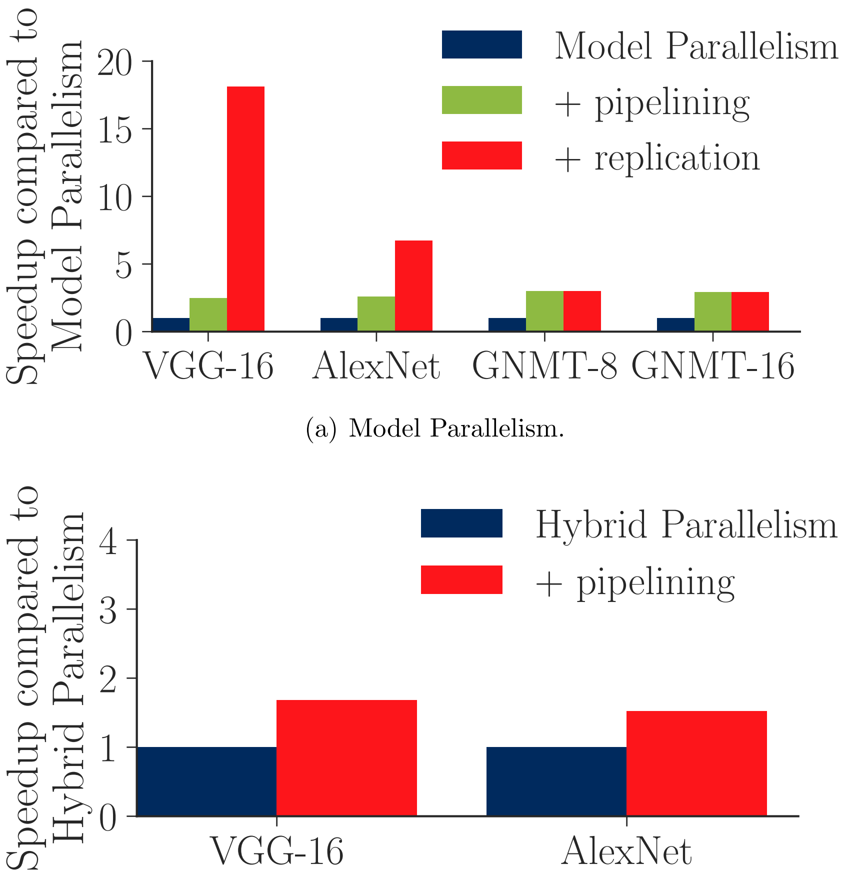
This section compares PipeDream to other intra-batch parallelization techniques besides data parallelism.

Model Parallelism. Figure 5.13(a) compares model parallelism (blue bars), straight pipelines without replication (green bars), and pipelining with stage replication (red bars). For all four models, pipelining alone increases throughput by 2.5× or more. For GNMT-8 and GNMT-16, PipeDream's optimizer chooses not to replicate any stages, resulting in identical configurations for the green and red bars. For VGG-16 and AlexNet, PipeDream replicates the first stage, leading to speedups of 18× and 7× compared to model parallelism.

Hybrid Parallelism. Figure 5.13(b) shows that pipelining for a configuration that combines data and model parallelism (similar to those proposed by Krizhevsky et al. [64] and FlexFlow [61, 60]) increases throughput by as much as 70%. In running FlexFlow for AlexNet on Cluster-B (not shown in Figure 5.13(b)), we observe that PipeDream is 1.9× faster; a speedup due to pipelining over hybrid parallelism.

5.5.4 Comparison to Inter-batch Parallelism

We compare training GNMT-16 using PipeDream and our implementation of GPipe using 16 GPUs on Cluster-A and Cluster-B. GPipe does not provide an algorithm for partitioning work across stages, so we use the same partitions as PipeDream. GPipe also does not provide



(a) Model Parallelism.

(b) Hybrid Parallelism.

Figure 5.13: Comparison of PipeDream (red) vs. non-DP intra-batch techniques (blue) for 4-GPU configurations on Cluster-A.

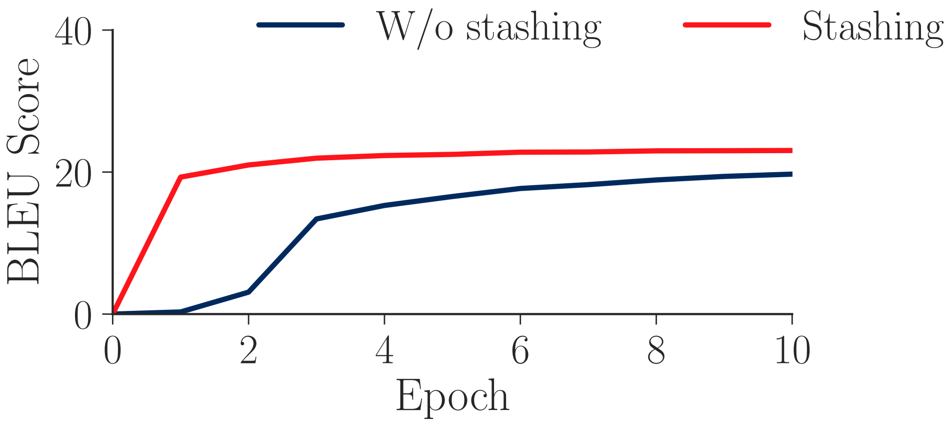


Figure 5.14: Impact of weight stashing on convergence for GNMT-8.

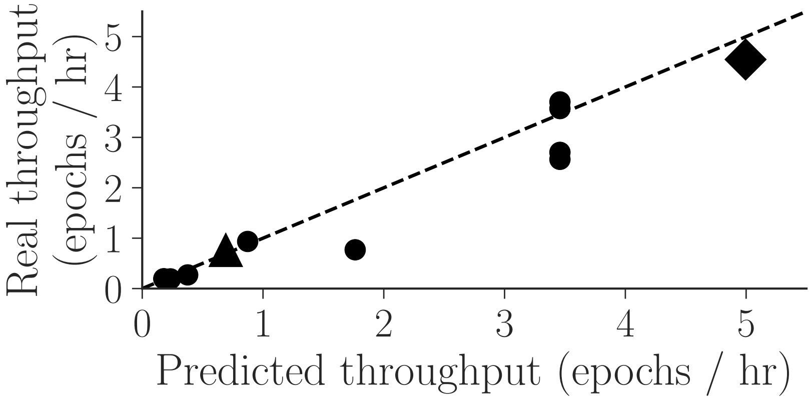


Figure 5.15: Real vs. optimizer's predicted throughput for VGG-16 with 16 workers. Each symbol represents a different partioning configuration, including the triangle for vanilla data-parallelism and the diamond for the optimizer's selection.

an algorithm for how many items should be permitted into the "pipeline" (pipeline depth). When we set the pipeline depth to be equivalent to "NOAM" in PipeDream (Sec. 5.3.2), GPipe experiences 55% and 71% throughput slowdowns compared to PipeDream on Cluster-A and Cluster-B, respectively. Setting the pipeline depth for GPipe to the largest number that does not cause an out-of-memory exception, leads to throughput slowdowns of 35% and 42% on Cluster-A and Cluster-B, respectively. Note that, unlike PipeDream, GPipe may suffer from reduced statistical efficiency because each weight is updated only when the pipeline is flushed (Figure 5.3), but we did not explicitly measure this.

5.5.5 Microbenchmarks

We evaluate PipeDream's optimizer and weight stashing mechanism, as well as its communication and memory overhead.

Optimizer. Throughout our experiments, we find the optimizer to be quick and effective, generating optimal training configurations in under 8 seconds. As one example, Figure 5.15 shows real vs. predicted throughputs for various configurations for VGG-16 with 16 workers. Predicted and real throughputs are strongly linearly correlated, and the optimizer picks the best configuration among those tested.

Weight Stashing. One of the key components required for performing effective pipeline-parallel training is weight stashing (Sec. 5.3.3). Figure 5.14 compares PipeDream training of GNMT-8 on a 4-GPU Cluster-A server with and without weight stashing. When weight stashing is *not used*, the training process fails to reach target accuracy, because the backward pass is performed on a different weight version from the one used in the forward pass. This leads to incorrect gradient computations, adversely affecting training quality.

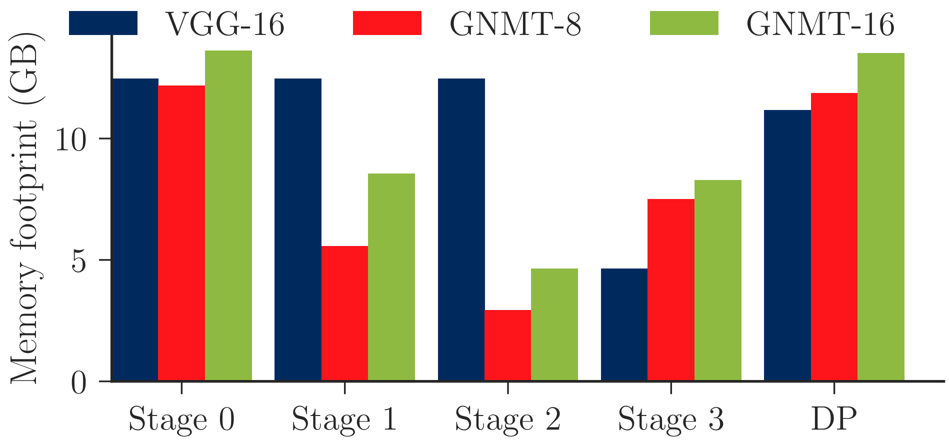


Figure 5.16: Memory footprint for various 4-GPU configurations in PipeDream. Data Parallel (DP) overhead is equivalent on all 4 GPUs.

Memory Footprint. Figure 5.16 shows the per-stage memory footprint of PipeDream for 4-stage configurations for three different models. PipeDream's worst-case memory footprint is on par with that of data parallelism, even though PipeDream stashes multiple weight and activation versions. This is because each stage in PipeDream is responsible for only a fraction of the total number of weights and activations in the model. As PipeDream scales to include more stages, the memory overheads remain consistent, as discussed in Sec. 5.3.3.

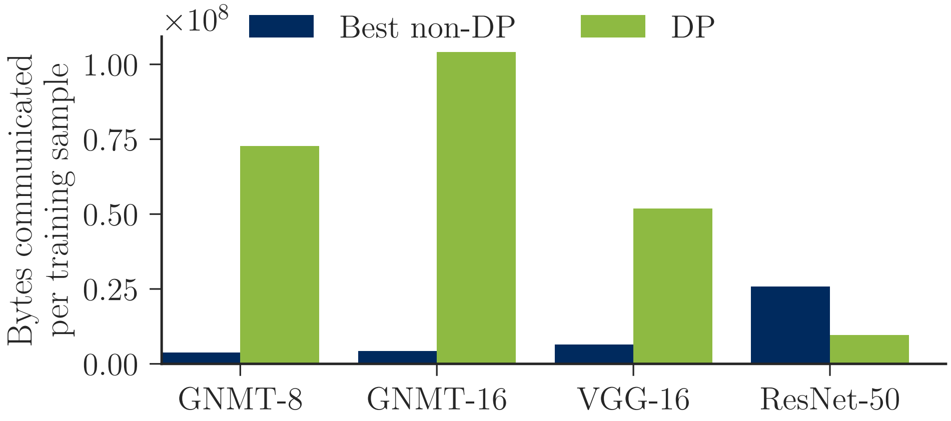
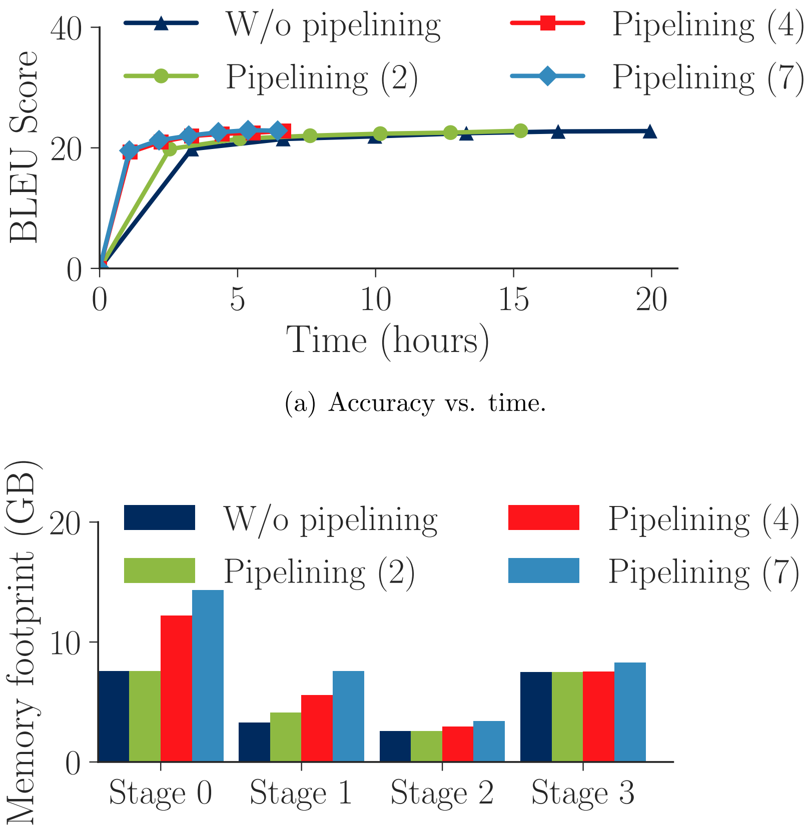


Figure 5.17: Bytes communicated per training sample by data-parallel (DP) and the best non-DP configurations for 4 GPUs on Cluster-A.

Communication Overhead. Figure 5.17 shows the amount of communication performed *per training sample* by PipeDream dwarfs data-parallel training for GNMT-8, GNMT-16, and VGG-16. For ResNet-50, the amount of communication for the best *non-data-parallel* configuration is higher than the DP configuration, thus explaining why PipeDream's optimizer chooses to train it using a data-parallel configuration.



(a) Accuracy vs. time.

(b) Memory overhead.

Figure 5.18: Effect of pipeline depth for GNMT-8 on 4 V100s in Cluster-A on accuracy-vs-time and memory overhead.

Effect of Pipeline Depth. Figure 5.18 shows the effect of the number of in-progress minibatches on (a) training time to accuracy and (b) memory overhead for GNMT-8. We make three observations. 1) Memory footprint with no pipelining is different across stages, since PipeDream's optimizer tries to load balance compute and communication, and *not* memory footprint (working set still fits comfortably in GPU memory). 2) As the pipeline depth increases, from 2 to 7, memory footprint increases because the number of weights and activations that need to be stashed increases proportionally. 3) In our experiments, a pipeline depths of 4 (NOAM) and 7 give the best time to target accuracy. While the working set of stages fits in GPU memory (16 GB), if required, pipeline depth can be decreased to trade time-to-accuracy for reduced memory footprint.

5.6 Summary

This chapter presents PipeDream, which uses pipeline-parallelism to address the communication bandwidth limitations for DNN training. PipeDream uses a combination of pipelining, model-, and data- parallelism to reduce the communication overheads that can bottleneck intra-batch parallelism. Our experiments show that PipeDream reduces training by up to 5× or more for seven different DNN models across four different ML tasks compared to intra-batch parallelism alone.

Chapter 6

Conclusion and Future Directions

6.1 Conclusion

This dissertation demonstrates that improvements of 5x or more can be achieved for training ML tasks in shared computing environments by structuring software frameworks and work distribution to exploit transient resources and to address performance jitter and communication bandwidth limitations. To do so, it presents three case study systems.

First, we designed FlexRR, a system that addresses the straggler problem for iterative convergent data-parallel ML. By integrating flexible consistency bounds with temporary peer-to-peer work reassignment, FlexRR successfully avoids having unhindered workers wait for workers experiencing slowdowns. Experiments with real ML applications under a variety of naturally-occurring and synthetic straggler behaviors confirm that FlexRR achieves near-ideal performance. On Amazon EC2 and Microsoft Azure, with their inherent variability (no injected delays), this results in 15–56% reduction in runtimes, on average, over SSP and BSP. Experiments with various synthetic straggler patterns confirm that FlexRR consistently mitigates stragglers, resulting in up to 5–10× improvement over BSP and SSP in extreme cases.

Second, we designed Proteus, a system that aggressively exploits transient revocable machines to complete ML model training faster and cheaper. For example, by exploiting the EC2's spot market, Proteus saves ≈85% compared to using only on-demand machines. By combining non-transient (e.g., on-demand) and transient (spot) machines, Proteus can rapidly and efficiently incorporate transient resources and deal with revocations. Combined with its aggressive allocation strategy, Proteus saves ≈50% compared to a state-of-the-art checkpointing-based approach using a standard spot market bidding strategy.

Third, we designed PipeDream, a distributed training system designed specifically for training Deep Neural Network models. PipeDream uses pipeline-parallel training, which addresses the communication overheads that can bottleneck data-parallel training of large DNNs. PipeDream automatically partitions and aggressively pipelines DNN training across worker machines. Compared to state-of-the-art intra-batch parallelization approaches, PipeDream is up to 5× or more faster in "time to target accuracy" for experiments with seven different DNNs on three different clusters.

6.2 Future Directions

This section discusses several future research directions for applying or extending the work described in this dissertation.

6.2.1 Combining FlexRR, Proteus, and PipeDream

The three case studies presented in this dissertation all address different challenges that arise in shared computing environments. An interesting continuation would explore how to effectively combine the solutions proposed into a single general solution and evaluate the aggregate benefits. For example, combining FlexRR and Proteus would improve performance as we found that using tiers of reliability in Proteus (Sec. 4.2.2) introduced stragglers into the training process.

6.2.2 Automatic Stage Transitions in Proteus

Section 4.2.3 presents the mechanisms used by Proteus to transition between different stages depending on the ratio of transient to reliable resources. Currently, the stage transitions are performed based on ratios that we observed from our experiments specifically targeted at our workloads. However, as the hardware and workload characteristics change, it is likely that the optimal transition ratios will also change. Instead of performing manual profiling, a future direction of research could explore how to design an automated mechanism for triggering stage transitions. One possible approach could be to monitor network traffic on the on-demand resources (which are the first to become bottle-necked), and trigger stage transitions when the traffic exceeds or falls below specified thresholds.

6.2.3 Pipeline Parallelism with Heterogeneous Resources

Chapter 5 shows that pipeline-parallelism can provide significant improvements in training of deep learning models compared to model and data-parallelism. All of our experiments were performed on a homogeneous set of resources. However, pipeline parallelism is more amenable to a heterogeneous approach than data-parallelism. Two interesting types of heterogeneity that can be explored in the context of pipeline parallelism are: 1) different machine sizes with homogeneous compute accelerators (e.g., 1-GPU and 4-GPU machines); and 2) machines with different types of compute accelerators (e.g., GPUs, TPUs). Using different machine sizes can be beneficial when there are different levels of replication amongst the stages in the pipeline. For example, when training the VGG-16 model using four Machines, with 4-GPUs per machine (Sec 5.5.2), it's faster to use three 4-GPU machines, and four 1-GPU machines. Heterogeneity in compute accelerators would allow mapping computation for different layer types onto the compute accelerators best suited to carry out the computation.

6.2.4 Operator Level Partitioning for Pipeline Parallelism

Section 5.3.1 presents the optimizer that is used to partition the DNN computation across the compute resources available to PipeDream. Currently, it only considers partitioning at the granularity of the DNN layers. Within each layer, there typically is a large number of mathematical operations (e.g., matrix-multiply) that could also be partitioned to different devices [61]. Finer-grained partitioning at the operator level would potentially allow PipeDream to distribute computation more evenly amongst the compute resources, further reduce communication, and efficiently scale to larger resource footprints. Finer-grained partitioning would also significantly increase the computational complexity of the problem being solved by PipeDream's partitioning algorithm.

6.2.5 Scheduling ML Training in Hybrid Clouds

Although not strictly a continuation of the three case studies, another future research area is at the intersection of ML and hybrid clouds. As cloud computing has increased in popularity, many users have begun using *hybrid cloud* configurations. Hybrid clouds are a combination of private and public computing resources (e.g., combination of private company cluster and AWS EC2). When scheduling machine learning training jobs in a hybrid cloud, there are several important factors that need to be considered including hardware requirements, dataset location and size, dataset privacy (can the data be stored on a public cloud), job deadline, and other important factors.

It will be interesting to explore scheduling policies for deciding whether to place ML training jobs in private or public portions of a hybrid cloud. Policies that prioritize "packing" the private cloud prior to acquiring public cloud computing resources appear to be most cost-efficient. Scenarios where incoming jobs with datasets that are not permitted to be run on public clouds arrive once the private cloud is already full, however, could potentially cause such policies to have overheads from migrating jobs to public clouds and/or increased job-completion time. Additionally, public cloud providers could have higher-grade hardware availability (e.g., newer GPUs) than private clouds, which should also impact scheduling decisions. Research questions regarding designing such a system include:

- Success Metrics. The first research question that would need to be answered is what the success metric should be defined as for a hybrid cloud scheduler designed specifically for ML training. Possible metrics include total cost, job utility, private cluster utilization. It is likely that the correct solution would be to allow users to tweak parameters that dictate the success metric.
- Resource Allocation. Once a success metric is defined, a system for job placement in both static and elastic settings will need to be designed. In static settings, where the user specifies amount of resources to be allocated to a job, the scheduler will need to decide whether to place the job in the private or public portion of the cloud. In elastic settings, the scheduler will additionally need to determine the amount of resources to allocate to each job.

Bibliography

[1] AWS EC2. `http://aws.amazon.com/ec2/`.

[2] Google Compute Engine. `https://cloud.google.com/compute/`.

[3] New York Times dataset. `http://www.ldc.upenn.edu/`.

[4] Spot Bid Advisor. `https://aws.amazon.com/ec2/spot/bid-advisor/`.

[5] Gloo. `https://github.com/facebookincubator/gloo`, 2018.

[6] Pytorch. `https://github.com/pytorch/pytorch`, 2018.

[7] Pytorch ddp. `https://pytorch.org/docs/stable/_modules/torch/nn/parallel/distributed.html`, 2018.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016.

[9] U. A. Acar, A. Chargueraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 219–228. ACM, 2013.

[10] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. *ACM Transactions on Economics and Computation*, 1(3):16, 2013.

[11] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012.

[12] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. Loose synchronization for large-scale networked systems. In *USENIX Annual Tech*, 2006.

[13] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *NSDI'13*, pages 185–198, 2013.

[14] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16. USENIX Association, 2010.

[15] Apache Hadoop. `http://hadoop.apache.org/`.

[16] Baidu Inc. Bringing HPC Techniques to Deep Learning, 2017.

[17] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. In *IEEE International Conference on Cluster Computing*, pages 1–12, 2006.

[18] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[19] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[20] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *Advances in neural information processing systems*, pages 161–168, 2008.

[21] D. L. Chen and W. B. Dolan. Collecting highly parallel data for paraphrase evaluation. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 190–200. Association for Computational Linguistics, 2011.

[22] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[23] X. Chen, A. Eversole, G. Li, D. Yu, and F. Seide. Pipelined Back-propagation for Context-dependent Deep Neural Networks. In *Interspeech*, 2012.

[24] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, pages 571–582, 2014.

[25] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the straggler problem with bounded staleness. In *USENIX conference on Hot topics in operating systems (HotOS)*, 2013.

[26] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Re, and M. Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *arXiv preprint arXiv:1806.01427*, 2018.

[27] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. 2017.

[28] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *USENIX Annual Technical Conference*, pages 37–48, 2014.

[29] H. Cui, A. Tumanov, J. Wei, L. Xu, W. Dai, J. Haber-Kucharsky, Q. Ho, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting iterative-ness for parallel ML computations. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[30] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.

[31] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC'14, pages 2:1–2:14. ACM, 2014.

[32] J. Dean. Achieving rapid response times in large online services. In *Berkeley AMPLab Cloud Seminar*, 2012.

[33] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[34] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[35] M. Denkowski and A. Lavie. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the ninth workshop on statistical machine translation*, pages 376–380, 2014.

[36] NVIDIA DGX-1. `https://www.nvidia.com/en-us/data-center/dgx-1/`.

[37] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC'09, pages 53:1–53:11. ACM, 2009.

[38] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng. Dynamic load balancing of unbalanced computations using message passing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[39] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European conference on Computer Systems*, pages 99–112. ACM, 2012.

[40] K. B. Ferreira, P. G. Bridges, R. Brightwell, and K. T. Pedretti. The impact of system design parameters on application noise sensitivity. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, CLUSTER'10, pages 146–155. IEEE Computer Society, 2010.

[41] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th Conference on Knowledge Discovery and Data Mining (KDD 11)*, 2011.

[42] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX; login*, 38(3), 2013.

[43] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. OSDI*, 2012.

[44] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica.

GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, 2014.

[45] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[46] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences of the United States of America*, 2004.

[47] A. Gupta, B. Acun, O. Sarood, and L. V. Kalé. Towards realizing the potential of malleable jobs. In *Proceedings of the 21st International Conference on High Performance Computing (HiPC 14)*, pages 1–10. IEEE, 2014.

[48] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons. Tributary: spot-dancing for elastic services with latency SLOs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 1–14, 2018.

[49] A. Harlap, H. Cui, W. Dai, J. Wei, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Addressing the straggler problem for iterative convergent parallel ML. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC 16)*, pages 98–111. ACM, 2016.

[50] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[51] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons. Proteus: agile ML elasticity through tiered reliability in dynamic resource markets. In *EuroSys*, pages 589–604, 2017.

[52] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[53] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[54] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[55] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965*, 2018.

[56] Z. Huo, B. Gu, Q. Yang, and H. Huang. Decoupled parallel backpropagation with convergence guarantee. *arXiv preprint arXiv:1804.10574*, 2018.

[57] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789. IEEE,

2018.

[58] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[59] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[60] Z. Jia, S. Lin, C. R. Qi, and A. Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In *ICML*, 2018.

[61] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. *arXiv preprint arXiv:1807.05358*, 2018.

[62] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[63] E. Krevat, J. Tucek, and G. R. Ganger. Disks are like snowflakes: no two are alike. In *USENIX conference on Hot topics in operating systems (HotOS)*, 2011.

[64] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[65] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[66] J. Langford, A. J. Smola, and M. Zinkevich. Slow learners are fast. In *Proceedings of the 23rd Annual Conference on Neural Information Processing Systems (NIPS 09)*, 2009.

[67] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proc. OSDI*, pages 583–598, 2014.

[68] M. Li, D. G. Andersen, A. J. Smola, and K. Yu. Communication efficient distributed machine learning with the parameter server. In *NIPS*, pages 19–27, 2014.

[69] J. Liu, J. Chen, and J. Ye. Large-scale sparse logistic regression. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 547–556. ACM, 2009.

[70] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.

[71] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[72] A. Marathe, R. Harris, D. Lowenthal, B. R. De Supinski, B. Rountree, and M. Schulz. Exploiting redundancy for cost-effective, time-constrained execution of HPC appli-

cations on Amazon EC2. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 279–290. ACM, 2014.

[73] S. Merity, N. S. Keskar, and R. Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

[74] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[75] A. Mirhoseini, H. Pham, Q. Le, M. Norouzi, S. Bengio, B. Steiner, Y. Zhou, N. Kumar, R. Larsen, and J. Dean. Device placement optimization with reinforcement learning. 2017.

[76] D. Mishkin and J. Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

[77] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[78] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC'03, pages 55–55. ACM, 2003.

[79] Power-law distribution. `http://en.wikipedia.org/wiki/Power_law`.

[80] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[81] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[82] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[83] F. Seide and A. Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016.

[84] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[85] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. On parallelizability of stochastic gradient descent for speech dnns. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE SPS, May 2014.

[86] H. Seltman. Approximations for $\mathbf{E}(\frac{R}{S})$ and $\mathbf{V}(\frac{R}{S})$ for any random variables R and S. `http://www.stat.cmu.edu/~hseltman/files/ratio.pdf`.

[87] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[88] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys 16)*, page 6. ACM, 2016.

[89] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[90] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 368–380. ACM, 2015.

[91] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[92] S. Subramanya, T. Guo, P. Sharma, D. Irwin, and P. Shenoy. SpotOn: a batch computing service for the spot market. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC 15)*, pages 329–341. ACM, 2015.

[93] S. Tang, J. Yuan, and X.-Y. Li. Towards optimal bidding strategy for Amazon EC2 cloud spot instance. In *Proceedings of the 5th IEEE International Conference on Cloud Computing(CLOUD 12)*, pages 91–98. IEEE, 2012.

[94] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[95] A. Tumanov, J. Cipar, G. R. Ganger, and M. A. Kozuch. alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 25. ACM, 2012.

[96] Uber Technologies Inc. Meet Horovod: Ubers Open Source Distributed Deep Learning Framework for TensorFlow, 2017.

[97] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), Aug. 1990.

[98] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[99] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, , B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC 13)*, page 5. ACM, 2013.

[100] S. Venugopalan, M. Rohrbach, J. Donahue, R. Mooney, T. Darrell, and K. Saenko. Sequence to sequence-video to text. In *Proceedings of the IEEE international conference on computer vision*, pages 4534–4542, 2015.

[101] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys 15)*, page 18. ACM, 2015.

[102] J. Wang, J. Yang, K. Yu, F. Lv, T. Huang, and Y. Gong. Locality-constrained linear coding for image classification. In *Proceedings of the Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3360–3367. IEEE, 2010.

[103] J. Wei, W. Dai, A. Qiao, Q. Ho, H. Cui, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *SoCC*, 2015.

[104] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[105] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *Proceedings of the 22th Conference on Knowledge Discovery and Data Mining (KDD 15)*, pages 1335–1344. ACM, 2015.

[106] Y. You, I. Gitman, and B. Ginsburg. Scaling SGD batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888*, 2017.

[107] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[108] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *OSDI*, 2008.

[109] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, 2017. USENIX Association.

[110] L. Zheng, C. Joe-Wong, C. W. Tan, M. Chiang, and X. Wang. How to bid the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 71–84. ACM, 2015.