Title
Ray: A Distributed Execution Engine for the Machine Learning Ecosystem

Author
Moritz, Philipp C

Publication Date
2019

Peer reviewed|Thesis/dissertation

Ray: A Distributed Execution Engine for the Machine Learning Ecosystem

by

Philipp C Moritz

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Michael I. Jordan, Chair
Professor Ion Stoica
Professor Ken Goldberg

Summer 2019

Ray: A Distributed Execution Engine for the Machine Learning Ecosystem

Abstract

Ray: A Distributed Execution Engine for the Machine Learning Ecosystem

by

Philipp C Moritz

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael I. Jordan, Chair

In recent years, growing data volumes and more sophisticated computational procedures have greatly increased the demand for computational power. Machine learning and artificial intelligence applications, for example, are notorious for their computational requirements. At the same time, Moores law is ending and processor speeds are stalling. As a result, distributed computing has become ubiquitous. While the cloud makes distributed hardware infrastructure widely accessible and therefore offers the potential of horizontal scale, developing these distributed algorithms and applications remains surprisingly hard. This is due to the inherent complexity of concurrent algorithms, the engineering challenges that arise when communicating between many machines, the requirements like fault tolerance and straggler mitigation that arise at large scale and the lack of a general-purpose distributed execution engine that can support a wide variety of applications.

In this thesis, we study the requirements for a general-purpose distributed computation model and present a solution that is easy to use yet expressive and resilient to faults. At its core our model takes familiar concepts from serial programming, namely functions and classes, and generalizes them to the distributed world, therefore unifying stateless and stateful distributed computation. This model not only supports many machine learning workloads like training or serving, but is also a good fit for cross-cutting machine learning applications like reinforcement learning and data processing applications like streaming or graph processing. We implement this computational model as an open-source system called Ray, which matches or exceeds the performance of specialized systems in many application domains, while also offering horizontally scalability and strong fault tolerance properties.

To my parents *Hugo and Birgit*,
and my sisters *Christine and Sophie*,
for their constant love and support!

Contents

List of Figures

List of Tables

Acknowledgments

I am deeply grateful to the many people who were part of my PhD journey. They helped me to grow professionally and as a person, and have made my time at Berkeley unforgettable. Without them this thesis would not have been possible. I would like to thank:

My advisor Michael Jordan for bringing me to Berkeley, for inspiring and encouraging me throughout my PhD, for bringing together such an exceptional and supportive group of peers and for motivating all of us with his kindness, enthusiasm and positivity.

My advisor Ion Stoica for mentoring me. His obsession with real-world impact and research that truly matters is unparalleled. He taught me many valuable lessons about research, systems design, planning, products and execution and truly expanded my horizon.

Robert Nishihara, who has influenced my PhD journey like nobody else. His ability to confidently bust through any obstacle that might arise has greatly inspired me and helped me to not only see, but also reach the light at the end of the tunnel.

John Schulman with whom I collaborated closely at the beginning of my PhD. We have had many great conversations over the years and he has been a source of inspiration and ideas ever since!

Cathy Wu for countless discussions about research and life, her kindness and all the unforgettable memories we forged.

I would like to thank all the members of the Ray team, including Stephanie Wang, Eric Liang, Richard Liaw, Devin Petersohn, Alexey Tumanov, Peter Schafhalter, Si-Yuan Zhuang, Zongheng Yang, William Paul, Melih Elibol, Simon Mo, William Ma, Alana Marzoev, and Romil Bhardwaj. Thanks for a great collaboration. I have learned a lot from you!

I would like to thank my friends and colleagues from the research groups I have been part of. SAIL has been incredible. It is hard to describe the amount of knowledge and ideas that were transferred at our weekly research meetings, and the positivity and support from all of you, including Stefanie Jegelka, Ashia Wilson, Horia Mania, Mitchell Stern, Tamara Broderick, Ahmed El Alaoui, Esther Rolf, Chi Jin, Max Rabinovich, Nilesh Tripuraneni, Karl Krauth, Ryan Giordano, John Duchi, and Nick Boyd. The AMPLab, RISELab and BAIR have been a great community of friends and collaborators. Berkeley is unique for its collaborative research style, and the lab culture plays a major role in that.

I would like to thank my quals and thesis committee, Ken Goldberg, Joey Gonzalez and Fernando Perez for their insights, advice and support over the years!

My friend Fanny Yang for constant friendship and support, the many races and memories. You truly made a difference!

Many friends who made this journey unforgettable, including Fan Wei, Olivia Anguli, Richard Shin, Frank Li, Atsuya Kumano, Jordan Sullivan, Vinay Ramasesh, Jacob Steinhardt, Ludwig Schmidt, Reinhard Heckel, Jacob Andreas, Alyssa Morrow, Jeff Mahler, Mehrdad Niknami, Smitha Milli, Marc Khoury and Sasha Targ.

My home for the last five years, a big house on the south side of Berkeley, called "Little Mountain". Rishi Gupta for founding it and everybody living there for the great time we had together.

I would like to thank the Nishihara family for so kindly inviting and integrating me into many family gatherings and making me feel at home in the Bay area.

This thesis is dedicated to my family. My parents Hugo and Birgit, who created the right environment for me to thrive. Their unconditional love and support have made all the difference. My sisters Christine and Sophie for being awesome life companions, for their support and guidance over the year.

Chapter 1

Introduction

We are living in a remarkable time. In the span of a single human lifetime, we have seen the birth of machines that can process data, automatically perform tasks and make decisions. They have grown to have substantial real-world impact. If you are looking for any piece of information, there is a good chance that Google can find it for you. If you want to buy a product or get recommendations on what to buy, there is a large number of services on the internet that will help you to spend money, including Amazon. If you want to quickly get from A to B without having to worry about the details, ride sharing services like Uber or Lyft are the way to go. And not only our personal lives but also society crucially depends on our digital infrastructure. Science, education, our health care system and public administration as well as corporations would be unable to operate and coordinate the work of so many people without the help of computers. Computers are capable of running such diverse workloads as crunching numbers for scientific simulations, running complex queries on relational data to help operate large corporations or connecting people around the globe, and they are slowly starting to perform some of the complex cognitive tasks that only humans were capable of in the past. And fast forward another human lifetime, we will look back and realize that todays capabilities pale in comparison to what will be possible then.

Much of the computation is happening in the cloud, a large collection of servers that can be rented from providers like Amazon, Microsoft or Google. We typically use "edge" devices like smartphones or laptops to interface with the digital world, but in most cases the actual logic is implemented in the cloud. You become painfully aware of this if your phone gets disconnected from the internet and many important applications stop working. There are good reasons to shift much of the processing into the cloud: Moore's law is ending, therefore single-core performance is not getting much faster, which means compute heavy application logic needs a large number of cores, which are typically only available on a cluster. Computing in the cloud also improves resource usage as processors can be shared between users and applications. One of the most important reasons why companies typically prefer running application logic in the cloud rather than the edge is control: They can determine the compute environment, have full access to the data and can secure private data and algorithms more easily.

Given this trend, it is quite surprising that distributed programming in the cloud is still very hard. Distributed systems is one of the more complex topics in computer science and while there is a large amount of research in this field, there is less work on making it easier for non-experts to build distributed software. If they want flexible systems, programmers typically have to build distributed applications from low-level primitives like remote procedure call layers, distributed key-value stores and cluster managers, which requires a lot of expertise, duplicates work between different distributed systems and makes the task of debugging a distributed application even harder than it already is. Clearly distributed programming in the cloud is not yet as easy as programming on a laptop where programmers can choose from a rich set of high-level libraries, write complex applications with ease by composing them, and inspect the flow of the program and stop and debug it if something goes wrong.

These observations especially apply in the fields of machine learning and artificial intelligence. In fact, artificial intelligence is one of the most computationally expensive workloads due to ever increasing sizes of models and datasets. Many distributed systems have been developed to handle the scale of these applications: There are distributed data processing systems like MapReduce, Hadoop or Spark, stream processing systems like Flink or Kafka, distributed training systems like distributed TensorFlow, PyTorch or MxNet, distributed model serving systems like TensorFlow Serving or Clipper, and hyperparameter search tools like Vizier. However, each of these systems have a fairly narrow design scope. Therefore, in order to build end-to-end applications, practitioners often have to glue several systems together, which incurs high costs: Data needs to be converted at the system boundaries with costs for both development productivity and runtime efficiency, different fault tolerance mechanisms need to be combined into an overall strategy, each of the systems needs to be managed and resources need to be allocated for each of them, which can lead to poor cluster utilization. Even worse, emerging workloads such as reinforcement learning, online learning and other cross-cutting applications need more flexible programming models and have stringent performance requirements that often cannot be fulfilled by gluing together existing systems. Practitioners are therefore often left to write their own distributed systems for such workloads from low-level primitives, reinventing many mechanisms of distributed systems like scheduling, data transfer or failure handling.

In this thesis, we instead advocate for a different approach. Instead of gluing together separate distributed systems, different workloads like data processing, streaming, distributed training and model serving should instead be implemented as reusable distributed libraries that run on top of one general-purpose system. The system should expose a programming model that is close to the programming models developers are familiar with from the single machine setting. This allows to expose common functionality like debugging, monitoring, distributed scheduling and fault tolerance through an underlying distributed system and allows us to bring the cluster programming experience much closer to programming a laptop. The main contribution of this thesis is in designing a programming model for distributed computation and an implementation of that model which can support a wide variety of different distributed computing workloads, including the machine learning and artificial intelligence

applications mentioned above. The system and a number of libraries for different applications have been implemented together with a large number of collaborators at Berkeley and many collaborators from both the wider open source community and various companies.

The thesis is organized as follows:

- In chapter 2, we give an overview over the spectrum of existing distributed programming models from more specialized to general. This gives the reader an appreciation of the design space and motivates the design decisions we make for the Ray programming model.
- In chapter 3, we describe a system for distributed training that we built on top of Apache Spark, which uses the BSP model, one of the programming models described in chapter 2. The shortcomings of this approach, together with the insights from our work in reinforcement learning (see [114] and [112]) were the main motivations for the design of Ray. This material was previously published in [83].
- In chapter 4, we study the requirements of a general purpose distributed system that can support emerging artificial intelligence applications like reinforcement learning. This material was previously published in [93].
- In chapter 5, the main chapter of the thesis, we describe the design and implementation of Ray. By decoupling the control and data plane and introducing stateful actors, it can fulfill the requirements outlined in chapter 4 and serves as an execution engine for a diverse set of tasks in distributed machine learning. This material was previously published in [82].
- In chapter 6, we present an algorithm for large-scale optimization with a linear convergence rate that is well-suited for the Ray architecture described in chapter 5. This material was previously published in [81].

Chapter 2

The Distributed Computation Landscape

To get more context on how a flexible distributed system should be designed, let us first review existing solutions to make sure we are not reinventing the wheel and understand the design space. In this chapter, we will focus on practical systems for distributed computing (as opposed to research systems that demonstrate the viability of specific ideas). We will also view these systems under the lens of their *programming model*, because that's their most important characteristic for users and for building distributed applications.

In table 2.1 we give an overview over existing parallel and distributed programming

	Programming Model	**State**	**Fault tolerance**	**Systems**
Bulk Synchronous Parallel	Iterative Computation	stateless	Checkpoint, Lineage	MapReduce, Hadoop, Spark
Task Parallel	Functional Programming	stateless	Lineage	TensorFlow, Dask, CIEL
Communicating Processes	Actors, Coroutines, Message Passing	stateful, but no shared state	Custom	Orleans, Erlang, Akka, MPI
Distributed Shared Memory	Threads	fully stateful	None	Unix Processes, Unified Parallel C

Table 2.1: The spectrum of distributed computing

models. The simplest way to do parallel computing is by executing a given function on a number of data items in parallel and storing the results. This is the SIMD (single instruction, multiple data) model, which is of course not sufficient, since more often than not the results need to be aggregated. SIMD plus aggregation is the Bulk Synchronous Parallel (BSP) model that we consider in section 2.1. Generalizing this pattern to arbitrary functions and data dependencies, but still keeping pure functions and not supporting stateful computation, gives the task parallel model, see section 2.2. Many applications like reinforcement learning or interactive serving systems need state however, which motivates extending the programming model to include stateful processes (see section 2.3). In the communicating processes model, state is still partitioned and processes can only exchange state by explicitly communicating. If we relax this restriction, we arrive at the distributed shared memory model (see section 2.4), which supports fully distributed state.

2.1 The Bulk Synchronous Parallel Model

The Bulk Synchronous Parallel (BSP) [129] model became very popular in the early days of the world wide web to crawl websites and process large amounts of data e.g. for building a search index. Implementations like MapReduce [33] or Hadoop [135] made it possible to run programs at a massive scale on cheap commodity hardware, without having to worry about faults. The programming model allows for a simple implementation, but is fairly restrictive. The program logic often has to be completely re-thought to adapt programs to this paradigm. In Fig. 2.2 we show how an inverted index can be built in MapReduce: In this example we have three mappers (one for page X, one for page Y and one for page Z) and two reducers (one for keys from A to L and one for keys from M to Z). Each mapper splits its document into tokens and classifies each of them according to whether they should be entered into the index or not. It then attaches the page to the token and sends all tokens with first letter in A-L to the first reducer and each token with first letter in M-Z to the second reducer. The reducers collect the tokens from each mapper, sort them and combine them into the inverted index.

The implementation in a system like Spark [137] is relatively easy, see the following code snippet. In the first two lines, we iterate through every document, convert it to lower case, split it into tokens and attach the document identifier to each token. In the third line, we then concatenate all the document identifiers corresponding to each token.

```
rdd.flatMap(lambda (document, contents):
                [(token, [document]) for token in contents.lower().split()])
   .reduceByKey(lambda a, b: a+b)
```

By chaining several such MapReduce phases together, we can express iterative computation in this model. By tracking the lineage of the computation, this model can be made fault tolerant [139]. This computation model is however fairly restrictive. It does not support state and makes it hard to parallelize applications that cannot naturally be decomposed

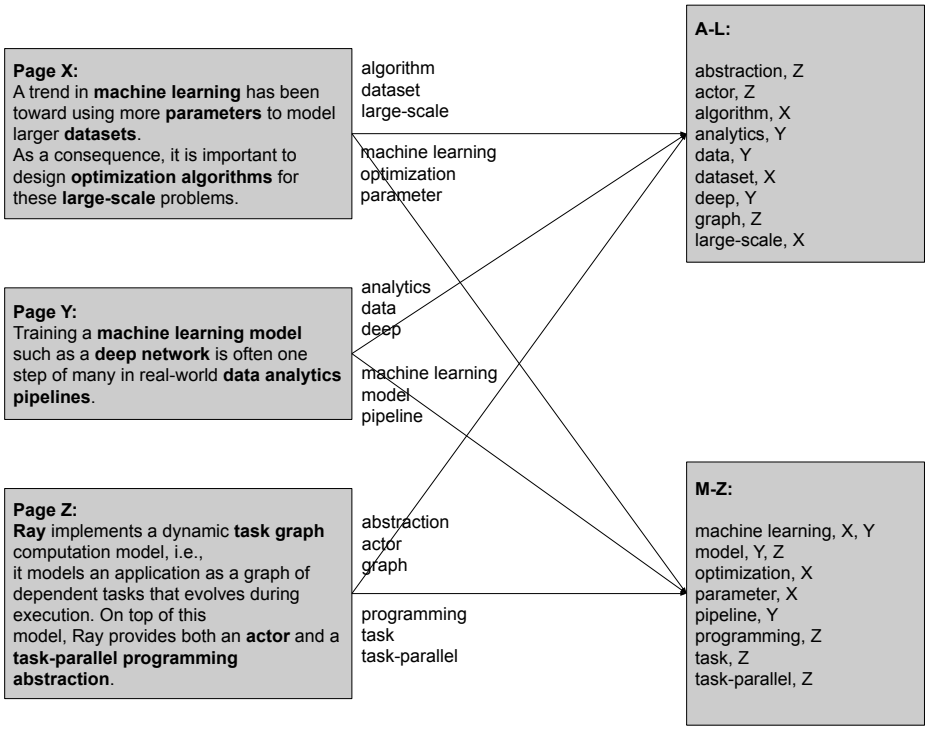


Figure 2.1: Building an inverted index with MapReduce

into a series of map reduce phases. While machine learning training can be expressed in this model [35], performance and flexibility requirements, e.g. for model parallel training, lead us to consider a generalization of the BSP model, the *task parallel model*.

2.2 The Task Parallel Model

The task parallel model allows the execution of arbitrary side-effect free functions in a distributed way. Each function has input arguments and produces outputs. As soon as all the inputs of a function are available, the function can run on one of the processors and produce its outputs, which will then itself trigger the computation of all functions that depend on these outputs. MapReduce computations are a special case of task parallel computations, where each mapper is a function that transforms a data item, and each reducer is a function that takes the transformed data items and combines them into the output. For the general task parallel model, each function can be different and data can be passed arbitrarily between them. In Fig. 2.2 we show how neural network computations are a natural fit for the task parallel model. There is some input data at the bottom, which gets reshaped and input into an affine layer (linear transform plus bias) and transformed with a rectified linear nonlinearity. Afterwards, a second affine layer is applied and the result is

transformed with a softmax nonlinearity. The last function then computes the loss by taking the cross entropy between the result and the ground truth labels.

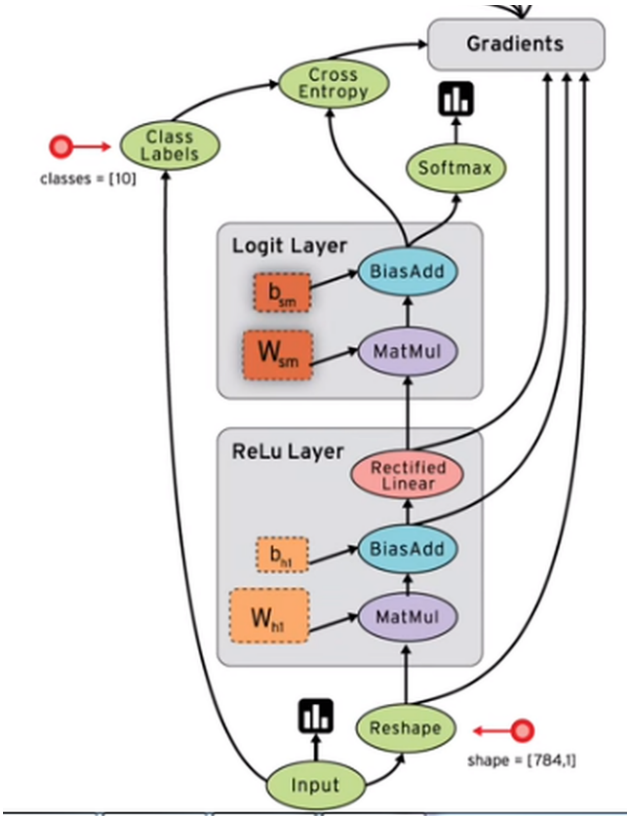


Figure 2.2: Neural network task graph, source `https://www.tensorflow.org/guide/graphs`

The main part of this task parallel computation can be expressed in TensorFlow [1] in the following way. The first three lines define the affine layer and the rectified linear nonlinearity. The second three lines define the second affine layer and the last line applies the softmax nonlinearity and computes the loss function.

```
weights1 = tf.Variable(...)
biases1 = tf.Variable(...)
hidden1 = tf.nn.relu(tf.matmul(images, weights1) + biases1)

weights2 = tf.Variable(...)
biases2 = tf.Variable(...)
logits = tf.matmul(hidden1, weights2) + biases2

result = tf.losses.sparse_softmax_cross_entropy(labels=labels, logits=logits)
```

These programs can be parallelized in distributed TensorFlow by annotating each function invocation with a device it shall be executed on. If there are functions that can be executed in parallel (in the above example, all the computations are serial), this can give large speedups. The task parallel programming model is more powerful than the BSP one and more compatible with the way that serial programs are typically expressed. It can be made fault tolerant either by checkpointing or by recording the lineage. However it cannot support state, which is required by many real applications.

2.3 The Communicating Processes Model

The first distributed computation model we are considering that supports state is communicating processes [53]. It is a generalization of the task parallel model because every task parallel program can be executed on communicating processes by scheduling the functions in the right order onto the processes. There are several realizations of communicating processes, including message passing implementations like MPI [44] or actor systems like Erlang [10], Akka [4] or Orleans [15]. We will focus on actor systems here, because they are as powerful as message passing systems but their programming model is more structured. The actor model was actually formally proposed in the context of artificial intelligence [50]. It models a distributed system as a collection of objects (called actors) that can invoke methods on each other remotely. Each actor has its own state and a mailbox of incoming methods that are invoked on it. Typically, these methods are executed in the order they arrive. Essentially, actors are distributed versions of objects in object oriented programming. As an example for a distributed actor application let us consider a simple chat service (in fact, WhatsApp, one of the widely used chat services is implemented in Erlang)[1]. The architecture is shown in Fig. 2.3. There is one manager actor that keeps track of the connected users and one client proxy actor per user that forwards messages to the user's device.

The code is as follows. The `listen` function takes as an argument the socket that is going to be used for communication with the clients. It starts the manager actor with the `spawn` call and registers it under the name `client_manager`. Then the program accepts new incoming clients with the `accept` method. For each new client, a new client proxy is started with `spawn` in line 9 and it is registered with the manager in 10. The exclamation mark (`!`) is Erlang's syntax for sending a message to another actor. In this case, the `connect` message is sent to `client_manager` with the argument `Socket`. The `manage_clients` is the main function of the program and the one that the manager actor executes. The state of the actor is in the `Sockets` argument, it is the list of client sockets. When the manager actor was spawned in line 2, it was initialized as the empty list `[]`. Whenever a new client connects, its socket is added to the list (see line 16), upon termination of the client, it is removed (see line 18). If new data is sent to the manager by one of the clients, it will be forwarded to all other clients in line 20, implemented by the `send_data` function.

[1]The example is taken from `http://www.jerith.za.net/writings/erlangsockettut.html`

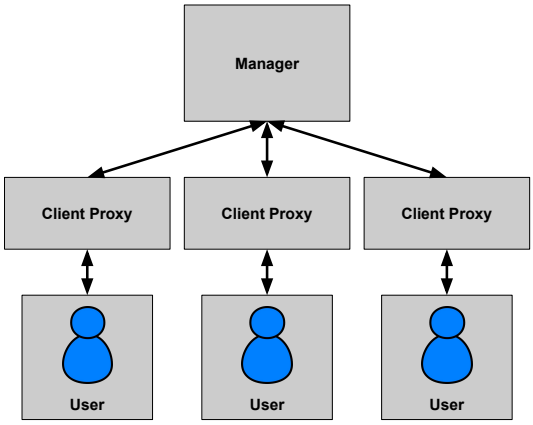


Figure 2.3: A chatroom implementation in the actor framework

```
1  listen(Port) ->
2      Pid = spawn(fun() -> manage_clients([]) end),
3      register(client_manager, Pid),
4      {ok, Listen} = gen_tcp:listen(Port, ?TCP_OPTIONS),
5      accept(Listen).
6
7  accept(Listen) ->
8      {ok, Socket} = gen_tcp:accept(Listen),
9      spawn(fun() -> handle_client(Socket) end),
10     client_manager ! {connect, Socket},
11     accept(Listen).
12
13 manage_clients(Sockets) ->
14     receive
15         {connect, Socket} ->
16             NewSockets = [Socket | Sockets];
17         {disconnect, Socket} ->
18             NewSockets = lists:delete(Socket, Sockets);
19         {data, Data} ->
20             send_data(Sockets, Data),
21             NewSockets = Sockets
22     end,
23     manage_clients(NewSockets).
24
25 send_data(Sockets, Data) ->
26     SendData = fun(Socket) -> gen_tcp:send(Socket, Data) end,
```

```
27      lists:foreach(SendData, Sockets).
```

Note that this example is one very common pattern to write distributed programs, the client server model. It is a special case of communicating processes. In the general case, there is no dedicated server and actors can call methods on each other in arbitrary patterns. Actors can also be made fault tolerant. In Erlang this is done with supervision trees: Actors are organized in a tree and if an actor fails, its parent will be notified and can restart the actor and reset the state [10].

2.4 The Distributed Shared Memory Model

The most powerful distributed programming model is the distributed shared memory model. In an ideal version, it would expose the whole cluster as a single large multicore machine, which can be programmed using multiple execution threads that communicate by reading and writing data from shared memory. In practice this ideal is however not achievable: Latencies of accessing remote memory over the network are typically much larger than latencies of accessing local memory. Therefore if programmers do not take into account the topology of the cluster and just use distributed shared memory in an unstructured fashion, it can lead to very inefficient programs. In addition, distributed shared memory architectures are typically not fault tolerant. However, on specialized supercomputers with special networks, they can lead to very efficient implementations for some workloads. On the cloud, where commodity hardware is typically used, it would be hard to make this programming paradigm successful.

Chapter 3

Motivation: Training Deep Networks in Spark

Optimization is a crucial step in machine learning. It is very computationally expensive and in many cases has to be executed in a distributed fashion to complete in a reasonable time frame. In the context of machine learning, the optimization problem to solve is to find good parameters for a model given some data by minimizing a loss function. For highly unstructured non-convex problems like optimizing the loss function of a deep neural network, first-order optimization algorithms like stochastic gradient descent (SGD) are often the method of choice. We can speed these methods up by distributing the gradient computation over minibatches. This can be quite demanding on the network interconnects because the full model parameters are sent to and from each node in the network on each minibatch update. In this chapter[1], we present an algorithm to run stochastic gradient descent in parallel in the setting where communication is expensive. Instead of communicating the parameters for each minibatch update, we run SGD locally on each node for a few iterations and then average the parameters. While this is a feasible way to train deep neural networks on a slow network, the research performed in this chapter also showed the limitations of data transfer speeds on Spark and served as a motivation to separate the control and data plane for Ray as described in chapter 5.

3.1 Introduction

Deep learning has advanced the state of the art in a number of application domains. Many of the recent advances involve fitting large models (often several hundreds megabytes) to larger datasets (often hundreds of gigabytes). Given the scale of these optimization problems, training can be time-consuming, often requiring multiple days on a single GPU using stochastic gradient descent (SGD). For this reason, much effort has been devoted to leverag-

[1]This material was previously published in [83].

ing the computational resources of a cluster to speed up the training of deep networks (and more generally to perform distributed optimization).

Many attempts to speed up the training of deep networks rely on asynchronous, lock-free optimization [35, 28]. This paradigm uses the parameter server model [67, 52], in which one or more master nodes hold the latest model parameters in memory and serve them to worker nodes upon request. The nodes then compute gradients with respect to these parameters on a minibatch drawn from the local data shard. These gradients are shipped back to the server, which updates the model parameters.

At the same time, batch-processing frameworks enjoy widespread usage and have been gaining in popularity. Beginning with MapReduce [34], a number of frameworks for distributed computing have emerged to make it easier to write distributed programs that leverage the resources of a cluster [141, 57, 88]. These frameworks have greatly simplified many large-scale data analytics tasks. However, state-of-the-art deep learning systems rely on custom implementations to facilitate their asynchronous, communication-intensive workloads. One reason is that popular batch-processing frameworks [34, 141] are not designed to support the workloads of existing deep learning systems. SparkNet implements a scalable, distributed algorithm for training deep networks that lends itself to batch computational frameworks such as MapReduce and Spark and works well out-of-the-box in bandwidth-limited environments.

The benefits of integrating model training with existing batch frameworks are numerous. Much of the difficulty of applying machine learning has to do with obtaining, cleaning, and processing data as well as deploying models and serving predictions. For this reason, it is convenient to integrate model training with the existing data-processing pipelines that have been engineered in today's distributed computational environments. Furthermore, this approach allows data to be kept in memory from start to finish, whereas a segmented approach requires writing to disk between operations. If a user wishes to train a deep network on the output of a SQL query or on the output of a graph computation and to feed the resulting predictions into a distributed visualization tool, this can be done conveniently within a single computational framework.

We emphasize that the hardware requirements of our approach are minimal. Whereas many approaches to the distributed training of deep networks involve heavy communication (often communicating multiple gradient vectors for every minibatch), our approach gracefully handles the bandwidth-limited setting while also taking advantage of clusters with low-latency communication. For this reason, we can easily deploy our algorithm on clusters that are not optimized for communication. Our implementation works well out-of-the box on a five-node EC2 cluster in which broadcasting and collecting model parameters (several hundred megabytes per worker) takes on the order of 20 seconds, and performing a single minibatch gradient computation requires about 2 seconds (for AlexNet). We achieve this by providing a simple algorithm for parallelizing SGD that involves minimal communication and lends itself to straightforward implementation in batch computational frameworks. Our goal is not to outperform custom computational frameworks but rather to propose a system that can be easily implemented in popular batch frameworks and that performs nearly as

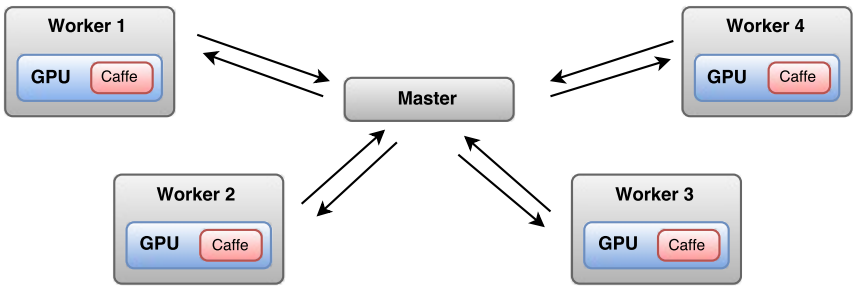


Figure 3.1: This figure depicts the SparkNet architecture.

well as what can be accomplished with specialized frameworks.

3.2 Implementation

Here we describe our implementation of SparkNet. SparkNet builds on Apache Spark [141] and the Caffe deep learning library [58]. In addition, we use Java Native Access for accessing Caffe data and weights natively from Scala, and we use the Java implementation of Google Protocol Buffers to allow the dynamic construction of Caffe networks at runtime.

The Net class wraps Caffe and exposes a simple API containing the methods shown in Listing 3.1. The `NetParams` type specifies a network architecture, and the `WeightCollection` type is a map from layer names to lists of weights. It allows the manipulation of network components and the storage of weights and outputs for individual layers. To facilitate manipulation of data and weights without copying memory from Caffe, we implement the `NDArray` class, which is a lightweight multi-dimensional tensor library. One benefit of building on Caffe is that any existing Caffe model definition or solver file is automatically compatible

```
class Net {
  def Net(netParams: NetParams): Net
  def setTrainingData(data: Iterator[(NDArray,Int)])
  def setValidationData(data: Iterator[(NDArray,Int)])
  def train(numSteps: Int)
  def test(numSteps: Int): Float
  def setWeights(weights: WeightCollection)
  def getWeights(): WeightCollection
}
```

Listing 3.1: SparkNet API

```
val netParams = NetParams(
  RDDLayer("data", shape=List(batchsize, 1, 28, 28)),
  RDDLayer("label", shape=List(batchsize, 1)),
  ConvLayer("conv1", List("data"), kernel=(5,5), numFilters=20),
  PoolLayer("pool1", List("conv1"), pool=Max, kernel=(2,2),
            stride=(2,2)),
  ConvLayer("conv2", List("pool1"), kernel=(5,5), numFilters=50),
  PoolLayer("pool2", List("conv2"), pool=Max, kernel=(2,2),
            stride=(2,2)),
  LinearLayer("ip1", List("pool2"), numOutputs=500),
  ActivationLayer("relu1", List("ip1"), activation=ReLU),
  LinearLayer("ip2", List("relu1"), numOutputs=10),
  SoftmaxWithLoss("loss", List("ip2", "label"))
)
```

Listing 3.2: Example network specification in SparkNet

with SparkNet. There is a large community developing Caffe models and extensions, and these can easily be used in SparkNet. By building on top of Spark, we inherit the advantages of modern batch computational frameworks. These include the high-throughput loading and preprocessing of data and the ability to keep data in memory between operations. In Listing 3.2, we give an example of how network architectures can be specified in SparkNet. In addition, model specifications or weights can be loaded directly from Caffe files. An example sketch of code that uses our API to perform distributed training is given in Listing 3.3.

Parallelizing SGD

To perform well in bandwidth-limited environments, we recommend a parallelization scheme for SGD that requires minimal communication. This approach is not specific to SGD. Indeed, SparkNet works out of the box with any Caffe solver.

The parallelization scheme is described in Listing 3.3. Spark consists of a single master node and a number of worker nodes. The data is split among the Spark workers. In every iteration, the Spark master broadcasts the model parameters to each worker. Each worker then runs SGD on the model with its subset of data for a fixed number of iterations τ (we use $\tau = 50$ in Listing 3.3) or for a fixed length of time, after which the resulting model parameters on each worker are sent to the master and averaged to form the new model parameters. We recommend initializing the network by running SGD for a small number of iterations on the master. A similar and more sophisticated approach to parallelizing SGD with minimal communication overhead is discussed in [142].

The standard approach to parallelizing each gradient computation requires broadcasting

```
var trainData = loadData(...)
var trainData = preprocess(trainData).cache()
var nets = trainData.foreachPartition(data => {
  var net = Net(netParams)
  net.setTrainingData(data)
  net)
var weights = initialWeights(...)
for (i <- 1 to 1000) {
  var broadcastWeights = broadcast(weights)
  nets.map(net => net.setWeights(broadcastWeights.value))
  weights = nets.map(net => {
    net.train(50)
    // an average of WeightCollection objects
    net.getWeights()}).mean()
}
```

Listing 3.3: Distributed training example

and collecting model parameters (hundreds of megabytes per worker and gigabytes in total) after every SGD update, which occurs tens of thousands of times during training. On our EC2 cluster, each broadcast and collection takes about twenty seconds, putting a bound on the speedup that can be expected using this approach without better hardware or without partitioning models across machines. Our approach broadcasts and collects the parameters a factor of τ times less for the same number of iterations. In our experiments, we set $\tau = 50$, but other values seem to work about as well.

We note that Caffe supports parallelism across multiple GPUs within a single node. This is not a competing form of parallelism but rather a complementary one. In some of our experiments, we use Caffe to handle parallelism within a single node, and we use the parallelization scheme described in Listing 3.3 to handle parallelism across nodes.

3.3 Experiments

In Section 3.3, we will benchmark the performance of SparkNet and measure the speedup that our system obtains relative to training on a single node. However, the outcomes of those experiments depend on a number of different factors. In addition to τ (the number of iterations between synchronizations) and K (the number of machines in our cluster), they depend on the communication overhead in our cluster S. In Section 3.3, we find it instructive to measure the speedup *in the idealized case of zero communication overhead* ($S = 0$). This idealized model gives us an upper bound on the maximum speedup that we could hope to

obtain in a real-world cluster, and it allows us to build a model for the speedup as a function of S (the overhead is easily measured in practice).

Theoretical Considerations

Before benchmarking our system, we determine the maximum possible speedup that could be obtained in principle in a cluster with no communication overhead. We determine the dependence of this speedup on the parameters τ (the number of iterations between synchronizations) and K (the number of machines in our cluster).

Limitations of Naive Parallelization

To begin with, we consider the theoretical limitations of a naive parallelism scheme which parallelizes SGD by distributing each minibatch computation over multiple machines (see Figure 3.2b). Let $N_a(b)$ be the number of serial iterations of SGD required to obtain an accuracy of a when training with a batch size of b (when we say accuracy, we are referring to test accuracy). Suppose that computing the gradient over a batch of size b requires $C(b)$ units of time. Then the running time required to achieve an accuracy of a with serial training is

$$N_a(b)C(b). \tag{3.1}$$

A naive parallelization scheme attempts to distribute the computation at each iteration by dividing each minibatch between the K machines, computing the gradients separately, and aggregating the results on one node. Under this scheme, the cost of the computation done on a single node in a single iteration is $C(b/K)$ and satisfies $C(b/K) \geq C(b)/K$ (the cost is sublinear in the batch size). In a system with no communication overhead and no overhead for summing the gradients, this approach could in principle achieve an accuracy of a in time $N_a(b)C(b)/K$. This represents a linear speedup in the number of machines (for values of K up to the batch size b).

In practice, there are several important considerations. First, for the approximation $C(b/K) \approx C(b)/K$ to hold, K must be much smaller than b, limiting the number of machines we can use to effectively parallelize the minibatch computation. One might imagine circumventing this limitation by using a larger batch size b. Unfortunately, the benefit of using larger batches is relatively modest. As the batch size b increases, $N_a(b)$ does not decrease enough to justify the use of a very large value of b.

Furthermore, the benefits of this approach depend greatly on the degree of communication overhead. If aggregating the gradients and broadcasting the model parameters requires S units of time, then the time required by this approach is at least $C(b)/K + S$ per iteration and $N_a(b)(C(b)/K + S)$ to achieve an accuracy of a. Therefore, the maximum achievable speedup is $C(b)/(C(b)/K + S) \leq C(b)/S$. We may expect S to increase modestly as K increases, but we suppress this effect here.

Limitations of SparkNet Parallelization

The performance of the naive parallelization scheme is easily understood because its behavior is equivalent to that of the serial algorithm. In contrast, SparkNet uses a parallelization scheme that is not equivalent to serial SGD (described in Section 3.2), and so its analysis is more complex.

SparkNet's parallelization scheme proceeds in rounds (see Figure 3.2c). In each round, each machine runs SGD for τ iterations with batch size b. Between rounds, the models on the workers are gathered together on the master, averaged, and broadcast to the workers.

We use $M_a(b, K, \tau)$ to denote the number of rounds required to achieve an accuracy of a. The number of parallel iterations of SGD under SparkNet's parallelization scheme required to achieve an accuracy of a is then $\tau M_a(b, K, \tau)$, and the wallclock time is

$$(\tau C(b) + S)M_a(b, K, \tau), \tag{3.2}$$

where S is the time required to gather and broadcast model parameters.

To measure the sensitivity of SparkNet's parallelization scheme to the parameters τ and K, we consider a grid of values of K and τ. For each pair of parameters, we run SparkNet using a modified version of AlexNet on a subset of ImageNet (the first 100 classes each with approximately 1000 data points) for a total of 20000 parallel iterations. For each of these training runs, we compute the ratio $\tau M_a(b, K, \tau)/N_a(b)$. This is the speedup achieved relative to training on a single machine when $S = 0$. In Figure 3.3, we plot a heatmap of the speedup given by the SparkNet parallelization scheme under different values of τ and K.

Figure 3.3 exhibits several trends. The top row of the heatmap corresponds to the case $K = 1$, where we use only one worker. Since we do not have multiple workers to synchronize when $K = 1$, the number of iterations τ between synchronizations does not matter, so all of the squares in the top row of the grid should behave similarly and should exhibit a speedup factor of 1 (up to randomness in the optimization). The rightmost column of each heatmap corresponds to the case $\tau = 1$, where we synchronize after every iteration of SGD. This is equivalent to running serial SGD with a batch size of Kb, where b is the batchsize on each worker (in these experiments we use $b = 100$). In this column, the speedup should increase sublinearly with K. We note that it is slightly surprising that the speedup does not increase monotonically from left to right as τ decreases. Intuitively, we might expect more synchronization to be strictly better (recall we are disregarding the overhead due to synchronization). However, our experiments suggest that modest delays between synchronizations can be beneficial.

This experiment capture the speedup that we can expect from the SparkNet parallelization scheme in the case of zero communication overhead (the numbers are dataset specific, but the trends are of interest). Having measured these numbers, it is straightforward to compute the speedup that we can expect as a function of the communication overhead.

In Figure 3.4, we plot the speedup expected both from naive parallelization and from SparkNet on a five-node cluster as a function of S (normalized so that $C(b) = 1$). As expected, naive parallelization gives a maximum speedup of 5 (on a five-node cluster) when

there is zero communication overhead (note that our plot does not go all the way to $S = 0$), and it gives no speedup when the communication overhead is comparable to or greater than the cost of a minibatch computation. In contrast, SparkNet gives a relatively consistent speedup even when the communication overhead is 100 times the cost of a minibatch computation.

The speedup given by the naive parallelization scheme can be computed exactly and is given by $C(b)/(C(b)/K + S)$. This formula is essentially Amdahl's law. Note that when $S \geq C(b)$, the naive parallelization scheme is slower than the computation on a single machine. The speedup obtained by SparkNet is $N_a(b)C(b)/[(\tau C(b) + S)M_a(b, K, \tau)]$ for a specific value of τ. The numerator is the time required by serial SGD to achieve an accuracy of a from Equation 3.1, and the denominator is the time required by SparkNet to achieve the same accuracy from Equation 3.2. Choosing the optimal value of τ gives us a speedup of $\max_\tau N_a(b)C(b)/[(\tau C(b) + S)M_a(b, K, \tau)]$. In practice, choosing τ is not a difficult problem. The ratio $N_a(b)/(\tau M_a(b, K, \tau))$ (the speedup when $S = 0$) degrades slowly as τ increases, so it suffices to choose τ to be a small multiple of S (say $5S$) so that the algorithm spends only a fraction of its time in communication.

When plotting the SparkNet speedup in Figure 3.4, we do not maximize over all positive integer values of τ but rather over the set $\tau \in \{1, 2, 5, 10, 25, 100, 500, 1000, 2500\}$, and we use the values of $N_a(b)$ and $M_a(b, K, \tau)$ corresponding to the fifth row of Figure 3.3. Including more values of τ would only increase the SparkNet speedup. The distributed training of deep networks is typically thought of as a communication-intensive procedure. However, Figure 3.4 demonstrates the value of SparkNet's parallelization scheme even in the most bandwidth-limited settings.

The naive parallelization scheme may appear to be a straw man. However, it is a frequently-used approach to parallelizing SGD [95, 55], especially when asynchronous updates are not an option (as in computational frameworks like MapReduce and Spark).

Training Benchmarks

To explore the scaling behavior of our algorithm and implementation, we perform experiments on EC2 using clusters of g2.8xlarge nodes. Each node has four NVIDIA GRID GPUs and 60GB memory. We train the default Caffe model of AlexNet [63] on the ImageNet dataset [108]. We run SparkNet with $K = 3$, 5, and 10 and plot the results in Figure 3.5. For comparison, we also run Caffe on the same cluster with a single GPU and no communication overhead to obtain the $K = 1$ plot. These experiments use only a single GPU on each node. To measure the speedup, we compare the wall-clock time required to obtain an accuracy of 45%. With 1 GPU and no communication overhead, this takes 55.6 hours. With 3, 5, and 10 GPUs, SparkNet takes 22.9, 14.5, and 12.8 hours, giving speedups of 2.4, 3.8, and 4.4.

We also train the default Caffe model of GoogLeNet [124] on ImageNet. We run SparkNet with $K = 3$ and $K = 6$ and plot the results in Figure 3.6. In these experiments, we use Caffe's multi-GPU support to take advantage of all four GPUs within each node, and we use

SparkNet's parallelization scheme to handle parallelism across nodes. For comparison, we train Caffe on a single node with four GPUs and no communication overhead. To measure the speedup, we compare the wall-clock time required to obtain an accuracy of 40%. Relative to the baseline of Caffe with four GPUs, SparkNet on 3 and 6 nodes gives speedups of 2.7 and 3.2. Note that this is on top of the speedup of roughly 3.5 that Caffe with four GPUs gets over Caffe with one GPU, so the speedups that SparkNet obtains over Caffe on a single GPU are roughly 9.4 and 11.2.

Furthermore, we explore the dependence of the parallelization scheme described in Section 3.2 on the parameter τ which determines the number of iterations of SGD that each worker does before synchronizing with the other workers. These results are shown in Figure 3.7. Note that in the presence of stragglers, it suffices to replace the fixed number of iterations τ with a fixed length of time, but in our experimental setup, the timing was sufficiently consistent and stragglers did not arise. The single GPU experiment in Figure 3.5 was trained on a single GPU node with no communication overhead.

3.4 Related Work

Much work has been done to build distributed frameworks for training deep networks. [29] build a model-parallel system for training deep networks on a GPU cluster using MPI over Infiniband. [35] build DistBelief, a distributed system capable of training deep networks on thousands of machines using stochastic and batch optimization procedures. In particular, they highlight asynchronous SGD and batch L-BFGS. Distbelief exploits both data parallelism and model parallelism. [28] build Project Adam, a system for training deep networks on hundreds of machines using asynchronous SGD. [67, 52] build parameter servers to exploit model and data parallelism, and though their systems are better suited to sparse gradient updates, they could very well be applied to the distributed training of deep networks. More recently, [75] build TensorFlow, a sophisticated system for training deep networks and more generally for specifying computation graphs and performing automatic differentiation. [55] build FireCaffe, a data-parallel system that achieves impressive scaling using naive parallelization in the high-performance computing setting. They minimize communication overhead by using a tree reduce for aggregating gradients in a supercomputer with Cray Gemini interconnects.

These custom systems have numerous advantages including high performance, fine-grained control over scheduling and task placement, and the ability to take advantage of low-latency communication between machines. On the other hand, due to their demanding communication requirements, they are unlikely to exhibit the same scaling on an EC2 cluster. Furthermore, due to their nature as custom systems, they lack the benefits of tight integration with general-purpose computational frameworks such as Spark. For some of these systems, preprocessing must be done separately by a MapReduce style framework, and data is written to disk between segments of the pipeline. With SparkNet, preprocessing and training are both done in Spark.

Training a machine learning model such as a deep network is often one step of many in real-world data analytics pipelines [121]. Obtaining, cleaning, and preprocessing the data are often expensive operations, as is transferring data between systems. Training data for a machine learning model may be derived from a streaming source, from a SQL query, or from a graph computation. A user wishing to train a deep network in a custom system on the output of a SQL query would need a separate SQL engine. In SparkNet, training a deep network on the output of a SQL query, or a graph computation, or a streaming data source is straightforward due to its general purpose nature and its support for SQL, graph computations, and data streams [9, 47, 138].

Some attempts have been made to train deep networks in general-purpose computational frameworks, however, existing work typically hinges on extremely low-latency intra-cluster communication. [95] train deep networks in Spark on top of YARN using SGD and leverage cluster resources to parallelize the computation of the gradient over each minibatch. To achieve competitive performance, they use remote direct memory accesses over Infiniband to exchange model parameters quickly between GPUs. In contrast, SparkNet tolerates low-bandwidth intra-cluster communication and works out of the box on Amazon EC2.

A separate line of work addresses speeding up the training of deep networks using single-machine parallelism. For example, Caffe con Troll [49] modifies Caffe to leverage both CPU and GPU resources within a single node. These approaches are compatible with SparkNet and the two can be used in conjunction.

Many popular computational frameworks provide support for training machine learning models [76] such as linear models and matrix factorization models. However, due to the demanding communication requirements and the larger scale of many deep learning problems, these libraries have not been extended to include deep networks.

Various authors have studied the theory of averaging separate runs of SGD. In the bandwidth-limited setting, [144] analyze a simple algorithm for convex optimization that is easily implemented in the MapReduce framework and can tolerate high-latency communication between machines. [142] define a parallelization scheme that penalizes divergences between parallel workers, and they provide an analysis in the convex case. [143] propose a general abstraction for parallelizing stochastic optimization algorithms along with a Spark implementation.

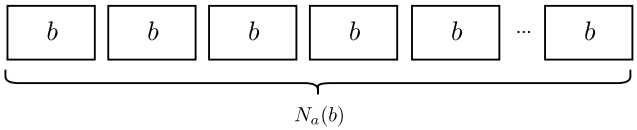
3.5 Discussion

We have described an approach to distributing the training of deep networks in communication-limited environments that lends itself to an implementation in batch computational frameworks like MapReduce and Spark. We provide SparkNet, an easy-to-use deep learning implementation for Spark that is based on Caffe and enables the easy parallelization of existing Caffe models with minimal modification. As machine learning increasingly depends on larger and larger datasets, integration with a fast and general engine for big data processing such as Spark allows researchers and practitioners to draw from a rich ecosystem of tools to develop

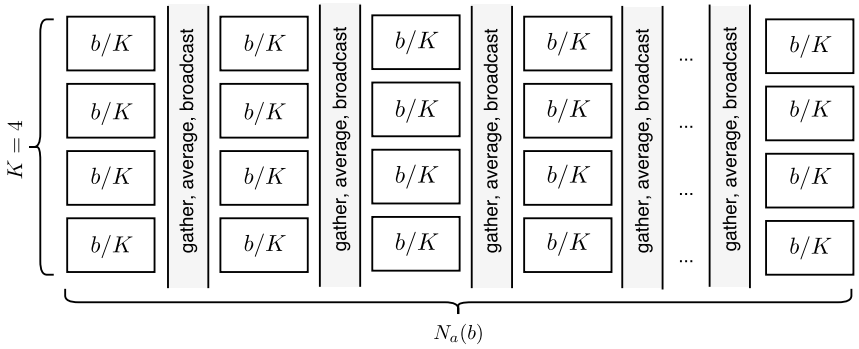
and deploy their models. They can build models that incorporate features from a variety of data sources like images on a distributed file system, results from a SQL query or graph database query, or streaming data sources.

Using a smaller version of the ImageNet benchmark we quantify the speedup achieved by SparkNet as a function of the size of the cluster, the communication frequency, and the cluster's communication overhead. We demonstrate that our approach is effective even in highly bandwidth-limited settings. On the full ImageNet benchmark we showed that our system achieves a sizable speedup over a single node experiment even with few GPUs.

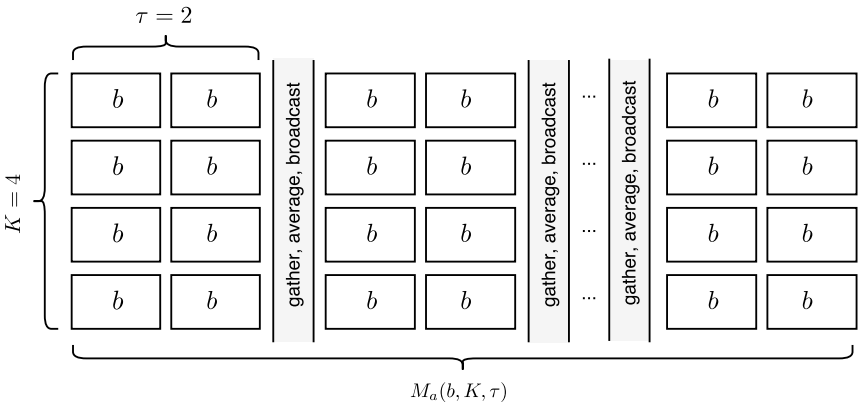
The code for SparkNet is available at `https://github.com/amplab/SparkNet`. We invite contributions and hope that the project will help bring a diverse set of deep learning applications to the Spark community.



(a) This figure depicts a serial run of SGD. Each block corresponds to a single SGD update with batch size b. The quantity $N_a(b)$ is the number of iterations required to achieve an accuracy of a.



(b) This figure depicts a parallel run of SGD on $K = 4$ machines under a naive parallelization scheme. At each iteration, each batch of size b is divided among the K machines, the gradients over the subsets are computed separately on each machine, the updates are aggregated, and the new model is broadcast to the workers. Algorithmically, this approach is exactly equivalent to the serial run of SGD in Figure 3.2a and so the number of iterations required to achieve an accuracy of a is the same value $N_a(b)$.



(c) This figure depicts a parallel run of SGD on $K = 4$ machines under SparkNet's parallelization scheme. At each step, each machine runs SGD with batch size b for τ iterations, after which the models are aggregated, averaged, and broadcast to the workers. The quantity $M_a(b, K, \tau)$ is the number of rounds (of τ iterations) required to obtain an accuracy of a. The total number of parallel iterations of SGD under SparkNet's parallelization scheme required to obtain an accuracy of a is then $\tau M_a(b, K, \tau)$.

Figure 3.2: Computational models for different parallelization schemes.

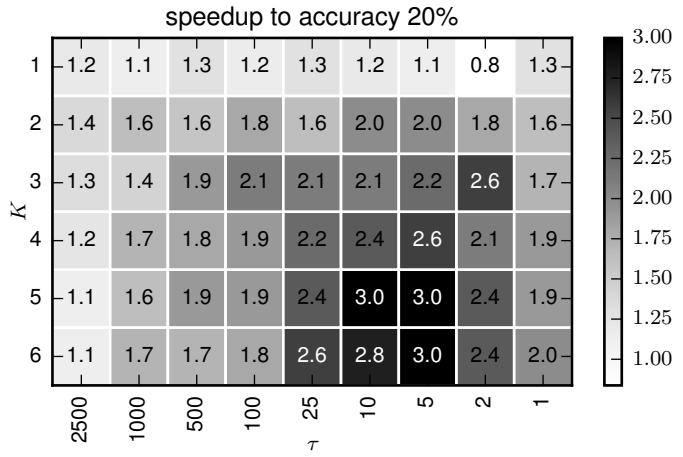


Figure 3.3: This figure shows the speedup $\tau M_a(b, \tau, K)/N_a(b)$ given by SparkNet's parallelization scheme relative to training on a single machine to obtain an accuracy of $a = 20\%$. Each grid square corresponds to a different choice of K and τ. We show the speedup in the zero communication overhead setting. This experiment uses a modified version of AlexNet on a subset of ImageNet (100 classes each with approximately 1000 images). Note that these numbers are dataset specific. Nevertheless, the trends they capture are of interest.

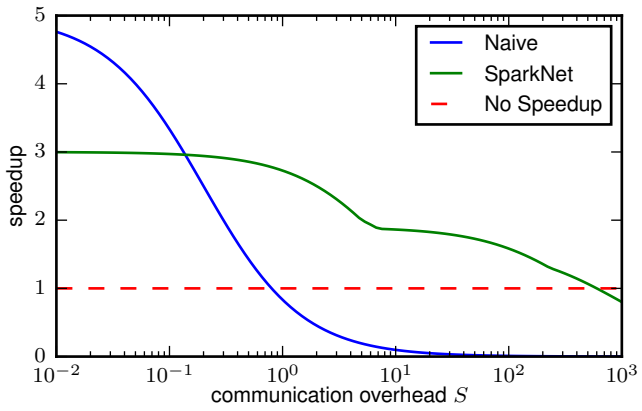


Figure 3.4: This figure shows the speedups obtained by the naive parallelization scheme and by SparkNet as a function of the cluster's communication overhead (normalized so that $C(b) = 1$). We consider $K = 5$. The data for this plot applies to training a modified version of AlexNet on a subset of ImageNet (approximately 1000 images for each of the first 100 classes). The speedup obtained by the naive parallelization scheme is $C(b)/(C(b)/K + S)$. The speedup obtained by SparkNet is $N_a(b)C(b)/[(\tau C(b)+S)M_a(b, K, \tau)]$ for a specific value of τ. The numerator is the time required by serial SGD to achieve an accuracy of a, and the denominator is the time required by SparkNet to achieve the same accuracy (see Equation 3.1 and Equation 3.2). For the optimal value of τ, the speedup is $\max_\tau N_a(b)C(b)/[(\tau C(b) + S)M_a(b, K, \tau)]$. To plot the SparkNet speedup curve, we maximize over the set of values $\tau \in \{1, 2, 5, 10, 25, 100, 500, 1000, 2500\}$ and use the values $M_a(b, K, \tau)$ and $N_a(b)$ from the experiments in the fifth row of Figure 3.3. In our experiments, we have $S \approx 20s$ and $C(b) \approx 2s$.

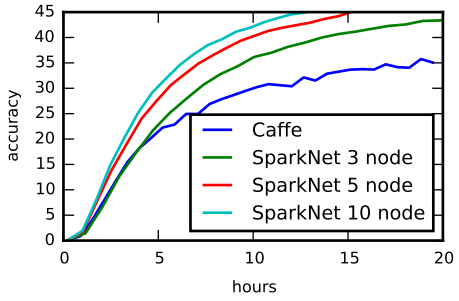


Figure 3.5: This figure shows the performance of SparkNet on a 3-node, 5-node, and 10-node cluster, where each node has 1 GPU. In these experiments, we use $\tau = 50$. The baseline was obtained by running Caffe on a single GPU with no communication. The experiments are performed on ImageNet using AlexNet.

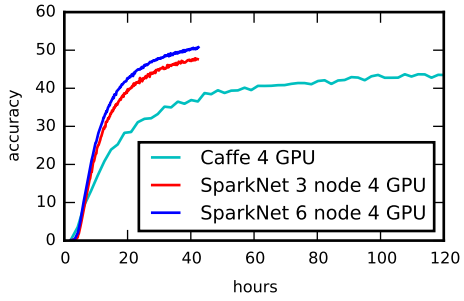


Figure 3.6: This figure shows the performance of SparkNet on a 3-node cluster and on a 6-node cluster, where each node has 4 GPUs. In these experiments, we use $\tau = 50$. The baseline uses Caffe on a single node with 4 GPUs and no communication overhead. The experiments are performed on ImageNet using GoogLeNet.

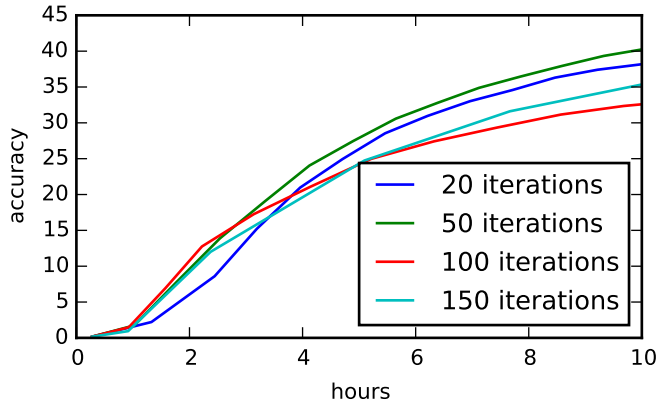


Figure 3.7: This figure shows the dependence of the parallelization scheme described in Section 3.2 on τ. Each experiment was run with $K = 5$ workers. This figure shows that good performance can be achieved without collecting and broadcasting the model after every SGD update.

Chapter 4

The System Requirements

As we have already seen in chapter 2, there are many different programming models for distributed computing. In this chapter[1] we study the requirements of a general purpose distributed system that can support emerging artificial intelligence applications like reinforcement learning both in terms of the programming model and the system implementation.

The landscape of machine learning (ML) applications is undergoing a significant change. While ML has predominantly focused on training and serving predictions based on static models (Figure 4.1a), there is now a strong shift toward the tight integration of ML models in feedback loops. Indeed, ML applications are expanding from the supervised learning paradigm, in which static models are trained on offline data, to a broader paradigm, exemplified by reinforcement learning (RL), in which applications may operate in real environments, fuse and react to sensory data from numerous input streams, perform continuous micro-simulations, and close the loop by taking actions that affect the sensed environment (Figure 4.1b).

Since learning by interacting with the real world can be unsafe, impractical, or bandwidth-limited, many reinforcement learning systems rely heavily on **simulating physical or virtual environments**. Simulations may be used during training (e.g., to learn a neural network policy), and during deployment. In the latter case, we may constantly update the simulated environment as we interact with the real world and perform many simulations to figure out the next action (e.g., using online planning algorithms like Monte Carlo tree search). This requires the ability to perform simulations faster than real time.

Such emerging applications require new levels of programming flexibility and performance. Meeting these requirements without losing the benefits of modern distributed execution frameworks (e.g., application-level fault tolerance) poses a significant challenge. Our own experience implementing ML and RL applications in Spark, MPI, and TensorFlow highlights some of these challenges and gives rise to three groups of requirements for supporting these applications. *Though these requirements are critical for ML and RL applications, we believe they are broadly useful.*

[1] This material was previously published in [93].

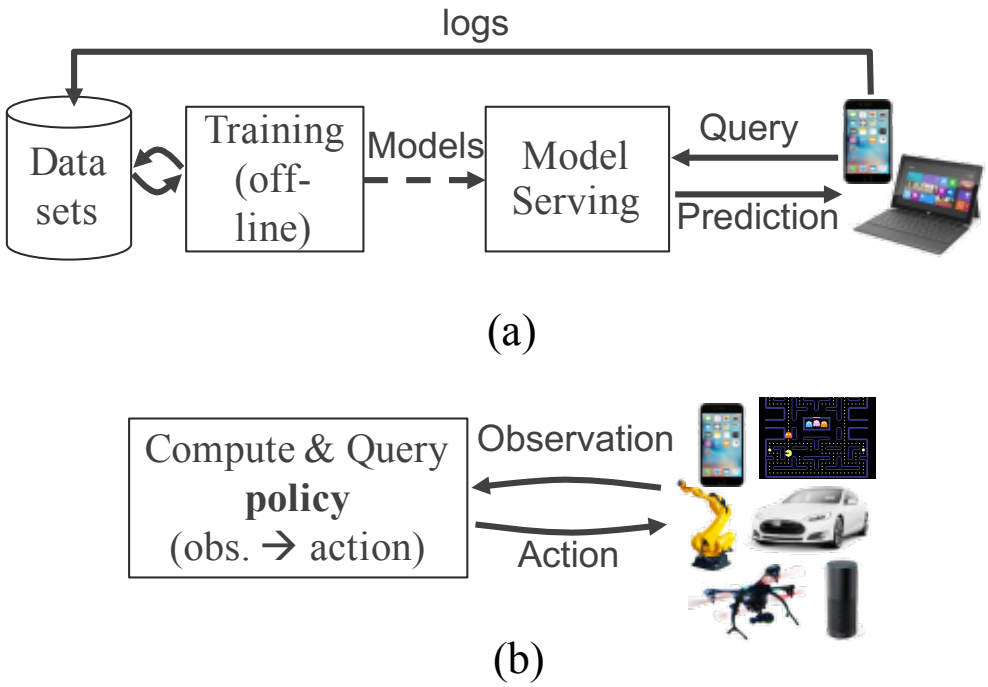


Figure 4.1: (a) Traditional ML pipeline (off-line training). (b) Example reinforcement learning pipeline: the system continously interacts with an environment to learn a policy, i.e., a mapping between observations and actions.

Performance Requirements. Emerging ML applications have stringent latency and throughput requirements.

- **R1:** *Low latency.* The real-time, reactive, and interactive nature of emerging ML applications calls for fine-granularity task execution with millisecond end-to-end latency [31].
- **R2:** *High throughput.* The volume of micro-simulations required both for training [90] as well as for inference during deployment [119] necessitates support for high-throughput task execution on the order of millions of tasks per second.

Execution Model Requirements. Though many existing parallel execution systems [33, 137] have gotten great mileage out of identifying and optimizing for common computational patterns, emerging ML applications require far greater flexibility [40].

- **R3:** *Dynamic task creation.* RL primitives such as Monte Carlo tree search may generate new tasks during execution based on the results or the durations of other tasks.
- **R4:** *Heterogeneous tasks.* Deep learning primitives and RL simulations produce tasks with widely different execution times and resource requirements. Explicit system support for heterogeneity of tasks and resources is essential for RL applications.

- **R5:** *Arbitrary dataflow dependencies.* Similarly, deep learning primitives and RL simulations produce arbitrary and often fine-grained task dependencies (not restricted to bulk synchronous parallel).

Practical Requirements.

- **R6:** *Transparent fault tolerance.* Fault tolerance remains a key requirement for many deployment scenarios, and supporting it alongside high-throughput and non-deterministic tasks poses a challenge.
- **R7:** *Debuggability and Profiling.* Debugging and performance profiling are the most time-consuming aspects of writing any distributed application. This is especially true for ML and RL applications, which are often compute-intensive and stochastic.

Existing frameworks fall short of achieving one or more of these requirements (Section 4.4). We propose a flexible distributed programming model (Section 5.2) to enable **R3-R5**. In addition, we propose a system architecture to support this programming model and meet our performance requirements (**R1-R2**) without giving up key practical requirements (**R6-R7**). The proposed system architecture (Section 5.3) builds on two principal components: a logically-centralized control plane and a hybrid scheduler. The former enables stateless distributed components and lineage replay. The latter allocates resources in a bottom-up fashion, splitting locally-born work between node-level and cluster-level schedulers.

The result is millisecond-level performance on microbenchmarks and a 63x end-to-end speedup on a representative RL application over a bulk synchronous parallel (BSP) implementation.

4.1 Motivating Example

To motivate requirements **R1-R7**, consider a hypothetical application in which a physical robot attempts to achieve a goal in an unfamiliar real-world environment. Various sensors may fuse video and LIDAR input to build multiple candidate models of the robot's environment (Fig. 2a). The robot is then controlled in real time using actions informed by a recurrent neural network (RNN) *policy* (Fig. 2c), as well as by Monte Carlo tree search (MCTS) and other online planning algorithms (Fig. 2b). Using a physics simulator along with the most recent environment models, MCTS tries millions of action sequences in parallel, adaptively exploring the most promising ones.

The Application Requirements. Enabling these kinds of applications involves simultaneously solving a number of challenges. In this example, the latency requirements (**R1**) are stringent, as the robot must be controlled in real time. High task throughput (**R2**) is needed to support the online simulations for MCTS as well as the streaming sensory input.

Task heterogeneity (**R4**) is present on many scales: some tasks run physics simulators, others process diverse data streams, and some compute actions using RNN-based policies.

Even similar tasks may exhibit substantial variability in duration. For example, the RNN consists of different functions for each "layer", each of which may require different amounts of computation. Or, in a task simulating the robot's actions, the simulation length may depend on whether the robot achieves its goal or not.

In addition to the heterogeneity of tasks, the dependencies between tasks can be complex (**R5**, Figs. 2a and 2c) and difficult to express as batched BSP stages.

Dynamic construction of tasks and their dependencies (**R3**) is critical. Simulations will adaptively use the most recent environment models as they become available, and MCTS may choose to launch more tasks exploring particular subtrees, depending on how promising they are or how fast the computation is. Thus, the dataflow graph must be constructed dynamically in order to allow the algorithm to adapt to real-time constraints and opportunities.

4.2 Proposed Solution

In this section, we outline a proposal for a distributed execution framework and a programming model satisfying requirements **R1-R7** for real-time ML applications.

API and Execution Model

In order to support the execution model requirements (**R3-R5**), we outline an API that allows arbitrary functions to be specified as remotely executable tasks, with dataflow dependencies between them.

1. Task creation is non-blocking. When a *task* is created, a *future* [13] representing the eventual return value of the task is returned immediately, and the task is executed asynchronously.

2. Arbitrary function invocation can be designated as a remote task, making it possible to support arbitrary execution kernels (**R4**). Task arguments can be either regular values or futures. When an argument is a future, the newly created task becomes dependent on the task that produces that future, enabling arbitrary DAG dependencies (**R5**).

3. Any task execution can create new tasks without blocking on their completion. Task throughput is therefore not limited by the bandwidth of any one worker (**R2**), and the computation graph is dynamically built (**R3**).

4. The actual return value of a task can be obtained by calling the `get` method on the corresponding future. This blocks until the task finishes executing.

5. The `wait` method takes a list of futures, a timeout, and a number of values. It returns the subset of futures whose tasks have completed when the timeout occurs or the requested number have completed.

The `wait` primitive allows developers to specify latency requirements (**R1**) with a timeout, accounting for arbitrarily sized tasks (**R4**). This is important for ML applications, in which a straggler task may produce negligible algorithmic improvement but block the entire computation. This primitive enhances our ability to dynamically modify the computation graph as a function of execution-time properties (**R3**).

To complement the fine-grained programming model, we propose using a dataflow execution model in which tasks become available for execution if and only if their dependencies have finished executing.

Proposed Architecture

Our proposed architecture consists of multiple *worker* processes running on each node in the cluster, one *local scheduler* per node, one or more *global schedulers* throughout the cluster, and an in-memory *object store* for sharing data between workers (see Figure 5.5).

The two principal architectural features that enable **R1-R7** are a *hybrid scheduler* and a *centralized control plane*.

Centralized Control State

As shown in Figure 5.5, our architecture relies on a logically-centralized control plane [62]. To realize this architecture, we use a database that provides both (1) storage for the system's control state, and (2) publish-subscribe functionality to enable various system components to communicate with each other.[2]

This design enables virtually any component of the system, except for the database, to be stateless. This means that as long as the database is fault-tolerant, we can recover from component failures by simply restarting the failed components. Furthermore, the database stores the computation lineage, which allows us to reconstruct lost data by replaying the computation [137]. As a result, this design is fault tolerant (**R6**). The database also makes it easy to write tools to profile and inspect the state of the system (**R7**).

To achieve the throughput requirement (**R2**), we shard the database. Since we require only exact matching operations and since the keys are computed as hashes, sharding is relatively straightforward. Our early experiments show that this design enables sub-millisecond scheduling latencies (**R1**).

Hybrid Scheduling

Our requirements for latency (**R1**), throughput (**R2**), and dynamic graph construction (**R3**) naturally motivate a hybrid scheduler in which local schedulers assign tasks to workers or delegate responsibility to one or more global schedulers.

[2]In our implementation we employ Redis [110], although many other fault-tolerant key-value stores could be used.

Workers submit tasks to their local schedulers which decide to either assign the tasks to other workers on the same physical node or to "spill over" the tasks to a global scheduler. Global schedulers can then assign tasks to local schedulers based on global information about factors including object locality and resource availability.

Since tasks may create other tasks, schedulable work may come from any worker in the cluster. Enabling any local scheduler to handle locally generated work without involving a global scheduler improves low latency (**R1**), by avoiding communication overheads, and throughput (**R2**), by significantly reducing the global scheduler load. This hybrid scheduling scheme fits well with the recent trend toward large multicore servers [134].

4.3 Feasibility

To demonstrate that these API and architectural proposals could in principle support requirements **R1-R7**, we provide some simple examples using the preliminary system design outlined in Section 4.2.

Latency Microbenchmarks

Using our prototype system, a task can be created, meaning that the task is submitted asynchronously for execution and a future is returned, in around 35μs. Once a task has finished executing, its return value can be retrieved in around 110μs. The end-to-end time, from submitting an empty task for execution to retrieving its return value, is around 290μs when the task is scheduled locally and 1ms when the task is scheduled on a remote node.

Reinforcement Learning

We implement a simple workload in which an RL agent is trained to play an Atari game. The workload alternates between stages in which actions are taken in parallel simulations and actions are computed in parallel on GPUs. Despite the BSP nature of the example, an implementation in Spark is **9x** slower than the single-threaded implementation due to system overhead. An implementation in our prototype is **7x** faster than the single-threaded version and **63x** faster than the Spark implementation.[3]

This example exhibits two key features. First, tasks are very small (around 7ms each), making low task overhead critical. Second, the tasks are heterogeneous in duration and in resource requirements (e.g., CPUs and GPUs).

This example is just one component of an RL workload, and would typically be used as a subroutine of a more sophisticated (non-BSP) workload. For example, using the `wait` primitive, we can adapt the example to process the simulation tasks in the order that they finish so as to better pipeline the simulation execution with the action computations on the

[3]In this comparison, the GPU model fitting could not be naturally parallelized on Spark, so the numbers are reported as if it had been perfectly parallelized with no overhead in Spark.

GPU, or run the entire workload nested within a larger adaptive hyperparameter search. These changes are all straightforward using the API described in Section 5.2 and involve a few extra lines of code.

4.4 Related Work

Static dataflow systems [33, 137, 56, 86] are well-established in analytics and ML, but they require the dataflow graph to be specified upfront, e.g., by a driver program. Some, like MapReduce [33] and Spark [137], emphasize BSP execution, while others, like Dryad [56] and Naiad [86], support complex dependency structures (**R5**). Others, such as TensorFlow [1] and MXNet [27], are optimized for deep-learning workloads. However, none of these systems fully support the ability to dynamically extend the dataflow graph in response to both input data and task progress (**R3**).

Dynamic dataflow systems like CIEL [84] and Dask [106] support many of the same features as static dataflow systems, with additional support for dynamic task creation **(R3)**. These systems meet our execution model requirements (**R3-R5**). However, their architectural limitations, such as entirely centralized scheduling, are such that low latency (**R1**) must often be traded off with high throughput (**R2**) (e.g., via batching), whereas our applications require both.

Other systems like Open MPI [44] and actor-model variants Orleans [22] and Erlang [11] provide low-latency (**R1**) and high-throughput (**R2**) distributed computation. Though these systems do in principle provide primitives for supporting our execution model requirements (**R3-R5**) and have been used for ML [29, 6], much of the logic required for systems-level features, such as fault tolerance (**R6**) and locality-aware task scheduling, must be implemented at the application level.

4.5 Conclusion

Machine learning applications are evolving to require dynamic dataflow parallelism with millisecond latency and high throughput, posing a severe challenge for existing frameworks. We outline the requirements for supporting this emerging class of real-time ML applications, and we propose a programming model and architectural design to address the key requirements (**R1-R5**), without compromising existing requirements (**R6-R7**). Preliminary, proof-of-concept results confirm millisecond-level system overheads and meaningful speedups for a representative RL application.

Chapter 5

The Design and Implementation of Ray

This chapter[1] contains the main contribution of the thesis. We describe the design and implementation of Ray, a system for distributed computing that satisfies the requirements described in chapter 4.

Over the past two decades, many organizations have been collecting—and aiming to exploit—ever-growing quantities of data. This has led to the development of a plethora of frameworks for distributed data analysis, including batch [33, 140, 56], streaming [24, 87, 64], and graph [72, 74, 46] processing systems. The success of these frameworks has made it possible for organizations to analyze large data sets as a core part of their business or scientific strategy, and has ushered in the age of "Big Data."

More recently, the scope of data-focused applications has expanded to encompass more complex artificial intelligence (AI) or machine learning (ML) techniques [60]. The paradigm case is that of *supervised learning*, where data points are accompanied by labels, and where the workhorse technology for mapping data points to labels is provided by deep neural networks. The complexity of these deep networks has led to another flurry of frameworks that focus on the training of deep neural networks and their use in prediction. These frameworks often leverage specialized hardware (e.g., GPUs and TPUs), with the goal of reducing training time in a batch setting. Examples include TensorFlow [1], MXNet [27], and PyTorch [101].

The promise of AI is, however, far broader than classical supervised learning. Emerging AI applications must increasingly operate in dynamic environments, react to changes in the environment, and take sequences of actions to accomplish long-term goals [2, 93]. They must aim not only to exploit the data gathered, but also to explore the space of possible actions. These broader requirements are naturally framed within the paradigm of *reinforcement learning* (RL). RL deals with learning to operate continuously within an uncertain environment based on delayed and limited feedback [123]. RL-based systems have already

[1]This material was previously published in [82]

yielded remarkable results, such as Google's AlphaGo beating a human world champion [119], and are beginning to find their way into dialogue systems, UAVs [92], and robotic manipulation [48, 130].

The central goal of an RL application is to learn a policy—a mapping from the state of the environment to a choice of action—that yields effective performance over time, e.g., winning a game or piloting a drone. Finding effective policies in large-scale applications requires three main capabilities. First, RL methods often rely on *simulation* to evaluate policies. Simulations make it possible to explore many different choices of action sequences and to learn about the long-term consequences of those choices. Second, like their supervised learning counterparts, RL algorithms need to perform *distributed training* to improve the policy based on data generated through simulations or interactions with the physical environment. Third, policies are intended to provide solutions to control problems, and thus it is necessary to *serve* the policy in interactive closed-loop and open-loop control scenarios.

These characteristics drive new systems requirements: a system for RL must support *fine-grained* computations (e.g., rendering actions in milliseconds when interacting with the real world, and performing vast numbers of simulations), must support *heterogeneity* both in time (e.g., a simulation may take milliseconds or hours) and in resource usage (e.g., GPUs for training and CPUs for simulations), and must support *dynamic* execution, as results of simulations or interactions with the environment can change future computations. Thus, we need a dynamic computation framework that handles millions of heterogeneous tasks per second at millisecond-level latencies.

Existing frameworks that have been developed for Big Data workloads or for supervised learning workloads fall short of satisfying these new requirements for RL. Bulk-synchronous parallel systems such as MapReduce [33], Apache Spark [140], and Dryad [56] do not support fine-grained simulation or policy serving. Task-parallel systems such as CIEL [85] and Dask [106] provide little support for distributed training and serving. The same is true for streaming systems such as Naiad [87] and Storm [64]. Distributed deep-learning frameworks such as TensorFlow [1] and MXNet [27] do not naturally support simulation and serving. Finally, model-serving systems such as TensorFlow Serving [125] and Clipper [30] support neither training nor simulation.

While in principle one could develop an end-to-end solution by stitching together several existing systems (e.g., Horovod [116] for distributed training, Clipper [30] for serving, and CIEL [85] for simulation), in practice this approach is untenable due to the *tight coupling* of these components within applications. As a result, researchers and practitioners today build one-off systems for specialized RL applications [127, 90, 119, 96, 109, 97]. This approach imposes a massive systems engineering burden on the development of distributed applications by essentially pushing standard systems challenges like scheduling, fault tolerance, and data movement onto each application.

In this main chapter of the thesis, we propose Ray, a general-purpose cluster-computing framework that enables simulation, training, and serving for RL applications. The requirements of these workloads range from lightweight and stateless computations, such as for simulation, to long-running and stateful computations, such as for training. To satisfy these

requirements, Ray implements a unified interface that can express both *task-parallel* and *actor-based* computations. *Tasks* enable Ray to efficiently and dynamically load balance simulations, process large inputs and state spaces (e.g., images, video), and recover from failures. In contrast, *actors* enable Ray to efficiently support stateful computations, such as model training, and expose shared mutable state to clients, (e.g., a parameter server). Ray implements the actor and the task abstractions on top of a single dynamic execution engine that is highly scalable and fault tolerant.

To meet the performance requirements, Ray distributes two components that are typically centralized in existing frameworks [140, 56, 85]: (1) the task scheduler and (2) a metadata store which maintains the computation lineage and a directory for data objects. This allows Ray to schedule millions of tasks per second with millisecond-level latencies. Furthermore, Ray provides lineage-based fault tolerance for tasks and actors, and replication-based fault tolerance for the metadata store.

While Ray supports serving, training, and simulation in the context of RL applications, this does not mean that it should be viewed as a replacement for systems that provide solutions for these workloads in other contexts. In particular, Ray does not aim to substitute for serving systems like Clipper [30] and TensorFlow Serving [125], as these systems address a broader set of challenges in deploying models, including model management, testing, and model composition. Similarly, despite its flexibility, Ray is not a substitute for generic data-parallel frameworks, such as Spark [140], as it currently lacks the rich functionality and APIs (e.g., straggler mitigation, query optimization) that these frameworks provide.

We make the following contributions:

- We design and build the first distributed framework that unifies training, simulation, and serving—necessary components of emerging RL applications.
- To support these workloads, we unify the actor and task-parallel abstractions on top of a dynamic task execution engine.
- To achieve scalability and fault tolerance, we propose a system design principle in which control state is stored in a sharded metadata store and all other system components are stateless.
- To achieve scalability, we propose a bottom-up distributed scheduling strategy.

5.1 Motivation and Requirements

We begin by considering the basic components of an RL system and fleshing out the key requirements for Ray. As shown in Figure 5.1, in an RL setting, an *agent* interacts repeatedly with the *environment*. The goal of the agent is to learn a policy that maximizes a *reward*. A *policy* is a mapping from the state of the environment to a choice of *action*. The precise definitions of environment, agent, state, action, and reward are application-specific.

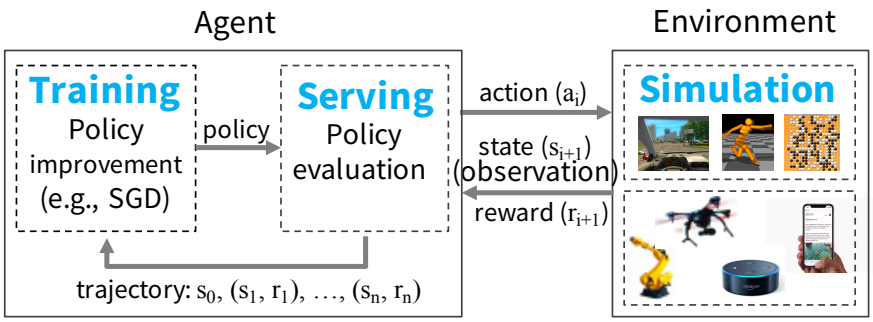


Figure 5.1: Example of an RL system.

```
// evaluate policy by interacting with env. (e.g., simulator)
rollout(policy, environment):
    trajectory = []
    state = environment.initial_state()
    while (not environment.has_terminated()):
        action = policy.compute(state) // Serving
        state, reward = environment.step(action) // Simulation
        trajectory.append(state, reward)
    return trajectory

// improve policy iteratively until it converges
train_policy(environment):
    policy = initial_policy()
    while (policy has not converged):
        trajectories = []
        for i from 1 to k:
            // evaluate policy by generating k rollouts
            trajectories.append(rollout(policy, environment))
            // improve policy
            policy = policy.update(trajectories) // Training
    return policy
```

Figure 5.2: Typical RL pseudocode for learning a policy.

To learn a policy, an agent typically employs a two-step process: (1) *policy evaluation* and (2) *policy improvement*. To evaluate the policy, the agent interacts with the environment (e.g., with a simulation of the environment) to generate *trajectories*, where a trajectory consists of a sequence of (state, reward) tuples produced by the current policy. Then, the agent uses these trajectories to improve the policy; i.e., to update the policy in the direction of the gradient that maximizes the reward. Figure 5.2 shows an example of the pseudocode used by an agent to learn a policy. This pseudocode evaluates the policy by invoking **rollout**(*environment*, *policy*) to generate trajectories. **train_policy()** then uses these trajectories to improve the current policy via *policy*.**update**(*trajectories*). This process repeats until the policy converges.

Thus, a framework for RL applications must provide efficient support for *training*, *serving*, and *simulation* (Figure 5.1). Next, we briefly describe these workloads.

Training typically involves running stochastic gradient descent (SGD), often in a distributed setting, to update the policy. Distributed SGD typically relies on an allreduce aggregation step or a parameter server [66].

Serving uses the trained policy to render an action based on the current state of the environment. A serving system aims to minimize latency, and maximize the number of decisions per second. To scale, load is typically balanced across multiple nodes serving the policy.

Finally, most existing RL applications use *simulations* to evaluate the policy—current RL algorithms are not sample-efficient enough to rely solely on data obtained from interactions with the physical world. These simulations vary widely in complexity. They might take a few ms (e.g., simulate a move in a chess game) to minutes (e.g., simulate a realistic environment for a self-driving car).

In contrast with supervised learning, in which training and serving can be handled separately by different systems, in RL *all three of these workloads are tightly coupled in a single application*, with stringent latency requirements between them. Currently, no framework supports this coupling of workloads. In theory, multiple specialized frameworks could be stitched together to provide the overall capabilities, but in practice, the resulting data movement and latency between systems is prohibitive in the context of RL. As a result, researchers and practitioners have been building their own one-off systems.

This state of affairs calls for the development of new distributed frameworks for RL that can efficiently support training, serving, and simulation. In particular, such a framework should satisfy the following requirements:

Fine-grained, heterogeneous computations. The duration of a computation can range from milliseconds (e.g., taking an action) to hours (e.g., training a complex policy). Additionally, training often requires heterogeneous hardware (e.g., CPUs, GPUs, or TPUs).

Flexible computation model. RL applications require both stateless and stateful computations. Stateless computations can be executed on any node in the system, which makes it easy to achieve load balancing and movement of computation to data, if needed. Thus stateless computations are a good fit for fine-grained simulation and data processing, such as extracting features from images or videos. In contrast stateful computations are a good

Name	Description
futures = **f.remote**(*args*)	Execute function f remotely. **f.remote()** can take objects or futures as inputs and returns one or more futures. This is non-blocking.
objects = **ray.get**(*futures*)	Return the values associated with one or more futures. This is blocking.
ready_futures = **ray.wait(*futures*, *k*, *timeout*)**	Return the futures whose corresponding tasks have completed as soon as either k have completed or the timeout expires.
actor = **Class.remote(*a*rgs)** *futures* = *actor*.**method.remote(*args*)**	Instantiate class *Class* as a remote actor, and return a handle to it. Call a method on the remote actor and return one or more futures. Both are non-blocking.

Table 5.1: Ray API

fit for implementing parameter servers, performing repeated computation on GPU-backed data, or running third-party simulators that do not expose their state.

Dynamic execution. Several components of RL applications require dynamic execution, as the order in which computations finish is not always known in advance (e.g., the order in which simulations finish), and the results of a computation can determine future computations (e.g., the results of a simulation will determine whether we need to perform more simulations).

We make two final comments. First, to achieve high utilization in large clusters, such a framework must handle *millions of tasks per second.*[2] Second, such a framework is not intended for implementing deep neural networks or complex simulators from scratch. Instead, it should enable seamless integration with existing simulators [21, 14, 128] and deep learning frameworks [1, 27, 101, 58].

[2]Assume 5ms single-core tasks and a cluster of 200 32-core nodes. This cluster can run $(1s/5ms) \times 32 \times 200 = 1.28$M tasks/sec.

Tasks (stateless)	Actors (stateful)
Fine-grained load balancing	Coarse-grained load balancing
Support for object locality	Poor locality support
High overhead for small updates	Low overhead for small updates
Efficient failure handling	Overhead from checkpointing

Table 5.2: Tasks vs. actors tradeoffs.

5.2 Programming and Computation Model

Ray implements a dynamic task graph computation model, i.e., it models an application as a graph of dependent tasks that evolves during execution. On top of this model, Ray provides both an actor and a task-parallel programming abstraction. This unification differentiates Ray from related systems like CIEL, which only provides a task-parallel abstraction, and from Orleans [22] or Akka [4], which primarily provide an actor abstraction.

Programming Model

Tasks. A *task* represents the execution of a remote function on a stateless worker. When a remote function is invoked, a *future* representing the result of the task is returned immediately. Futures can be retrieved using **ray.get()** and passed as arguments into other remote functions without waiting for their result. This allows the user to express parallelism while capturing data dependencies. Table 5.1 shows Ray's API.

Remote functions operate on immutable objects and are expected to be *stateless* and side-effect free: their outputs are determined solely by their inputs. This implies idempotence, which simplifies fault tolerance through function re-execution on failure.

Actors. An *actor* represents a stateful computation. Each actor exposes methods that can be invoked remotely and are executed serially. A method execution is similar to a task, in that it executes remotely and returns a future, but differs in that it executes on a *stateful* worker. A *handle* to an actor can be passed to other actors or tasks, making it possible for them to invoke methods on that actor.

Table 5.2 summarizes the properties of tasks and actors. Tasks enable fine-grained load balancing through leveraging load-aware scheduling at task granularity, input data locality, as each task can be scheduled on the node storing its inputs, and low recovery overhead, as there is no need to checkpoint and recover intermediate state. In contrast, actors provide much more efficient fine-grained updates, as these updates are performed on internal rather than external state, which typically requires serialization and deserialization. For example, actors can be used to implement parameter servers [66] and GPU-based iterative computations (e.g., training). In addition, actors can be used to wrap third-party simulators and other opaque handles that are hard to serialize.

To satisfy the requirements for heterogeneity and flexibility (Section 5.1), we augment the API in three ways. First, to handle concurrent tasks with heterogeneous durations, we introduce **ray.wait()**, which waits for the first k available results, instead of waiting for *all* results like **ray.get()**. Second, to handle resource-heterogeneous tasks, we enable developers to specify resource requirements so that the Ray scheduler can efficiently manage resources. Third, to improve flexibility, we enable *nested remote functions*, meaning that remote functions can invoke other remote functions. This is also critical for achieving high scalability (Section 5.3), as it enables multiple processes to invoke remote functions in a distributed fashion.

```python
@ray.remote
def create_policy():
  # Initialize the policy randomly.
  return policy

@ray.remote(num_gpus=1)
class Simulator(object):
  def __init__(self):
    # Initialize the environment.
    self.env = Environment()
  def rollout(self, policy, num_steps):
    observations = []
    observation = self.env.current_state()
    for _ in range(num_steps):
      action = policy(observation)
      observation = self.env.step(action)
      observations.append(observation)
    return observations

@ray.remote(num_gpus=2)
def update_policy(policy, *rollouts):
  # Update the policy.
  return policy

@ray.remote
def train_policy():
  # Create a policy.
  policy_id = create_policy.remote()
  # Create 10 actors.
  simulators = [Simulator.remote() for _ in range(10)]
  # Do 100 steps of training.
  for _ in range(100):
    # Perform one rollout on each actor.
    rollout_ids = [s.rollout.remote(policy_id)
                   for s in simulators]
    # Update the policy with the rollouts.
    policy_id =
        update_policy.remote(policy_id, *rollout_ids)
  return ray.get(policy_id)
```

Figure 5.3: Python code implementing the example in Figure 5.2 in Ray. Note that @ray.remote indicates remote functions and actors. Invocations of remote functions and actor methods return futures, which can be passed to subsequent remote functions or actor methods to encode task dependencies. Each actor has an environment object self.env shared between all of its methods.

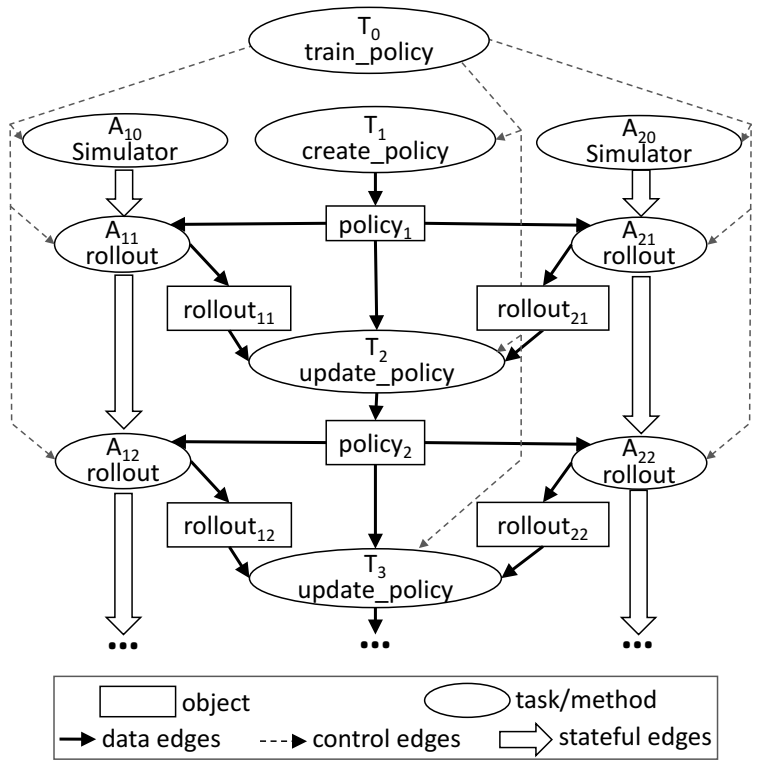


Figure 5.4: The task graph corresponding to an invocation of train_policy.remote() in Figure 5.3. Remote function calls and the actor method calls correspond to tasks in the task graph. The figure shows two actors. The method invocations for each actor (the tasks labeled A_{1i} and A_{2i}) have stateful edges between them indicating that they share the mutable actor state. There are control edges from train_policy to the tasks that it invokes. To train multiple policies in parallel, we could call train_policy.remote() multiple times.

Computation Model

Ray employs a dynamic task graph computation model [39], in which the execution of both remote functions and actor methods is automatically triggered by the system when their inputs become available. In this section, we describe how the computation graph (Figure 5.4) is constructed from a user program (Figure 5.3). This program uses the API in Table 5.1 to implement the pseudocode from Figure 5.2.

Ignoring actors first, there are two types of nodes in a computation graph: data objects and remote function invocations, or tasks. There are also two types of edges: data edges and control edges. Data edges capture the dependencies between data objects and tasks. More precisely, if data object D is an output of task T, we add a data edge from T to D. Similarly, if D is an input to T, we add a data edge from D to T. Control edges capture the computation dependencies that result from nested remote functions (Section 5.2): if task T_1 invokes task T_2, then we add a control edge from T_1 to T_2.

Actor method invocations are also represented as nodes in the computation graph. They are identical to tasks with one key difference. To capture the state dependency across subsequent method invocations on the same actor, we add a third type of edge: a stateful edge. If method M_j is called right after method M_i on the same actor, then we add a stateful edge from M_i to M_j. Thus, all methods invoked on the same actor object form a chain that is connected by stateful edges (Figure 5.4). This chain captures the order in which these methods were invoked.

Stateful edges help us embed actors in an otherwise stateless task graph, as they capture the implicit data dependency between successive method invocations sharing the internal state of an actor. Stateful edges also enable us to maintain lineage. As in other dataflow systems [140], we track data lineage to enable reconstruction. By explicitly including stateful edges in the lineage graph, we can easily reconstruct lost data, whether produced by remote functions or actor methods (Section 5.3).

5.3 Architecture

Ray's architecture comprises (1) an application layer implementing the API, and (2) a system layer providing high scalability and fault tolerance.

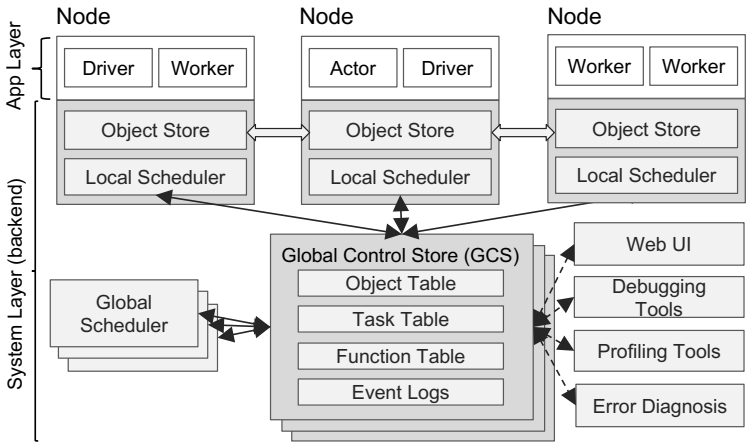


Figure 5.5: Ray's architecture consists of two parts: an *application* layer and a *system* layer. The application layer implements the API and the computation model described in Section 5.2, the system layer implements task scheduling and data management to satisfy the performance and fault-tolerance requirements.

Application Layer

The application layer consists of three types of processes:

- *Driver*: A process executing the user program.

- *Worker*: A stateless process that executes tasks (remote functions) invoked by a driver or another worker. Workers are started automatically and assigned tasks by the system layer. When a remote function is declared, the function is automatically published to all workers. A worker executes tasks serially, with no local state maintained across tasks.

- *Actor*: A stateful process that executes, when invoked, only the methods it exposes. Unlike a worker, an actor is explicitly instantiated by a worker or a driver. Like workers, actors execute methods serially, except that each method depends on the state resulting from the previous method execution.

System Layer

The system layer consists of three major components: a global control store, a distributed scheduler, and a distributed object store. All components are horizontally scalable and fault-tolerant.

Global Control Store (GCS)

The global control store (GCS) maintains the entire control state of the system, and it is a unique feature of our design. At its core, GCS is a key-value store with pub-sub functionality. We use sharding to achieve scale, and per-shard chain replication [104] to provide fault tolerance. The primary reason for the GCS and its design is to maintain fault tolerance and low latency for a system that can dynamically spawn millions of tasks per second.

Fault tolerance in case of node failure requires a solution to maintain lineage information. Existing lineage-based solutions [140, 135, 85, 56] focus on coarse-grained parallelism and can therefore use a single node (e.g., master, driver) to store the lineage without impacting performance. However, this design is not scalable for a fine-grained and dynamic workload like simulation. Therefore, we decouple the durable lineage storage from the other system components, allowing each to scale independently.

Maintaining low latency requires minimizing overheads in task scheduling, which involves choosing where to execute, and subsequently task dispatch, which involves retrieving remote inputs from other nodes. Many existing dataflow systems [140, 85, 106] couple these by storing object locations and sizes in a centralized scheduler, a natural design when the scheduler is not a bottleneck. However, the scale and granularity that Ray targets requires keeping the centralized scheduler off the critical path. Involving the scheduler in each object transfer is prohibitively expensive for primitives important to distributed training like allreduce, which is both communication-intensive and latency-sensitive. Therefore, we store the object metadata in the GCS rather than in the scheduler, fully decoupling task dispatch from task scheduling.

In summary, the GCS significantly simplifies Ray's overall design, as it *enables every component in the system to be stateless*. This not only simplifies support for fault tolerance

(i.e., on failure, components simply restart and read the lineage from the GCS), but also makes it easy to scale the distributed object store and scheduler independently, as all components share the needed state via the GCS. An added benefit is the easy development of debugging, profiling, and visualization tools.

Bottom-Up Distributed Scheduler

As discussed in Section 5.1, Ray needs to dynamically schedule millions of tasks per second, tasks which may take as little as a few milliseconds. None of the cluster schedulers we are aware of meet these requirements. Most cluster computing frameworks, such as Spark [140], CIEL [85], and Dryad [56] implement a centralized scheduler, which can provide locality but at latencies in the tens of ms. Distributed schedulers such as work stealing [17], Sparrow [98] and Canary [102] can achieve high scale, but they either don't consider data locality [17], or assume tasks belong to independent jobs [98], or assume the computation graph is known [102].

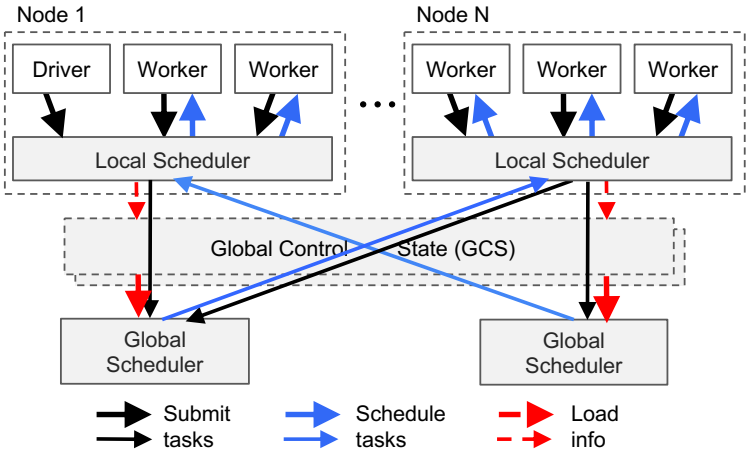


Figure 5.6: Bottom-up distributed scheduler. Tasks are submitted bottom-up, from drivers and workers to a local scheduler and forwarded to the global scheduler only if needed (Section 5.3). The thickness of each arrow is proportional to its request rate.

To satisfy the above requirements, we design a two-level hierarchical scheduler consisting of a global scheduler and per-node local schedulers. To avoid overloading the global scheduler, the tasks created at a node are submitted first to the node's local scheduler. A local scheduler schedules tasks locally unless the node is overloaded (i.e., its local task queue exceeds a predefined threshold), or it cannot satisfy a task's requirements (e.g., lacks a GPU). If a local scheduler decides not to schedule a task locally, it forwards it to the global scheduler. Since this scheduler attempts to schedule tasks locally first (i.e., at the leaves of the scheduling hierarchy), we call it a *bottom-up scheduler*.

The global scheduler considers each node's load and task's constraints to make scheduling decisions. More precisely, the global scheduler identifies the set of nodes that have enough

resources of the type requested by the task, and of these nodes selects the node which provides the lowest *estimated waiting time*. At a given node, this time is the sum of (i) the estimated time the task will be queued at that node (i.e., task queue size times average task execution), and (ii) the estimated transfer time of tasks remote inputs (i.e., total size of remote inputs divided by average bandwidth). The global scheduler gets the queue size at each node and the node resource availability via heartbeats, and the location of the task's inputs and their sizes from GCS. Furthermore, the global scheduler computes the average task execution and the average transfer bandwidth using simple exponential averaging. If the global scheduler becomes a bottleneck, we can instantiate more replicas all sharing the same information via GCS. This makes our scheduler architecture highly scalable.

In-Memory Distributed Object Store

To minimize task latency, we implement an in-memory distributed storage system to store the inputs and outputs of every task, or stateless computation. On each node, we implement the object store via *shared memory*. This allows zero-copy data sharing between tasks running on the same node. As a data format, we use Apache Arrow [8].

If a task's inputs are not local, the inputs are replicated to the local object store before execution. Also, a task writes its outputs to the local object store. Replication eliminates the potential bottleneck due to hot data objects and minimizes task execution time as a task only reads/writes data from/to the local memory. This increases throughput for computation-bound workloads, a profile shared by many AI applications. For low latency, we keep objects entirely in memory and evict them as needed to disk using an LRU policy.

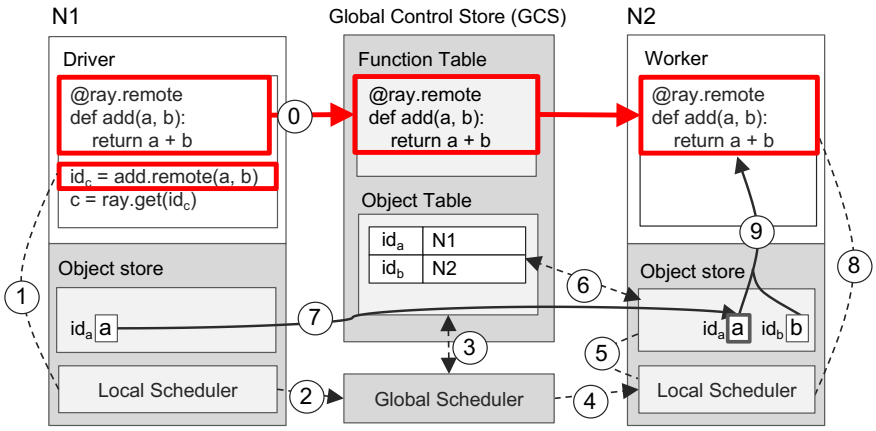
As with existing cluster computing frameworks, such as Spark [140], and Dryad [56], the object store is limited to *immutable data*. This obviates the need for complex consistency protocols (as objects are not updated), and simplifies support for fault tolerance. In the case of node failure, Ray recovers any needed objects through lineage re-execution. The lineage stored in the GCS tracks both stateless tasks and stateful actors during initial execution; we use the former to reconstruct objects in the store.

For simplicity, our object store does not support distributed objects, i.e., each object fits on a single node. Distributed objects like large matrices or trees can be implemented at the application level as collections of futures.

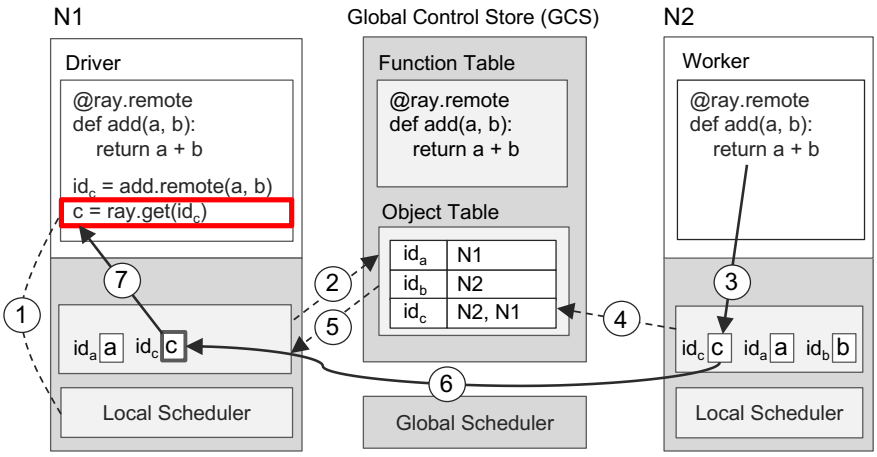
Implementation

Ray is an active open source project[3] developed at the University of California, Berkeley. Ray fully integrates with the Python environment and is easy to install by simply running `pip install ray`. The implementation comprises $\approx$ 40K lines of code (LoC), 72% in C++ for the system layer, 28% in Python for the application layer. The GCS uses one Redis [110] key-value store per shard, with entirely single-key operations. GCS tables are sharded by object and task IDs to scale, and every shard is chain-replicated [104] for fault tolerance. We

[3] `https://github.com/ray-project/ray`



(a) Executing a task remotely



(b) Returning the result of a remote task

Figure 5.7: An end-to-end example that adds a and b and returns c. Solid lines are data plane operations and dotted lines are control plane operations. (a) The function **add()** is registered with the GCS by node 1 ($N1$), invoked on $N1$, and executed on $N2$. (b) $N1$ gets **add()**'s result using **ray.get()**. The Object Table entry for c is created in step 4 and updated in step 6 after c is copied to $N1$.

implement both the local and global schedulers as event-driven, single-threaded processes. Internally, local schedulers maintain cached state for local object metadata, tasks waiting for inputs, and tasks ready for dispatch to a worker. To transfer large objects between different object stores, we stripe the object across multiple TCP connections.

Putting Everything Together

Figure 5.7 illustrates how Ray works end-to-end with a simple example that adds two objects a and b, which could be scalars or matrices, and returns result c. The remote function **add()**

is automatically registered with the GCS upon initialization and distributed to every worker in the system (step 0 in Figure 5.7a).

Figure 5.7a shows the step-by-step operations triggered by a driver invoking **add.remote(**a, b**)**, where a and b are stored on nodes $N1$ and $N2$, respectively. The driver submits **add(**a**,** b**)** to the local scheduler (step 1), which forwards it to a global scheduler (step 2).[4] Next, the global scheduler looks up the locations of **add(**a**,** b**)**'s arguments in the GCS (step 3) and decides to schedule the task on node $N2$, which stores argument b (step 4). The local scheduler at node $N2$ checks whether the local object store contains **add(**a**,** b**)**'s arguments (step 5). Since the local store doesn't have object a, it looks up a's location in the GCS (step 6). Learning that a is stored at $N1$, $N2$'s object store replicates it locally (step 7). As all arguments of **add()** are now stored locally, the local scheduler invokes **add()** at a local worker (step 8), which accesses the arguments via shared memory (step 9).

Figure 5.7b shows the step-by-step operations triggered by the execution of **ray.get()** at $N1$, and of **add()** at $N2$, respectively. Upon **ray.get(**id_c**)**'s invocation, the driver checks the local object store for the value c, using the future id_c returned by **add()** (step 1). Since the local object store doesn't store c, it looks up its location in the GCS. At this time, there is no entry for c, as c has not been created yet. As a result, $N1$'s object store registers a callback with the Object Table to be triggered when c's entry has been created (step 2). Meanwhile, at $N2$, **add()** completes its execution, stores the result c in the local object store (step 3), which in turn adds c's entry to the GCS (step 4). As a result, the GCS triggers a callback to $N1$'s object store with c's entry (step 5). Next, $N1$ replicates c from $N2$ (step 6), and returns c to **ray.get()** (step 7), which finally completes the task.

While this example involves a large number of RPCs, in many cases this number is much smaller, as most tasks are scheduled locally, and the GCS replies are cached by the global and local schedulers.

5.4 Evaluation

In our evaluation, we study the following questions:

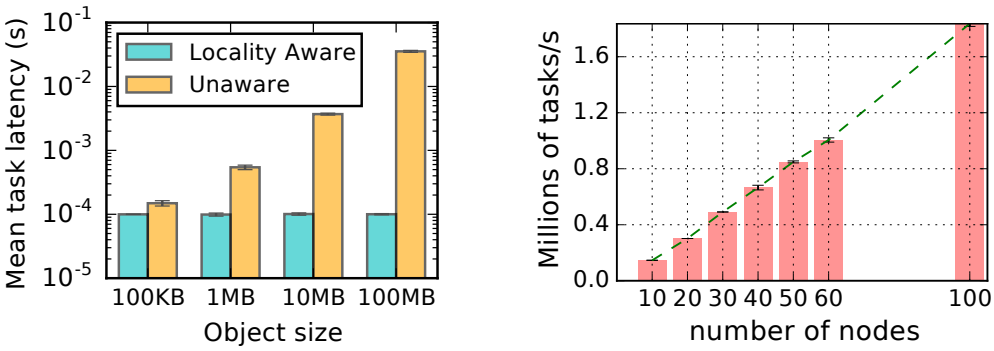
1. How well does Ray meet the latency, scalability, and fault tolerance requirements listed in Section 5.1? (Section 5.4)

2. What overheads are imposed on distributed primitives (e.g., allreduce) written using Ray's API? (Section 5.4)

3. In the context of RL workloads, how does Ray compare against specialized systems for training, serving, and simulation? (Section 5.4)

4. What advantages does Ray provide for RL applications, compared to custom systems? (Section 5.4)

[4] Note that $N1$ could also decide to schedule the task locally.



(a) Ray locality scheduling (b) Ray scalability

Figure 5.8: (a) Tasks leverage locality-aware placement. 1000 tasks with a random object dependency are scheduled onto one of two nodes. With locality-aware policy, task latency remains independent of the size of task inputs instead of growing by 1-2 orders of magnitude. (b) Near-linear scalability leveraging the GCS and bottom-up distributed scheduler. Ray reaches 1 million tasks per second throughput with 60 nodes. $x \in \{70, 80, 90\}$ omitted due to cost.

All experiments were run on Amazon Web Services. Unless otherwise stated, we use m4.16xlarge CPU instances and p3.16xlarge GPU instances.

Microbenchmarks

Locality-aware task placement. Fine-grain load balancing and locality-aware placement are primary benefits of tasks in Ray. Actors, once placed, are unable to move their computation to large remote objects, while tasks can. In Figure 5.8a, tasks placed without data locality awareness (as is the case for actor methods), suffer 1-2 orders of magnitude latency increase at 10-100MB input data sizes. Ray unifies tasks and actors through the shared object store, allowing developers to use tasks for e.g., expensive postprocessing on output produced by simulation actors.

End-to-end scalability. One of the key benefits of the Global Control Store (GCS) and the bottom-up distributed scheduler is the ability to horizontally scale the system to support a high throughput of fine-grained tasks, while maintaining fault tolerance and low-latency task scheduling. In Figure 5.8b, we evaluate this ability on an embarrassingly parallel workload of empty tasks, increasing the cluster size on the x-axis. We observe near-perfect linearity in progressively increasing task throughput. Ray exceeds 1 million tasks per second throughput at 60 nodes and continues to scale linearly beyond 1.8 million tasks per second at 100 nodes. The rightmost datapoint shows that Ray can process 100 million tasks in less than a minute (54s), with minimum variability. As expected, increasing task duration reduces throughput proportionally to mean task duration, but the overall scalability remains linear. While many realistic workloads may exhibit more limited scalability due to object dependencies and inherent limits to application parallelism, this demonstrates the scalability of our overall architecture under high load.

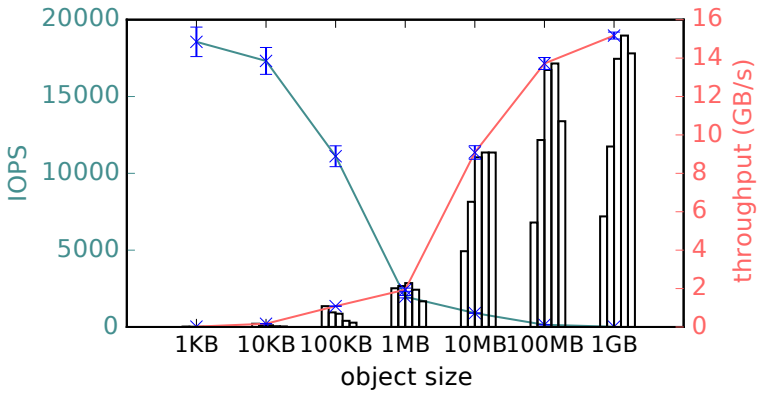


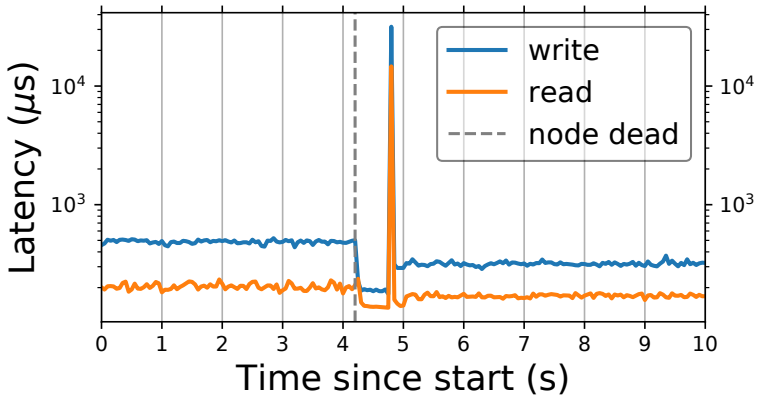
Figure 5.9: Object store write throughput and IOPS. From a single client, throughput exceeds 15GB/s (red) for large objects and 18K IOPS (cyan) for small objects on a 16 core instance (m4.4xlarge). It uses 8 threads to copy objects larger than 0.5MB and 1 thread for small objects. Bar plots report throughput with 1, 2, 4, 8, 16 threads. Results are averaged over 5 runs.

Object store performance. To evaluate the performance of the object store (Section 5.3), we track two metrics: IOPS (for small objects) and write throughput (for large objects). In Figure 5.9, the write throughput from a single client exceeds 15GB/s as object size increases. For larger objects, memcpy dominates object creation time. For smaller objects, the main overheads are in serialization and IPC between the client and object store.

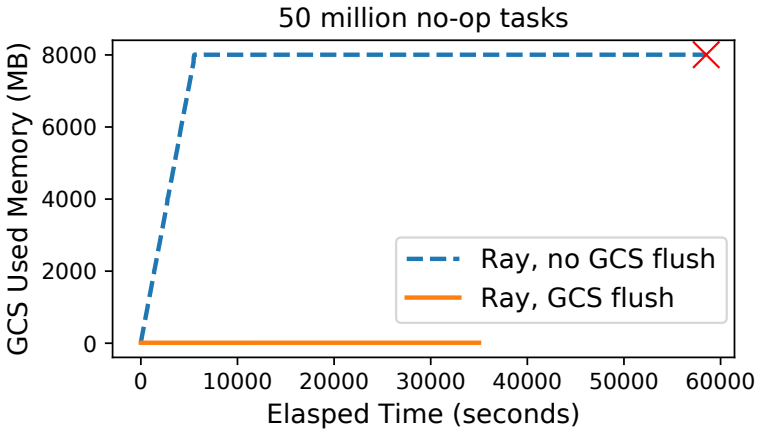
GCS fault tolerance. To maintain low latency while providing strong consistency and fault tolerance, we build a lightweight chain replication [104] layer on top of Redis. Figure 5.10a simulates recording Ray tasks to and reading tasks from the GCS, where keys are 25 bytes and values are 512 bytes. The client sends requests as fast as it can, having at most one in-flight request at a time. Failures are reported to the chain master either from the client (having received explicit errors, or timeouts despite retries) or from any server in the chain (having received explicit errors). Overall, reconfigurations caused a maximum *client-observed* delay of under 30ms (this includes both failure detection and recovery delays).

GCS flushing. Ray is equipped to periodically flush the contents of GCS to disk. In Figure 5.10b we submit 50 million empty tasks sequentially and monitor GCS memory consumption. As expected, it grows linearly with the number of tasks tracked and eventually reaches the memory capacity of the system. At that point, the system becomes stalled and the workload fails to finish within a reasonable amount of time. With periodic GCS flushing, we achieve two goals. First, the memory footprint is capped at a user-configurable level (in the microbenchmark we employ an aggressive strategy where consumed memory is kept as low as possible). Second, the flushing mechanism provides a natural way to snapshot lineage to disk for long-running Ray applications.

Recovering from task failures. In Figure 5.11a, we demonstrate Ray's ability to transparently recover from worker node failures and elastically scale, using the durable GCS lineage storage. The workload, run on m4.xlarge instances, consists of linear chains of 100ms tasks submitted by the driver. As nodes are removed (at 25s, 50s, 100s), the local schedulers



(a) A timeline for GCS read and write latencies as viewed from a client submitting tasks. The chain starts with 2 replicas. We manually trigger reconfiguration as follows. At $t \approx 4.2$s, a chain member is killed; immediately after, a new chain member joins, initiates state transfer, and restores the chain to 2-way replication. The maximum client-observed latency is under 30ms despite reconfigurations.

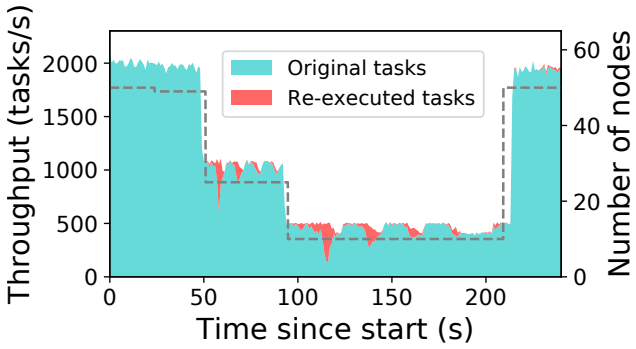


(b) The Ray GCS maintains a constant memory footprint with GCS flushing. Without GCS flushing, the memory footprint reaches a maximum capacity and the workload fails to complete within a predetermined duration (indicated by the red cross).

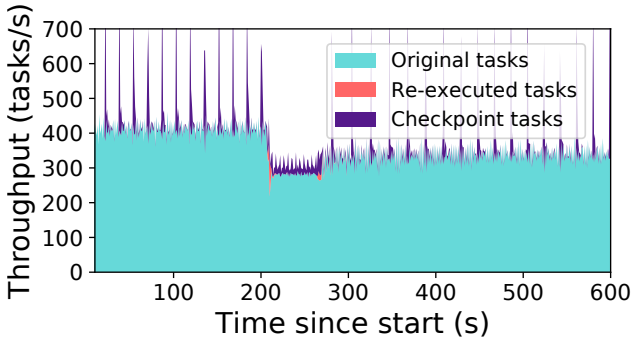
Figure 5.10: Ray GCS fault tolerance and flushing.

reconstruct previous results in the chain in order to continue execution. Overall *per-node* throughput remains stable throughout.

Recovering from actor failures. By encoding actor method calls as stateful edges directly in the dependency graph, we can reuse the same object reconstruction mechanism as in Figure 5.11a to provide transparent fault tolerance for *stateful computation*. Ray additionally leverages user-defined checkpoint functions to bound the reconstruction time for actors (Figure 5.11b). With minimal overhead, checkpointing enables only 500 methods to be re-executed, versus 10k re-executions without checkpointing. In the future, we hope to further reduce actor reconstruction time, e.g., by allowing users to annotate methods that



(a) Task reconstruction



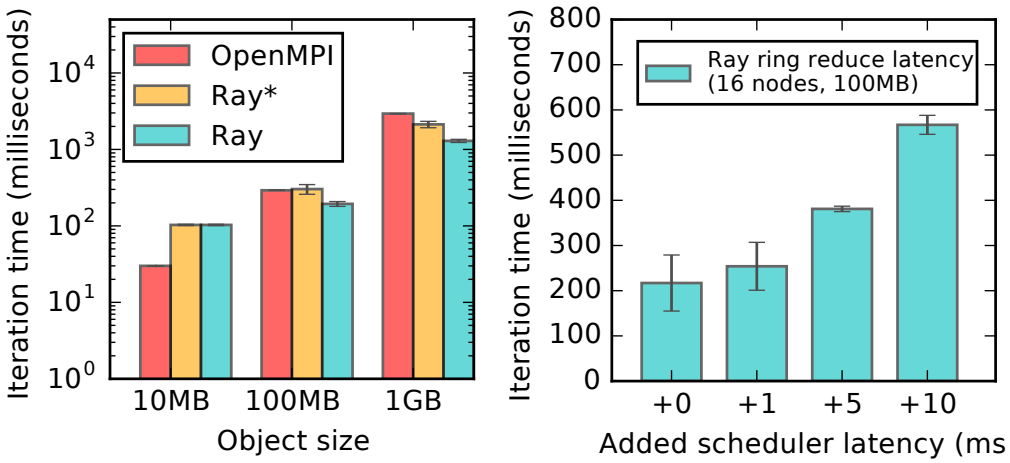
(b) Actor reconstruction

Figure 5.11: Ray fault-tolerance. **(a)** Ray reconstructs lost task dependencies as nodes are removed (dotted line), and recovers to original throughput when nodes are added back. Each task is 100ms and depends on an object generated by a previously submitted task. **(b)** Actors are reconstructed from their last checkpoint. At $t = 200$s, we kill 2 of the 10 nodes, causing 400 of the 2000 actors in the cluster to be recovered on the remaining nodes ($t = 200$–270s).

do not mutate state.

Allreduce. Allreduce is a distributed communication primitive important to many machine learning workloads. Here, we evaluate whether Ray can natively support a ring allreduce [126] implementation with low enough overhead to match existing implementations [116]. We find that Ray completes allreduce across 16 nodes on 100MB in ∼200ms and 1GB in ∼1200ms, surprisingly outperforming OpenMPI (v1.10), a popular MPI implementation, by 1.5× and 2× respectively (Figure 5.12a). We attribute Ray's performance to its use of multiple threads for network transfers, taking full advantage of the 25Gbps connection between nodes on AWS, whereas OpenMPI sequentially sends and receives data on a single thread [44]. For smaller objects, OpenMPI outperforms Ray by switching to a lower overhead algorithm, an optimization we plan to implement in the future.

Ray's scheduler performance is critical to implementing primitives such as allreduce. In Figure 5.12b, we inject artificial task execution delays and show that performance drops nearly 2× with just a few ms of extra latency. Systems with centralized schedulers like Spark and CIEL typically have scheduler overheads in the tens of milliseconds [131, 89], making such



(a) Ray vs OpenMPI (b) Ray scheduler ablation

Figure 5.12: (a) Mean execution time of allreduce on 16 m4.16xl nodes. Each worker runs on a distinct node. Ray* restricts Ray to 1 thread for sending and 1 thread for receiving. (b) Ray's low-latency scheduling is critical for allreduce.

workloads impractical. Scheduler *throughput* also becomes a bottleneck since the number of tasks required by ring reduce scales quadratically with the number of participants.

Building blocks

End-to-end applications (e.g., AlphaGo [119]) require a tight coupling of training, serving, and simulation. In this section, we isolate each of these workloads to a setting that illustrates a typical RL application's requirements. Due to a flexible programming model targeted to RL, and a system designed to support this programming model, Ray matches and sometimes exceeds the performance of dedicated systems for these individual workloads.

Distributed Training

We implement data-parallel synchronous SGD leveraging the Ray actor abstraction to represent model replicas. Model weights are synchronized via allreduce (5.4) or parameter server, both implemented on top of the Ray API.

In Figure 5.13, we evaluate the performance of the Ray (synchronous) parameter-server SGD implementation against state-of-the-art implementations [116], using the same TensorFlow model and synthetic data generator for each experiment. We compare only against TensorFlow-based systems to accurately measure the overhead imposed by Ray, rather than differences between the deep learning frameworks themselves. In each iteration, model replica actors compute gradients in parallel, send the gradients to a sharded parameter server, then read the summed gradients from the parameter server for the next iteration.

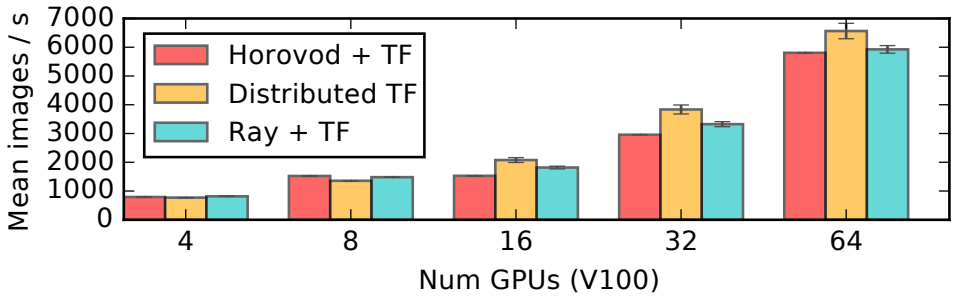


Figure 5.13: Images per second reached when distributing the training of a ResNet-101 TensorFlow model (from the official TF benchmark). All experiments were run on p3.16xl instances connected by 25Gbps Ethernet, and workers allocated 4 GPUs per node as done in Horovod [116]. We note some measurement deviations from previously reported, likely due to hardware differences and recent TensorFlow performance improvements. We used OpenMPI 3.0, TF 1.8, and NCCL2 for all runs.

Figure 5.13 shows that Ray matches the performance of Horovod and is within 10% of distributed TensorFlow (in `distributed_replicated` mode). This is due to the ability to express the same application-level optimizations found in these specialized systems in Ray's general-purpose API. A key optimization is the pipelining of gradient computation, transfer, and summation within a single iteration. To overlap GPU computation with network transfer, we use a custom TensorFlow operator to write tensors directly to Ray's object store.

Serving

Model serving is an important component of end-to-end applications. Ray focuses primarily on the *embedded* serving of models to simulators running within the same dynamic task graph (e.g., within an RL application on Ray). In contrast, systems like Clipper [30] focus on serving predictions to external clients.

In this setting, low latency is critical for achieving high utilization. To show this, in Table 5.3 we compare the server throughput achieved using a Ray actor to serve a policy versus using the open source Clipper system over REST. Here, both client and server processes are co-located on the same machine (a p3.8xlarge instance). This is often the case for RL applications but not for the general web serving workloads addressed by systems like Clipper. Due to its low-overhead serialization and shared memory abstractions, Ray achieves an order of magnitude higher throughput for a small fully connected policy model that takes in a large input and is also faster on a more expensive residual network policy model, similar to one used in AlphaGo Zero, that takes smaller input.

Simulation

Simulators used in RL produce results with variable lengths ("timesteps") that, due to the tight loop with training, must be used as soon as they are available. The task heterogene-

System	Small Input	Larger Input
Clipper	4400 ± 15 states/sec	290 ± 1.3 states/sec
Ray	6200 ± 21 states/sec	6900 ± 150 states/sec

Table 5.3: Throughput comparisons for Clipper [30], a dedicated serving system, and Ray for two embedded serving workloads. We use a residual network and a small fully connected network, taking 10ms and 5ms to evaluate, respectively. The server is queried by clients that each send states of size 4KB and 100KB respectively in batches of 64.

ity and timeliness requirements make simulations hard to support efficiently in BSP-style systems. To demonstrate, we compare (1) an MPI implementation that submits $3n$ parallel simulation runs on n cores in 3 rounds, with a global barrier between rounds[5], to (2) a Ray program that issues the same $3n$ tasks while concurrently gathering simulation results back to the driver. Table 5.4 shows that both systems scale well, yet Ray achieves up to 1.8× throughput. This motivates a programming model that can dynamically spawn and collect the results of fine-grained simulation tasks.

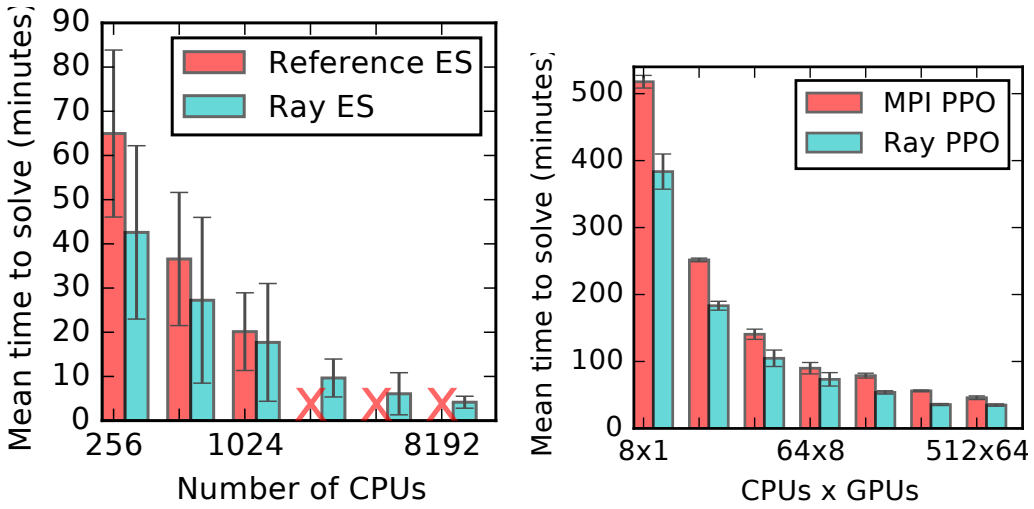
System, programming model	1 CPU	16 CPUs	256 CPUs
MPI, bulk synchronous	22.6K	208K	2.16M
Ray, asynchronous tasks	22.3K	290K	4.03M

Table 5.4: Timesteps per second for the Pendulum-v0 simulator in OpenAI Gym [21]. Ray allows for better utilization when running heterogeneous simulations at scale.

RL Applications

Without a system that can tightly couple the training, simulation, and serving steps, reinforcement learning algorithms today are implemented as one-off solutions that make it difficult to incorporate optimizations that, for example, require a different computation structure or that utilize different architectures. Consequently, with implementations of two representative reinforcement learning applications in Ray, we are able to match and even outperform custom systems built specifically for these algorithms. The primary reason is the flexibility of Ray's programming model, which can express application-level optimizations that would require substantial engineering effort to port to custom-built systems, but are transparently supported by Ray's dynamic task graph execution engine.

[5]Note that experts *can* use MPI's asynchronous primitives to get around barriers—at the expense of increased program complexity —we nonetheless chose such an implementation to simulate BSP.



(a) Evolution Strategies (b) PPO

Figure 5.14: Time to reach a score of 6000 in the Humanoid-v1 task [21]. **(a)** The Ray ES implementation scales well to 8192 cores and achieves a median time of 3.7 minutes, over twice as fast as the best published result. The special-purpose system failed to run beyond 1024 cores. ES is faster than PPO on this benchmark, but shows greater runtime variance. **(b)** The Ray PPO implementation outperforms a specialized MPI implementation [97] with fewer GPUs, at a fraction of the cost. The MPI implementation required 1 GPU for every 8 CPUs, whereas the Ray version required at most 8 GPUs (and never more than 1 GPU per 8 CPUs).

Evolution Strategies

To evaluate Ray on large-scale RL workloads, we implement the evolution strategies (ES) algorithm and compare to the reference implementation [109]—a system specially built for this algorithm that relies on Redis for messaging and low-level multiprocessing libraries for data-sharing. The algorithm periodically broadcasts a new policy to a pool of workers and aggregates the results of roughly 10000 tasks (each performing 10 to 1000 simulation steps).

As shown in Figure 5.14a, an implementation on Ray scales to 8192 cores. Doubling the cores available yields an average completion time speedup of 1.6×. Conversely, the special-purpose system fails to complete at 2048 cores, where the work in the system exceeds the processing capacity of the application driver. To avoid this issue, the Ray implementation uses an aggregation tree of actors, reaching a median time of 3.7 minutes, more than twice as fast as the best published result (10 minutes).

Initial parallelization of a serial implementation using Ray required modifying only 7 lines of code. Performance improvement through hierarchical aggregation was easy to realize with Ray's support for nested tasks and actors. In contrast, the reference implementation had several hundred lines of code dedicated to a protocol for communicating tasks and data between workers, and would require further engineering to support optimizations like hierarchical aggregation.

Proximal Policy Optimization

We implement Proximal Policy Optimization (PPO) [113] in Ray and compare to a highly-optimized reference implementation [97] that uses OpenMPI communication primitives. The algorithm is an asynchronous scatter-gather, where new tasks are assigned to simulation actors as they return rollouts to the driver. Tasks are submitted until 320000 simulation steps are collected (each task produces between 10 and 1000 steps). The policy update performs 20 steps of SGD with a batch size of 32768. The model parameters in this example are roughly 350KB. These experiments were run using p2.16xlarge (GPU) and m4.16xlarge (high CPU) instances.

As shown in Figure 5.14b, the Ray implementation outperforms the optimized MPI implementation in all experiments, while using a fraction of the GPUs. The reason is that Ray is heterogeneity-aware and allows the user to utilize asymmetric architectures by expressing resource requirements at the granularity of a task or actor. The Ray implementation can then leverage TensorFlow's single-process multi-GPU support and can pin objects in GPU memory when possible. This optimization cannot be easily ported to MPI due to the need to asynchronously gather rollouts to a single GPU process. Indeed, [97] includes two custom implementations of PPO, one using MPI for large clusters and one that is optimized for GPUs but that is restricted to a single node. Ray allows for an implementation suitable for both scenarios.

Ray's ability to handle resource heterogeneity also decreased PPO's cost by a factor of 4.5 [42], since CPU-only tasks can be scheduled on cheaper high-CPU instances. In contrast, MPI applications often exhibit symmetric architectures, in which all processes run the same code and require identical resources, in this case preventing the use of CPU-only machines for scale-out. Furthermore, the MPI implementation requires on-demand instances since it does not transparently handle failure. Assuming 4× cheaper spot instances, *Ray's fault tolerance and resource-aware scheduling together cut costs by 18×*.

5.5 Related Work

Dynamic task graphs

Ray is closely related to CIEL [85] and Dask [106]. All three support dynamic task graphs with nested tasks and implement the futures abstraction. CIEL also provides lineage-based fault tolerance, while Dask, like Ray, fully integrates with Python. However, Ray differs in two aspects that have important performance consequences. First, Ray extends the task model with an actor abstraction. This is necessary for efficient stateful computation in distributed training and serving, to keep the model data collocated with the computation. Second, Ray employs a fully distributed and decoupled control plane and scheduler, instead of relying on a single master storing all metadata. This is critical for efficiently supporting primitives like allreduce without system modification. At peak performance for 100MB on 16 nodes, allreduce on Ray (Section 5.4) submits 32 rounds of 16 tasks in 200ms. Meanwhile,

Dask reports a maximum scheduler throughput of 3k tasks/s on 512 cores [32]. With a centralized scheduler, each round of allreduce would then incur a minimum of ∼5ms of scheduling delay, translating to up to 2× worse completion time (Figure 5.12b). Even with a decentralized scheduler, coupling the control plane information with the scheduler leaves the latter on the critical path for data transfer, adding an extra roundtrip to every round of allreduce.

Dataflow systems

Popular dataflow systems, such as MapReduce [33], Spark [137], and Dryad [56] have widespread adoption for analytics and ML workloads, but their computation model is too restrictive for a fine-grained and dynamic simulation workload. Spark and MapReduce implement the BSP execution model, which assumes that tasks within the same stage perform the same computation and take roughly the same amount of time. Dryad relaxes this restriction but lacks support for dynamic task graphs. Furthermore, none of these systems provide an actor abstraction, nor implement a distributed scalable control plane and scheduler. Finally, Naiad [87] is a dataflow system that provides improved scalability for some workloads, but only supports static task graphs.

Machine learning frameworks

TensorFlow [1] and MXNet [27] target deep learning workloads and efficiently leverage both CPUs and GPUs. While they achieve great performance for training workloads consisting of static DAGs of linear algebra operations, they have limited support for the more general computation required to tightly couple training with simulation and embedded serving. TensorFlow Fold [71] provides some support for dynamic task graphs, as well as MXNet through its internal C++ APIs, but neither fully supports the ability to modify the DAG during execution in response to task progress, task completion times, or faults. TensorFlow and MXNet in principle achieve generality by allowing the programmer to simulate low-level message-passing and synchronization primitives, but the pitfalls and user experience in this case are similar to those of MPI. OpenMPI [44] can achieve high performance, but it is relatively hard to program as it requires explicit coordination to handle heterogeneous and dynamic task graphs. Furthermore, it forces the programmer to explicitly handle fault tolerance.

Actor systems

Orleans [22] and Akka [4] are two actor frameworks well suited to developing highly available and concurrent distributed systems. However, compared to Ray, they provide less support for recovery from data loss. To recover *stateful actors*, the Orleans developer must explicitly checkpoint actor state and intermediate responses. *Stateless actors* in Orleans can be replicated for scale-out, and could therefore act as tasks, but unlike in Ray, they have no

lineage. Similarly, while Akka explicitly supports persisting actor state across failures, it does not provide efficient fault tolerance for *stateless computation* (i.e., tasks). For message delivery, Orleans provides at-least-once and Akka provides at-most-once semantics. In contrast, Ray provides transparent fault tolerance and exactly-once semantics, as each method call is logged in the GCS and both arguments and results are immutable. We find that in practice these limitations do not affect the performance of our applications. Erlang [11] and C++ Actor Framework [26], two other actor-based systems, have similarly limited support for fault tolerance.

Global control store and scheduling

The concept of logically centralizing the control plane has been previously proposed in software defined networks (SDNs) [25], distributed file systems (e.g., GFS [45]), resource management (e.g., Omega [115]), and distributed frameworks (e.g., MapReduce [33], BOOM [5]), to name a few. Ray draws inspiration from these pioneering efforts, but provides significant improvements. In contrast with SDNs, BOOM, and GFS, Ray decouples the storage of the control plane information (e.g., GCS) from the logic implementation (e.g., schedulers). This allows both storage and computation layers to scale independently, which is key to achieving our scalability targets. Omega uses a distributed architecture in which schedulers coordinate via globally shared state. To this architecture, Ray adds global schedulers to balance load across local schedulers, and targets ms-level, not second-level, task scheduling.

Ray implements a unique distributed bottom-up scheduler that is horizontally scalable, and can handle dynamically constructed task graphs. Unlike Ray, most existing cluster computing systems [33, 140, 85] use a centralized scheduler architecture. While Sparrow [98] is decentralized, its schedulers make independent decisions, limiting the possible scheduling policies, and all tasks of a job are handled by the same global scheduler. Mesos [51] implements a two-level hierarchical scheduler, but its top-level scheduler manages frameworks, not individual tasks. Canary [102] achieves impressive performance by having each scheduler instance handle a portion of the task graph, but does not handle dynamic computation graphs.

Cilk [17] is a parallel programming language whose work-stealing scheduler achieves provably efficient load-balancing for dynamic task graphs. However, with no central coordinator like Ray's global scheduler, this fully parallel design is also difficult to extend to support data locality and resource heterogeneity in a distributed setting.

5.6 Discussion and Experiences

Building Ray has been a long journey. It started two years ago with a Spark library to perform distributed training and simulations. However, the relative inflexibility of the BSP model, the high per-task overhead, and the lack of an actor abstraction led us to develop a new system. Since we released Ray roughly one year ago, several hundreds of people have

used it and several companies are running it in production. Here we discuss our experience developing and using Ray, and some early user feedback.

The Ray API

In designing the API, we have emphasized minimalism. Initially we started with a basic *task* abstraction. Later, we added the **wait()** primitive to accommodate rollouts with heterogeneous durations and the *actor* abstraction to accommodate third-party simulators and amortize the overhead of expensive initializations. While the resulting API is relatively low-level, it has proven both powerful and simple to use. We have already used this API to implement many state-of-the-art RL algorithms on top of Ray, including A3C [77], PPO [113], DQN [78], ES [109], DDPG [118], and Ape-X [54]. In most cases it took us just a few tens of lines of code to port these algorithms to Ray. Based on early user feedback, we are considering enhancing the API to include higher level primitives and libraries, which could also inform scheduling decisions.

Limitations

Given the workload generality, specialized optimizations are hard. For example, we must make scheduling decisions without full knowledge of the computation graph. Scheduling optimizations in Ray might require more complex runtime profiling. In addition, storing lineage for each task requires the implementation of garbage collection policies to bound storage costs in the GCS, a feature we are actively developing.

Fault tolerance

We are often asked if fault tolerance is really needed for AI applications. After all, due to the statistical nature of many AI algorithms, one could simply ignore failed rollouts. Based on our experience, our answer is "yes". First, the ability to ignore failures makes applications much easier to write and reason about. Second, our particular implementation of fault tolerance via deterministic replay dramatically simplifies debugging as it allows us to easily reproduce most errors. This is particularly important since, due to their stochasticity, AI algorithms are notoriously hard to debug. Third, fault tolerance helps save money since it allows us to run on cheap resources like spot instances on AWS. Of course, this comes at the price of some overhead. However, we found this overhead to be minimal for our target workloads.

GCS and Horizontal Scalability

The GCS dramatically simplified Ray development and debugging. It enabled us to query the entire system state while debugging Ray itself, instead of having to manually expose internal

component state. In addition, the GCS is also the backend for our timeline visualization tool, used for application-level debugging.

The GCS was also instrumental to Ray's horizontal scalability. In Section 6.6, we were able to scale by adding more shards whenever the GCS became a bottleneck. The GCS also enabled the global scheduler to scale by simply adding more replicas. Due to these advantages, we believe that centralizing control state will be a key design component of future distributed systems.

5.7 Conclusion

No general-purpose system today can efficiently support the tight loop of training, serving, and simulation. To express these core building blocks and meet the demands of emerging AI applications, Ray unifies task-parallel and actor programming models in a single dynamic task graph and employs a scalable architecture enabled by the global control store and a bottom-up distributed scheduler. The programming flexibility, high throughput, and low latencies simultaneously achieved by this architecture is particularly important for emerging artificial intelligence workloads, which produce tasks diverse in their resource requirements, duration, and functionality. Our evaluation demonstrates linear scalability up to 1.8 million tasks per second, transparent fault tolerance, and substantial performance improvements on several contemporary RL workloads. Thus, Ray provides a powerful combination of flexibility, performance, and ease of use for the development of future AI applications.

Chapter 6

Use Case: Large Scale Optimization

In this chapter[1], we will study an optimization algorithm that addresses the challenge of large scale optimization for machine learning. The algorithm makes it possible to improve the convergence of large scale optimization by constructing a second order approximation of the objective. We show a linear convergence rate of this algorithm in the stochastic setting that is common in machine learning. The algorithm is well suited for implementation with the distributed execution engine described in chapter 5, which can readily be used to parallelize gradient computations over minibatches and also (potentially asynchronously) perform the computations needed for the variance reduction step.

6.1 Introduction

A trend in machine learning has been toward using more parameters to model larger datasets. As a consequence, it is important to design optimization algorithms for these large-scale problems. A typical optimization problem arising in this setting is empirical risk minimization. That is,

$$\min_w \frac{1}{N} \sum_{i=1}^{N} f_i(w), \tag{6.1}$$

where $w \in \mathbb{R}^d$ may specify the parameters of a machine learning model, and $f_i(w)$ quantifies how well the model w fits the ith data point. Two challenges arise when attempting to solve Equation 6.1. First, d may be extremely large. Second, N may be extremely large.

When d is small, Newton's method is often the algorithm of choice due to its rapid convergence (both in theory and in practice). However, Newton's method requires the computation and inversion of the Hessian matrix $\nabla^2 f(w)$, which may be computationally too expensive in high dimensions. As a consequence, practitioners are often limited to using first-order methods which only compute gradients of the objective, requiring $O(d)$ computation per iteration. The gradient method is the simplest example of a first-order method, but much

[1]This material was previously published in [81].

work has been done to design quasi-Newton methods which incorporate information about the curvature of the objective without ever computing second derivatives. L-BFGS [70], the limited-memory version of the classic BFGS algorithm, is one of the most successful algorithms in this space. Inexact Newton methods are another approach to using second order information for large-scale optimization. They approximately invert the Hessian in $O(d)$ steps. This can be done by using a constant number of iterations of the conjugate gradient method [37, 38, 94].

When N is large, batch algorithms such as the gradient method, which compute the gradient of the full objective at every iteration, are slowed down by the fact that they have to process every data point before updating the model. Stochastic optimization algorithms get around this problem by updating the model w after processing only a small subset of the data, allowing them to make much progress in the time that it takes the gradient method to make a single step.

For many machine learning problems, where both d and N are large, stochastic gradient descent (SGD) and its variants are the most widely used algorithms [105, 19, 20], often because they are some of the few algorithms that can realistically be applied in this setting.

Given this context, much research in optimization has been directed toward designing better stochastic first-order algorithms. For a partial list, see [61, 122, 41, 117, 59, 107, 132, 91, 43, 3]. In particular, much progress has gone toward designing stochastic variants of L-BFGS [79, 133, 23, 18, 111, 120].

Unlike gradient descent, L-BFGS does not immediately lend itself to a stochastic version. The updates in the stochastic gradient method average together to produce a downhill direction in expectation. However, as pointed out in [23], the updates used in L-BFGS to construct the inverse Hessian approximation overwrite one another instead of averaging. Our algorithm addresses this problem in the same ways as [23], by computing Hessian vector products formed from larger minibatches.

Though stochastic methods often make rapid progress early on, the variance of the estimates of the gradient slow their convergence near the optimum. To illustrate this phenomenon, even if SGD is initialized at the optimum, it will immediately move to a point with a worse objective value. For this reason, convergence guarantees typically require diminishing step sizes. One promising line of work involves speeding up the convergence of stochastic first-order methods by reducing the variance of the gradient estimates [59, 107, 36, 117].

We introduce a stochastic variant of L-BFGS that incorporates the idea of variance reduction and has two desirable features. First, it obtains a guaranteed linear rate of convergence in the strongly-convex case. In particular, it does not require a diminishing step size in order to guarantee convergence (as partially evidenced by the fact that if our algorithm is initialized at the optimum it will stay there). Second, it performs very well on large-scale optimization problems, exhibiting a qualitatively linear rate of convergence in practice.

6.2 The Algorithm

We consider the problem of minimizing the function

$$f(w) = \frac{1}{N} \sum_{i=1}^{N} f_i(w) \tag{6.2}$$

over $w \in \mathbb{R}^d$. For a subset $\mathcal{S} \subseteq \{1, \ldots, N\}$, we define the subsampled function $f_{\mathcal{S}}$ by

$$f_{\mathcal{S}}(w) = \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} f_i(w). \tag{6.3}$$

Our updates will use stochastic estimates of the gradient $\nabla f_{\mathcal{S}}$ as well as stochastic approximations to the inverse Hessian $\nabla^2 f_{\mathcal{T}}$. Following [23], we use distinct subsets $\mathcal{S}, \mathcal{T} \subseteq \{1, \ldots, N\}$ in order to decouple the estimation of the gradient from the estimation of the Hessian. We let $b = |\mathcal{S}|$ and $b_H = |\mathcal{T}|$.

Following [59], we occasionally compute full gradients, which we use to reduce the variance of our stochastic gradient estimates.

The update rule for our algorithm will take the form

$$w_{k+1} = w_k - \eta_k H_k v_k.$$

In the gradient method, H_k is the identity matrix. In Newton's method, it is the inverse Hessian $(\nabla^2 f(w_k))^{-1}$. In our algorithm, as in L-BFGS, H_k will be an approximation to the inverse Hessian. Instead of the usual stochastic estimate of the gradient, v_k will be a stochastic estimate of the gradient with reduced variance.

Code for our algorithm is given in Algorithm 1. Our algorithm is specified by several parameters. It requires a step size η, a memory size M, and positive integers m and L. Every m iterations, the algorithm performs a full gradient computation, which it uses to reduce the variance of the stochastic gradient estimates. Every L iterations, the algorithm updates the inverse Hessian approximation. The vector s_r records the average direction in which the algorithm has made progress over the past $2L$ iterations. The vector y_r is obtained by multiplying s_r by a stochastic estimate of the Hessian. Note that this differs from the usual L-BFGS algorithm, which produces y_r by taking the difference between successive gradients. We find that this approach works better in the stochastic setting. The inverse Hessian approximation H_r is defined from the pairs (s_j, y_j) for $r - M + 1 \leq j \leq r$ using the standard L-BFGS update rule, which is described in Section 6.2. The user must also choose batch sizes b and b_H from which to construct the stochastic gradient and stochastic Hessian estimates.

In Algorithm 1 and below, we use I to refer to the identity matrix. We use $\mathcal{F}_{k,t}$ to denote the sigma algebra generated by the random variables introduced up to the time when the iteration counters k and t have the specified values. That is,

$$\mathcal{F}_{k,t} = \sigma \left(\begin{array}{c} \{\mathcal{S}_{k',t'} \,:\, k' < k \text{ or } k' = k \text{ and } t' < t\} \\ \cup \{\mathcal{T}_r : rL \leq mk + t\} \end{array} \right).$$

Algorithm 1 Stochastic L-BFGS

Input: initial state w_0, parameters m, M, and L, batch sizes b and b_H, and step size η

Initialize $r = 0$
Initialize $H_0 = I$
for $k = 0, \ldots$ **do**
    Compute a full gradient $\mu_k = \nabla f(w_k)$
    Set $x_0 = w_k$
    for $t = 0, \ldots, m-1$ **do**
        Sample a minibatch $\mathcal{S}_{k,t} \subseteq \{1, \ldots, N\}$
        Compute a stochastic gradient $\nabla f_{\mathcal{S}_{k,t}}(x_t)$
        Compute a variance reduced gradient $v_t = \nabla f_{\mathcal{S}_{k,t}}(x_t) - \nabla f_{\mathcal{S}_{k,t}}(w_k) + \mu_k$
        Set $x_{t+1} = x_t - \eta H_r v_t$
        if $t \equiv 0 \bmod L$ **then**
            Increment $r \leftarrow r + 1$
            Set $u_r = \frac{1}{L} \sum_{j=t-L}^{t-1} x_j$
            Sample $\mathcal{T}_r \subseteq \{1, \ldots, N\}$ to define the stochastic approximation $\nabla^2 f_{\mathcal{T}_r}(u_r)$
            Compute $s_r = u_r - u_{r-1}$
            Compute $y_r = \nabla^2 f_{\mathcal{T}_r}(u_r) s_r$
            Define H_r as in Section 6.2
    Set $w_{k+1} = x_i$ for randomly chosen $i \in \{0, \ldots, m-1\}$

We will use $\mathbb{E}_{k,t}$ to denote the conditional expectation with respect to $\mathcal{F}_{k,t}$.

We define the inverse Hessian approximation H_r in Section 6.2. Note that we do not actually construct the matrix H_r because doing so would require $O(d^2)$ computation. In practice, we directly compute products of the form $H_r v$ using the two-loop recursion [[]Algorithm 7.4]nocedal2006numerical.

Construction of the Inverse Hessian Approximation H_r

To define the inverse Hessian approximation H_r from the pairs (s_j, y_j), we follow the usual L-BFGS method. Let $\rho_j = 1/s_j^\top y_j$ and recursively define

$$H_r^{(j)} = (I - \rho_j s_j y_j^\top)^\top H_r^{(j-1)} (I - \rho_j s_j y_j^\top) + \rho_j s_j s_j^\top, \tag{6.4}$$

for $r - M + 1 \leq j \leq r$. Initialize $H_r^{(r-M)} = (s_r^\top y_r / \|y_r\|^2) I$ and set $H_r = H_r^{(r)}$.

Note that the update in Equation 6.4 preserves positive definiteness (note that $\rho_j > 0$), which implies that H_r and each $H_r^{(j)}$ will be positive definite, as will their inverses.

6.3 Preliminaries

Our analysis makes use of the following assumptions.

Assumption 1. *The function $f_i \colon \mathbb{R}^n \to \mathbb{R}$ is convex and twice continuously differentiable for each $1 \le i \le N$.*

Assumption 2. *There exist positive constants λ and Λ such that*

$$\lambda I \preceq \nabla^2 f_{\mathcal{T}}(w) \preceq \Lambda I \tag{6.5}$$

for all $w \in \mathbb{R}^d$ and all nonempty subsets $\mathcal{T} \subseteq \{1, \dots, N\}$. Note the lower bound trivially holds in the regularized case.

We will typically force strong convexity to hold by adding a strongly-convex regularizer to our objective (which can be absorbed into the f_i's). These assumptions imply that f has a unique minimizer, which we denote by w_*.

Lemma 3. *Suppose that Assumption 1 and Assumption 2 hold. Let $B_r = H_r^{-1}$. Then*

$$\begin{aligned} \operatorname{tr}(B_r) &\le (d+M)\Lambda \\ \det(B_r) &\ge \frac{\lambda^{d+M}}{((d+M)\Lambda)^M}. \end{aligned}$$

We prove Lemma 3 in Section 6.7.

Lemma 4. *Suppose that Assumption 1 and Assumption 2 hold. Then there exist constants $0 < \gamma \le \Gamma$ such that H_r satisfies*

$$\gamma I \preceq H_r \preceq \Gamma I \tag{6.6}$$

for all $r \ge 1$.

In Section 6.7, we prove Lemma 4 with the values

$$\gamma = \frac{1}{(d+M)\Lambda} \quad \text{and} \quad \Gamma = \frac{((d+M)\Lambda)^{d+M-1}}{\lambda^{d+M}}.$$

We will make use of Lemma 5, a simple result for strongly convex functions. We include a proof for completeness.

Lemma 5. *Suppose that f is continuously differentiable and strongly convex with parameter λ. Let w_* be the unique minimizer of f. Then for any $x \in \mathbb{R}^d$, we have*

$$\|\nabla f(x)\|^2 \ge 2\lambda(f(x) - f(w_*)).$$

Proof. By the strong convexity of f,

$$\begin{aligned} f(w_*) &\ge f(x) + \nabla f(x)^\top (w_* - x) + \frac{\lambda}{2}\|w_* - x\|^2 \\ &\ge f(x) + \min_v \left(\nabla f(x)^\top v + \frac{\lambda}{2}\|v\|^2 \right) \\ &= f(x) - \frac{1}{2\lambda}\|\nabla f(x)\|^2. \end{aligned}$$

The last equality holds by plugging in the minimizer $v = -\nabla f(x)/\lambda$. □

In Lemma 6, we bound the variance of our variance-reduced gradient estimates. The proof of Lemma 6, given in Section 6.7, closely follows that of [[]Theorem 1]johnson2013accelerating.

Lemma 6. *Let w_* be the unique minimizer of f. Let $\mu_k = \nabla f(w_k)$ and let $v_t = \nabla f_{\mathcal{S}}(x_t) - \nabla f_{\mathcal{S}}(w_k) + \mu_k$ be the variance-reduced stochastic gradient. Conditioning on $\mathcal{F}_{k,t}$ and taking an expectation with respect to $\mathcal{S}$, we have*

$$\mathbb{E}_{k,t}[\|v_t\|^2] \leq 4\Lambda(f(x_t) - f(w_*) + f(w_k) - f(w_*)). \tag{6.7}$$

6.4 Convergence Analysis

Theorem 7 states our main result.

Theorem 7. *Suppose that Assumption 1 and Assumption 2 hold. Let w_* be the unique minimizer of f. Then for all $k \geq 0$, we have*

$$\mathbb{E}[f(w_k) - f(w_*)] \leq \alpha^k \mathbb{E}[f(w_0) - f(w_*)],$$

where the convergence rate α is given by

$$\alpha = \frac{1/(2m\eta) + \eta\Gamma^2\Lambda^2}{\gamma\lambda - \eta\Gamma^2\Lambda^2} < 1,$$

assuming that we choose $\eta < \gamma\lambda/(2\Gamma^2\Lambda^2)$ and that we choose m large enough to satisfy

$$\gamma\lambda > \frac{1}{2m\eta} + 2\eta\Gamma^2\Lambda^2. \tag{6.8}$$

Proof. Using the Lipschitz continuity of ∇f, which follows from Assumption 2, we have

$$\begin{aligned} & f(x_{t+1}) \\ & \leq f(x_t) + \nabla f(x_t)^\top (x_{t+1} - x_t) + \frac{\Lambda}{2}\|x_{t+1} - x_t\|^2 \\ & = f(x_t) - \eta \nabla f(x_t)^\top H_r v_t + \frac{\eta^2\Lambda}{2}\|H_k v_t\|^2. \end{aligned} \tag{6.9}$$

Conditioning on $\mathcal{F}_{k,t}$ and taking expectations in Equation 6.9, this becomes

$$\begin{aligned} & \mathbb{E}_{k,t}[f(x_{t+1})] \\ & \leq f(x_t) - \eta \nabla f(x_t)^\top H_r \nabla f(x_t) + \frac{\eta^2\Lambda}{2}\mathbb{E}_{k,t}\|H_k v_t\|^2, \end{aligned} \tag{6.10}$$

where we used the fact that $\mathbb{E}_{k,t}[v_t] = \nabla f(x_t)$. We then use Lemma 4 to bound the second and third terms on the bottom line of Equation 6.10 to get

$$\mathbb{E}_{k,t}[f(x_{t+1})] \leq f(x_t) - \eta\gamma\|\nabla f(x_t)\|^2 + \frac{\eta^2\Gamma^2\Lambda}{2}\mathbb{E}_{k,t}\|v_t\|^2.$$

Now, we bound $\mathbb{E}_{k,t}\|v_t\|^2$ using Lemma 6 and we bound $\|\nabla f(x_t)\|^2$ using Lemma 5. Doing so gives

$$\begin{aligned}
&\mathbb{E}_{k,t}[f(x_{t+1})] \\
&\leq f(x_t) - 2\eta\gamma\lambda(f(x_t) - f(w_*)) \\
&\quad + 2\eta^2\Gamma^2\Lambda^2(f(x_t) - f(w_*) + f(w_k) - f(w_*)) \\
&= f(x_t) - 2\eta(\gamma\lambda - \eta\Gamma^2\Lambda^2)(f(x_t) - f(w_*)) \\
&\quad + 2\eta^2\Gamma^2\Lambda^2(f(w_k) - f(w_*)).
\end{aligned}$$

Taking expectations over all random variables, summing over $t = 0, \ldots, m-1$, and using a telescoping sum gives

$$\begin{aligned}
&\mathbb{E}[f(x_m)] \\
&\leq \mathbb{E}[f(x_0)] + 2m\eta^2\Gamma^2\Lambda^2\mathbb{E}[f(w_k) - f(w_*)] \\
&\quad - 2\eta(\gamma\lambda - \eta\Gamma^2\Lambda^2)\left(\sum_{t=0}^{m-1}\mathbb{E}[f(x_t)] - mf(w_*)\right) \\
&= \mathbb{E}[f(w_k)] + 2m\eta^2\Gamma^2\Lambda^2\mathbb{E}[f(w_k) - f(w_*)] \\
&\quad - 2m\eta(\gamma\lambda - \eta\Gamma^2\Lambda^2)\mathbb{E}[f(w_{k+1}) - f(w_*)].
\end{aligned}$$

Rearranging the above gives

$$\begin{aligned}
0 &\leq \mathbb{E}[f(w_k) - f(x_m)] + 2m\eta^2\Gamma^2\Lambda^2\mathbb{E}[f(w_k) - f(w_*)] \\
&\quad - 2m\eta(\gamma\lambda - \eta\Gamma^2\Lambda^2)\mathbb{E}[f(w_{k+1}) - f(w_*)] \\
&\leq \mathbb{E}[f(w_k) - f(w_*)] + 2m\eta^2\Gamma^2\Lambda^2\mathbb{E}[f(w_k) - f(w_*)] \\
&\quad - 2m\eta(\gamma\lambda - \eta\Gamma^2\Lambda^2)\mathbb{E}[f(w_{k+1}) - f(w_*)] \\
&= (1 + 2m\eta^2\Gamma^2\Lambda^2)\mathbb{E}[f(w_k) - f(w_*)] \\
&\quad - 2m\eta(\gamma\lambda - \eta\Gamma^2\Lambda^2)\mathbb{E}[f(w_{k+1}) - f(w_*)].
\end{aligned}$$

The second inequality follows from the fact that $f(w_*) \leq f(x_m)$. Using the fact that $\eta < \gamma\lambda/(2\Gamma^2\Lambda^2)$, it follows that

$$\begin{aligned}
&\mathbb{E}[f(w_{k+1}) - f(w_*)] \\
&\leq \frac{1 + 2m\eta^2\Gamma^2\Lambda^2}{2m\eta(\gamma\lambda - \eta\Gamma^2\Lambda^2)}\mathbb{E}[f(w_k) - f(w_*)].
\end{aligned}$$

Since we chose m and η to satisfy Equation 6.8, it follows that the rate α is less than one. This completes the proof. □

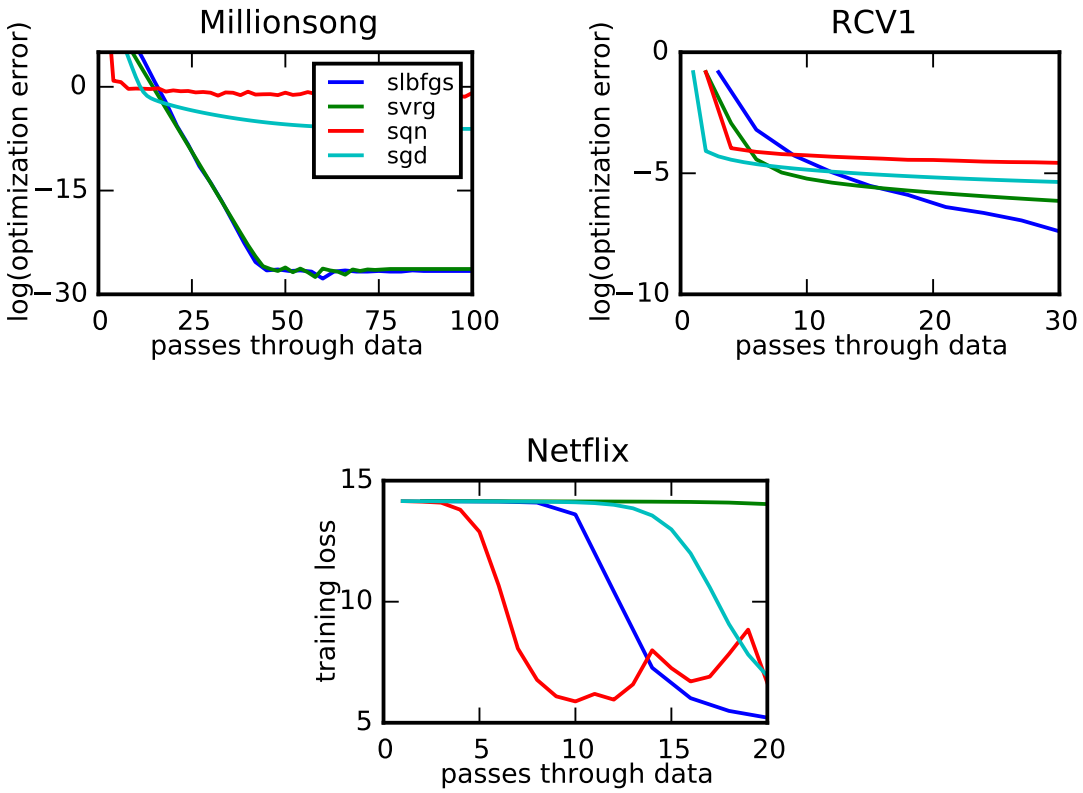


Figure 6.1: The left figure plots the log of the optimization error as a function of the number of passes through the data for SLBFGS, SVRG, SQN, and SGD for a ridge regression problem (Millionsong). The middle figure does the same for a support vector machine (RCV1). The right plot shows the training loss as a function of the number of passes through the data for the same algorithms for a matrix completion problem (Netflix).

6.5 Related Work

There is a large body of work that attempts to improve on stochastic gradient descent by reducing variance. [117] propose stochastic dual coordinate ascent (SDCA). [107] propose the stochastic average gradient method (SAG). [59] propose the stochastic variance reduced gradient (SVRG). [132] develop an approach based on the construction of control variates. More recently, [43] devise an online version of SVRG that uses streaming estimates of the gradient to perform variance reduction.

Similarly, a number of stochastic quasi-Newton methods have been proposed. [18] propose a variant of stochastic gradient descent that makes use of second order information. [79] analyze the straightforward application of L-BFGS in the stochastic setting and prove a $O(1/k)$ convergence rate in the strongly-convex setting. [23] propose a modified version of L-BFGS in the stochastic setting and prove a $O(1/k)$ convergence rate in the strongly-convex setting. [120] propose a stochastic quasi-Newton method for minimizing sums of functions

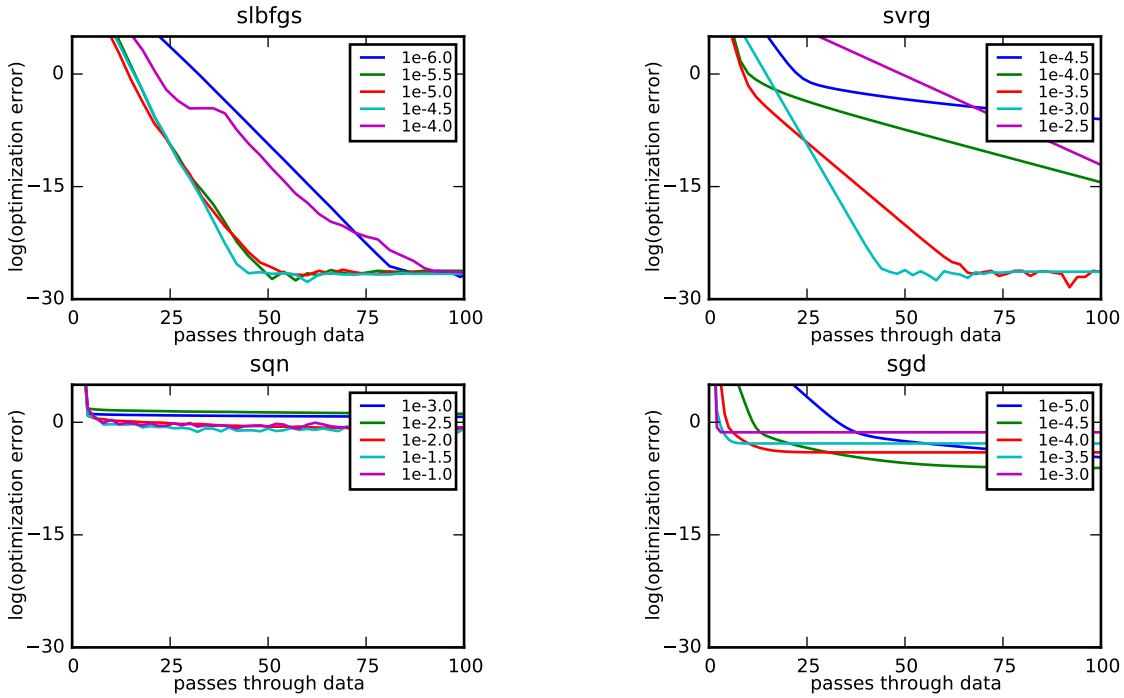


Figure 6.2: These figures show the log of the optimization error for SLBFGS, SVRG, SQN, and SGD on a ridge regression problem (millionsong) for a wide range of step sizes.

by maintaining a separate approximation of the inverse Hessian for each function in the sum. [111] develop a stochastic version of L-BFGS for the online convex optimization setting. [133] prove the convergence of various stochastic quasi-Newton methods in the nonconvex setting. Our work differs from the preceding in that we guarantee a linear rate of convergence.

[73] independently propose a variance-reduction procedure to speed up stochastic quasi-Newton methods and to achieve a linear rate of convergence. Their approach to updating the inverse-Hessian approximation is similar to that of L-BFGS, whereas our method leverages Hessian-vector products to stabilize the approximation.

6.6 Experimental Results

To probe our theoretical results, we compare Algorithm 1 (SLBFGS) to the stochastic variance-reduced gradient method (SVRG) [59], the stochastic quasi-Newton method (SQN) [23], and stochastic gradient descent (SGD). We evaluate these algorithms on several popular machine learning models, including ridge regression, support vector machines, and matrix completion. Our experiments show the effeciveness of the algorithm on real-world problems that are not neccessarily (strongly) convex.

Because SLBFGS and SVRG require computations of the full gradient, each epoch requires an additional pass through the data. Additionally, SLBFGS and SQN require Hessian-

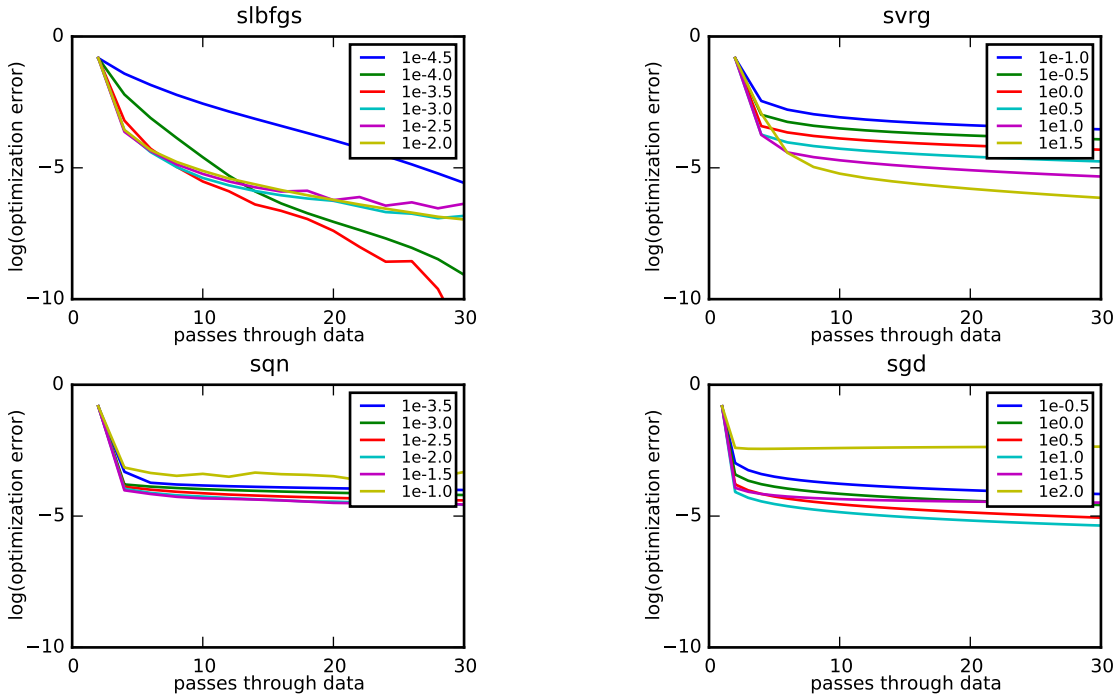


Figure 6.3: These figures show the log of the optimization error for SLBFGS, SVRG, SQN, and SGD on a support vector machine (RCV1) for a wide range of step sizes.

vector-product calculations, each of which is about as expensive as a gradient calculation [99]. The number of Hessian-vector-product computations per epoch introduced by this is $(b_H N)/(bL)$, which in our experiments is either N or $2N$. To incorporate these additional costs, our plots show error with respect to the number of passes through the data (that is, the number of gradient or Hessian-vector-product computations divided by N). For this reason, the first iterations of SLBFGS, SVRG, SQN, and SGD all begin at different times, with SGD appearing first and SLBFGS appearing last.

For all experiments, we set the batch size b to either 20 or 100, we set the Hessian batch size b_H to $10b$ or $20b$, we set the Hessian update interval L to 10, we set the memory size M to 10, and we set the number of stochastic updates m to N/b. We optimize the learning rate via grid search. SLBFGS and SVRG use a constant step size. For SQN and SGD, we try three different step-size schemes: constant, $1/\sqrt{t}$, and $1/t$, and we report the best one. All experiments are initialized with a vector of zeros, except for the matrix completion problem, where in order to break symmetry, we initialize the experiments with a vector of standard normal random variables scaled by 10^{-5}.

First, we performed ridge regression on the millionsong dataset [16] consisting of approximately 4.6×10^5 data points. We set the regularization parameter $\lambda = 10^{-3}$. In this experiment, both SLBFGS and SVRG rapidly solve the problem to high levels of precision. Second, we trained a support vector machine on RCV1 [65], with approximately 7.8×10^6

data points. We set the regularization parameter to $\lambda = 0$. In this experiment, SGD and SQN make more progress initially as expected, but SLBFGS finds a better optimum. Third, we solve a nonconvex matrix completion problem on the Netflix Prize dataset, as formulated in [103], with approximately 10^8 data points. We set the regularization parameter to $\lambda = 10^{-4}$. The poor performance of SVRG and SGD on this problem may be accounted for by the fact that the algorithms are initialized near the vector of all zeros, which is a stationary point (though not the optimum). Presumably the use of curvature information helps SLBFGS and SQN escape the neighborhood of the all zeros vector faster than SVRG and SGD.

Figure 6.1 plots a comparison of these methods on the three problems. For the convex problems, we plot the logarithm of the optimization error with respect to a precomputed reference solution. For the nonconvex problem, we simply plot the objective value as the global optimum is not necessarily known.

Robustness to Choice of Step Size

In this section, we illustrate that SLBFGS performs well on convex problems for a large range of step sizes. The windows in which SVRG, SQN, and SGD perform well are much narrower. In Figure 6.2, we plot the performance of SLBFGS, SVRG, SQN, and SGD for ridge regression on the millionsong dataset for step sizes varying over a couple orders of magnitude. In Figure 6.3, we show a similar plot for a support vector machine on the RCV1 dataset. In both cases, SLBFGS performs well, solving the problem to a high degree of precision over a large range of step sizes, whereas the performance of SVRG, SQN, and SGD degrade much more rapidly with poor step-size choices.

6.7 Proofs of Preliminaries

Proof of Lemma 3

The analysis below closely follows many other analyses of the inverse Hessian approximation used in L-BFGS [94, 23, 79, 80], and we include it for completeness.

Note that $s_j^\top y_j = s_j \nabla^2 f_{\mathcal{T}_j}(u_j) s_j$, it follows from Assumption 2 that

$$\lambda \|s_j\|^2 \leq s_j^\top y_j \leq \Lambda \|s_j\|^2. \tag{6.11}$$

Similarly, letting $z_j = (\nabla^2 f_{\mathcal{T}_j}(u_j))^{1/2} s_j$ and noting that

$$\frac{\|y_j\|^2}{s_j^\top y_j} = \frac{z_j^\top \nabla^2 f_{\mathcal{T}_j}(u_j) z_j}{z_j^\top z_j},$$

Assumption 2 again implies that

$$\lambda \leq \frac{\|y_j\|^2}{s_j^\top y_j} \leq \Lambda. \tag{6.12}$$

Note that using the Sherman-Morrison-Woodbury formula, we can equivalently write Equation 6.4 in terms of the Hessian approximation $B_r = H_r^{-1}$ as

$$B_r^{(j)} = B_r^{(j-1)} - \frac{B_r^{(j-1)} s_j s_j^\top B_r^{(j-1)}}{s_j^\top B_r^{(j-1)} s_j} + \frac{y_j y_j^\top}{y_j^\top s_j}. \tag{6.13}$$

We will begin by bounding the eigenvalues of B_r. We will do this indirectly by bounding the trace and determinant of B_r. We have

$$\begin{aligned}
\operatorname{tr}(B_r^{(j)}) &= \operatorname{tr}(B_r^{(j-1)}) - \frac{\operatorname{tr}(B_r^{(j-1)} s_j s_j^\top B_r^{(j-1)})}{s_j^\top B_r^{(j-1)} s_j} + \frac{\operatorname{tr}(y_j y_j^\top)}{y_j^\top s_j} \\
&= \operatorname{tr}(B_r^{(j-1)}) - \frac{\|B_r^{(j-1)} s_j\|^2}{s_j^\top B_r^{(j-1)} s_j} + \frac{\|y_j\|^2}{y_j^\top s_j} \\
&\leq \operatorname{tr}(B_r^{(j-1)}) + \frac{\|y_j\|^2}{y_j^\top s_j} \\
&\leq \operatorname{tr}(B_r^{(j-1)}) + \Lambda.
\end{aligned}$$

The first equality follows from the linearity of the trace operator. The second equality follows from the fact that $\operatorname{tr}(AB) = \operatorname{tr}(BA)$. The fourth relation follows from Equation 6.12. Since

$$\operatorname{tr}(B_r^{(0)}) = d\frac{\|y_r\|^2}{s_r^\top y_r} \leq d\Lambda,$$

it follows inductively that

$$\operatorname{tr}(B_k) \leq (d + M)\Lambda.$$

Now to bound the determinant, we write

$$\begin{aligned}
\det(B_r^{(j)}) &= \det(B_r^{(j-1)}) \\
&\quad \det\left(I - \frac{s_j s_j^\top B_r^{(j-1)}}{s_j^\top B_r^{(j-1)} s_j} + \frac{(B_r^{(j-1)})^{-1} y_j y_j^\top}{y_j^\top s_j}\right) \\
&= \det(B_r^{(j-1)}) \frac{y_j^\top s_j}{s_j^\top B_r^{(j-1)} s_j} \\
&= \det(B_r^{(j-1)}) \frac{y_j^\top s_j}{\|s_j\|^2} \frac{\|s_j\|^2}{s_j^\top B_r^{(j-1)} s_j} \\
&\geq \det(B_r^{(j-1)}) \frac{\lambda}{\lambda_{\max}(B_r^{(j-1)})} \\
&\geq \det(B_r^{(j-1)}) \frac{\lambda}{\operatorname{tr}(B_r^{(j-1)})} \\
&\geq \det(B_r^{(j-1)}) \frac{\lambda}{(d + M)\Lambda}.
\end{aligned}$$

The first equality uses $\det(AB) = \det(A)\det(B)$. The second equality follows from the identity

$$\begin{aligned} &\det(I + u_1 v_1^\top + u_2 v_2^\top) \\ &= (1 + u_1^\top v_1)(1 + u_2^\top v_2) - (u_1^\top v_2)(v_1^\top u_2) \end{aligned} \tag{6.14}$$

by setting $u_1 = -s_j$, $v_1 = (B_r^{(j-1)} s_j)/(s_j^\top B_r^{(j-1)} s_j)$, $u_2 = (B_r^{(j-1)})^{-1} y_j$, and $v_2 = y_j/(y_j^\top s_j)$. See [[]Lemma 7.6]dennis1977quasi for a proof, or simply note that Equation 6.14 follows from two applications of the identity $\det(A + uv^\top) = (1 + v^\top A^{-1} u)\det(A)$ when $I + u_1 v_1^\top$ is invertible and by continuity when it isn't. The third equality follows by multiplying the numerator and denominator by $\|s_j\|^2$. The fourth relation follows from Equation 6.11 and from the fact that $s_j^\top B_r^{(j-1)} s_j \leq \lambda_{\max}(B_r^{(j-1)})\|s_j\|^2$. The fifth relation uses the fact that the largest eigenvalue of a positive definite matrix is bounded by its trace. The sixth relation uses the previous bound on $\mathrm{tr}(B_r^{(j-1)})$. Since

$$\det(B_r^{(0)}) = \left(\frac{\|y_r\|^2}{s_r^\top y_r}\right)^d \geq \lambda^d,$$

it follows inductively that

$$\det(B_r) \geq \frac{\lambda^{d+M}}{((d+M)\Lambda)^M}.$$

Proof of Lemma 4

Using Lemma 3 as well as the fact that H_r is positive definite, we have

$$\lambda_{\max}(B_r) \leq \mathrm{tr}(B_r) \leq (d+M)\Lambda.$$

and

$$\lambda_{\min}(B_r) \geq \frac{\det(B_r)}{\lambda_{\max}(B_r)^{d-1}} \geq \frac{\lambda^{d+M}}{((d+M)\Lambda)^{d+M-1}}.$$

Since we defined $B_r = H_r^{-1}$, it follows that

$$\frac{1}{(d+M)\Lambda} I \preceq H_r \preceq \frac{((d+M)\Lambda)^{d+M-1}}{\lambda^{d+M}} I.$$

Proof of Lemma 6

Define the function $g_{\mathcal{S}}(w) = f_{\mathcal{S}}(w) - f_{\mathcal{S}}(w_*) - \nabla f_{\mathcal{S}}(w_*)^\top (w - w_*)$ to get the linearization of $f_{\mathcal{S}}$ around the optimum w_*, and note that $g_{\mathcal{S}}$ is minimized at w_*. It follows that for any w, we have

$$0 = g_{\mathcal{S}}(w_*) \leq g_{\mathcal{S}}\left(w - \frac{1}{\Lambda}\nabla g_{\mathcal{S}}(w)\right) \leq g_{\mathcal{S}}(w) - \frac{1}{2\Lambda}\|\nabla g_{\mathcal{S}}\|^2.$$

Rearranging, we have

$$\begin{aligned}&\|\nabla f_{\mathcal{S}}(w) - \nabla f_{\mathcal{S}}(w_*)\|^2 \\ &\leq 2\Lambda(f_{\mathcal{S}}(w) - f_{\mathcal{S}}(w_*) - \nabla f_{\mathcal{S}}(w_*)^\top (w - w_*)).\end{aligned}$$

Averaging over all possible minibatches $\mathcal{S} \subseteq \{1, \ldots, N\}$ of cardinality b and using the fact that $\nabla f(w_*) = 0$, we see that

$$\begin{aligned}&\binom{N}{b}^{-1} \sum_{|\mathcal{S}|=b} \|\nabla f_{\mathcal{S}}(w) - \nabla f_{\mathcal{S}}(w_*)\|^2 \\ &\leq 2\Lambda(f(w) - f(w_*)).\end{aligned} \tag{6.15}$$

Now, let $\mu_k = \nabla f(w_k)$ and $v_t = \nabla f_{\mathcal{S}}(x_t) - \nabla f_{\mathcal{S}}(w_k) + \mu_k$. Conditioning on $\mathcal{F}_{k,t}$ and taking an expectation with respect to $\mathcal{S}$, we find

$$\begin{aligned}\mathbb{E}_{k,t}[\|v_t\|^2] &\leq 2\mathbb{E}_{k,t}[\|\nabla f_{\mathcal{S}}(x_t) - \nabla f_{\mathcal{S}}(w_*)\|^2] \\ &\quad + 2\mathbb{E}_{k,t}[\|\nabla f_{\mathcal{S}}(w_k) - \nabla f_{\mathcal{S}}(w_*) - \mu_k\|^2] \\ &\leq 2\mathbb{E}_{k,t}[\|\nabla f_{\mathcal{S}}(x_t) - \nabla f_{\mathcal{S}}(w_*)\|^2] \\ &\quad + 2\mathbb{E}_{k,t}[\|\nabla f_{\mathcal{S}}(w_k) - \nabla f_{\mathcal{S}}(w_*)\|^2] \\ &\leq 4\Lambda(f(x_t) - f(w_*) + f(w_k) - f(w_*)).\end{aligned} \tag{6.16}$$

The first inequality uses the fact that $\|a + b\|^2 \leq 2\|a\|^2 + 2\|b\|^2$. The second inequality follows by noting that $\mu_k = \mathbb{E}_{k,t}[\nabla f_{\mathcal{S}}(w_k) - \nabla f_{\mathcal{S}}(w_*)]$ and that $\mathbb{E}[\|\xi - \mathbb{E}[\xi]\|^2] \leq \mathbb{E}[\|\xi\|^2]$ for any random variable ξ. The third inequality follows from Equation 6.15.

6.8 Discussion

This chapter introduces a stochastic version of L-BFGS and proves a linear rate of convergence in the strongly convex case. Theorem 7 captures the qualitatively linear rate of convergence of SLBFGS, which is reflected in our experimental results. We expect SLBFGS to outperform other stochastic first-order methods in poorly conditioned settings where curvature information is valuable as well in settings where we wish to solve the optimization problem to high precision.

There are a number of interesting points to address in future work. The proof of Theorem 7 and many similar proofs used to analyze quasi-Newton methods result in constants that scale poorly with the problem size. At a deeper level, the point of studying quasi-Newton methods is to devise algorithms that lie somewhere along the spectrum from gradient descent to Newton's method, reaping the computational benefits of gradient descent and the rapid convergence of Newton's method. Many of the proofs in the literature, including the proof of Theorem 7, bound the extent to which the quasi-Newton method deviates from gradient descent by bounding the extent to which the inverse Hessian approximation deviates from

the identity matrix. Those bounds are then used to show that the quasi-Newton method does not perform too much worse than gradient descent. A future avenue of research is to study if stochastic quasi-Newton methods can be designed that provably exhibit superlinear convergence as has been done in the non-stochastic case.

Chapter 7

Conclusion

In this thesis we have described Ray, a general-purpose distributed system that can be used to implement a wide variety of distributed workloads. This is in contrast to most widely used distributed systems like Hadoop, Apache Spark, Apache Flink, Kafka, Distributed TensorFlow, etc. which are built with a specific use case in mind (e.g. data processing, streaming, event processing, distributed machine learning training, etc.). These systems typically expose APIs that are optimized for the specific workloads the system supports but make it hard to build applications or libraries outside of the use cases they are designed for. As a result, most end-to-end applications are built by gluing many distributed systems together. This incurs overheads: Developers and system administrators need to be trained to use and deploy these systems, separate resources need to be allocated for them, data needs to be converted, they each have their own failure handling mechanisms that need to be conciliated. For new applications (like distributed reinforcement learning or before that, distributed machine learning), it is often necessary to build new distributed systems from scratch and rewrite much of the logic in existing distributed systems like scheduling, data transfers, error handling and fault tolerance.

Our approach is different: We provide a simple API based on well-known programming constructs: Functions and classes. This has several advantages: First it makes it easy for developers to port existing serial code to our system. This is important because often systems are initially written serially and scaled up only when the need arises. In fact there is a lot of code already written that people would like to parallelize and Ray has a good programming model to do so. Second, it allows to express arbitrary parallelism patterns and therefore allows to express all distributed applications. In fact, a number of powerful libraries have already been developed for Ray, including for distributed reinforcement learning [68], hyperparameter optimization [69], data processing [100], online planning [7] and traffic control [136]. A few other ones, including for streaming, model serving and distributed training are in the works. Because we can express all these workloads in one common framework, Ray is also a great platform for end-to-end applications like reinforcement learning and online learning.

However, computer systems are never good enough and theses are never complete. There

are a number of directions we would like to see Ray develop and we will outline some of them below.

Future Work

Implementation improvements. While the Ray programming model is very general, there are currently a few limitations in the Ray implementation that should be addressed. Currently, Ray supports workloads with runtimes on the order of several days or weeks well, but there are certain long-running workloads (say on the order of months or years) that are not well supported. There are a few improvements that would facilitate such longer running workloads:

- Using a distributed garbage collection scheme based on reference counting, ownership or a full garbage collector instead of the local least recently used (LRU) eviction policy which is currently deployed. This will make sure that the system will not evict objects that are very infrequently accessed.
- Making it possible to impose memory limits on Ray workers and actors for shared memory, in-process memory and message queues. This will make sure that code with memory leaks cannot use up all memory in the system and crash other important processes as a result.
- Designing the GCS in such a way that no tables are stored there which grow substantially during the execution of a program (this includes the profiling table and the task table). While we handle growing tables by removing old keys, it is possible to construct workloads that need to access already removed keys. Instead we should make sure such tables are written to disk or get rid of them entirely.
- Making it possible to replicate the GCS or storing it in a durable or replicated database. This will allow the system to survive even if the node(s) that run the GCS go down. This is challenging because we need low latency at high throughput and also the GCS needs to provide consistency for reads and writes.

Each of these improvements can be done incrementally within the current architecture.

A more radical approach is also possible. It would be desirable to simplify the system architecture and write the system in such a way that there is a small "trusted base" of code that if correct will guarantee the correct execution of Ray programs (including isolation and fault tolerance properties). In that case we could attempt to formally define the semantics of Ray programs and prove that the implementation adheres to the specification with program verification tools. This would make it possible to write distributed mission critical software with the highest demands for reliability in Ray.

A serverless runtime for Ray. The Ray programming model is very general and more powerful than existing serverless models like Amazon's *Lambda*, Microsoft's *Azure Functions* or Google's *Cloud Functions*. These struggle to support latency and throughput sensitive applications like distributed machine learning training and interactive model serving, online learning, streaming or reinforcement learning. Ray on the other hand supports latency and throughput sensitive applications well and also enables stateful computation and placement control via custom resources. These allow expressing a very wide variety of applications, including ones that require

- co-location of data and compute,
- high performance data transfer patterns or
- specialized hardware accelerators.

An interesting future research challenge is to modify Ray's runtime to be fully serverless, i.e. allow multi-tenant environments with very strong isolation guarantees between tenants, and allow pay-what-you-use. This will enable programmers to not worry about machines and concentrate on building powerful cloud applications.

Building higher level distributed primitives and libraries. As we described earlier, with Ray we shift the paradigm of building distributed software from gluing together specialized distributed systems to building libraries on top of one distributed system by composing them to build distributed applications. Thinking about the right primitives and libraries to build to facilitate this paradigm is a great avenue for further research. Concepts like worker pools, distributed data structures, synchronization and locking primitives, scheduling primitives (including exposing control over data aware scheduling), different strategies for fault recovery etc. can be extracted from existing distributed systems and put into Ray libraries to simplify writing distributed applications that need these primitives. There are also a number of concrete distributed libraries that we have not built yet that would be valuable, for example for scientific programming or web crawling.

An event based API. There is an increasing number of applications that are event based in nature. Examples are backends for internet of things (IoT) applications. While these can be implemented in the current programming model by calling methods in an actor whenever the event happens, there are more natural programming models to express them. We think reactive programming [12] would be a good fit to support such event based applications well. Coming up with a proposal for an an API, implementing it on top of the current runtime and validating it by writing some distributed event based applications would be an interesting research project.

Bibliography

[1] Martín Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*. 2016.

[2] Alekh Agarwal et al. "A Multiworld Testing Decision Service". In: *arXiv preprint arXiv:1606.03966* (2016).

[3] Alekh Agarwal et al. "A reliable effective terascale linear learning system". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1111–1133.

[4] "Akka". In: URL: `http://doc.akka.io/docs/akka/current/scala/actors.html`.

[5] Peter Alvaro et al. "BOOM Analytics: exploring data-centric, declarative programming for the cloud". In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 223–236.

[6] Dario Amodei et al. "Deep speech 2: End-to-end speech recognition in english and mandarin". In: *arXiv preprint arXiv:1512.02595* (2015).

[7] Thomas Anthony et al. "Policy Gradient Search: Online Planning and Expert Iteration without Search Trees". In: *CoRR* abs/1904.03646 (2019). arXiv: `1904.03646`. URL: `http://arxiv.org/abs/1904.03646`.

[8] *Apache Arrow*. `https://arrow.apache.org/`.

[9] Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1383–1394. ISBN: 978-1-4503-2758-9. DOI: `10.1145/2723372.2742797`. URL: `http://doi.acm.org/10.1145/2723372.2742797`.

[10] Joe Armstrong. "Making Reliable Distributed Systems in the Presence of Software Errors". PhD thesis. The Royal Institute of Technology, Stockholm, Sweden, Dec. 2003. URL: `citeseer.ist.psu.edu/armstrong03making.html`.

[11] Joe Armstrong et al. "Concurrent programming in ERLANG". In: (1993).

[12] Engineer Bainomugisha et al. "A survey on reactive programming." In: *ACM Comput. Surv.* 45.4 (2013), 52:1–52:34. URL: `http://dblp.uni-trier.de/db/journals/csur/csur45.html#BainomugishaCCMM13`.

[13] Henry C. Baker Jr. and Carl Hewitt. "The Incremental Garbage Collection of Processes". In: *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: ACM, 1977, pp. 55–59. DOI: 10.1145/800228.806932. URL: http://doi.acm.org/10.1145/800228.806932.

[14] Charles Beattie et al. "DeepMind Lab". In: *arXiv preprint arXiv:1612.03801* (2016).

[15] Phil Bernstein et al. *Orleans: Distributed Virtual Actors for Programmability and Scalability*. Tech. rep. MSR-TR-2014-41. Mar. 2014. URL: https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/.

[16] Thierry Bertin-Mahieux et al. "The Million Song Dataset". In: *International Conference on Music Information Retrieval*. 2011.

[17] Robert D. Blumofe and Charles E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 0004-5411.

[18] Antoine Bordes, Léon Bottou, and Patrick Gallinari. "SGD-QN: Careful quasi-Newton stochastic gradient descent". In: *The Journal of Machine Learning Research* 10 (2009), pp. 1737–1754.

[19] Léon Bottou. "Large-scale machine learning with stochastic gradient descent". In: *International Conference on Computational Statistics*. 2010, pp. 177–186.

[20] Léon Bottou and Yann LeCun. "Large scale online learning". In: *Advances in neural information processing systems* 16 (2004), p. 217.

[21] Greg Brockman et al. "OpenAI gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[22] Sergey Bykov et al. "Orleans: Cloud computing for everyone". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM. 2011, p. 16.

[23] Richard H Byrd et al. "A stochastic quasi-Newton method for large-scale optimization". In: *arXiv preprint arXiv:1401.7020* (2014).

[24] Paris Carbone et al. "State Management in Apache Flink: Consistent Stateful Distributed Stream Processing". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1718–1729. ISSN: 2150-8097.

[25] Martin Casado et al. "Ethane: Taking Control of the Enterprise". In: *SIGCOMM Comput. Commun. Rev.* 37.4 (Aug. 2007), pp. 1–12. ISSN: 0146-4833. DOI: 10.1145/1282427.1282382. URL: http://doi.acm.org/10.1145/1282427.1282382.

[26] Dominik Charousset et al. "Native Actors: A scalable software platform for distributed, heterogeneous environments". In: *Proceedings of the 2013 workshop on Programming based on actors, agents, and decentralized control*. ACM. 2013, pp. 87–96.

[27] Tianqi Chen et al. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: *NIPS Workshop on Machine Learning Systems (LearningSys'16)*. 2016.

[28] Trishul Chilimbi et al. "Project Adam: Building an efficient and scalable deep learning training system". In: *11th USENIX Symposium on Operating Systems Design and Implementation*. 2014, pp. 571–582.

[29] Adam Coates et al. "Deep learning with COTS HPC systems". In: *Proceedings of The 30th International Conference on Machine Learning*. 2013, pp. 1337–1345.

[30] Daniel Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 613–627. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

[31] Daniel Crankshaw et al. "The missing piece in complex analytics: Low latency, scalable model management and serving with Velox". In: *arXiv preprint arXiv:1409.3809* (2014).

[32] *Dask Benchmarks*. http://matthewrocklin.com/blog/work/2017/07/03/scaling.

[33] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[34] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[35] Jeffrey Dean et al. "Large scale distributed deep networks". In: *Advances in Neural Information Processing Systems*. 2012, pp. 1223–1231.

[36] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. "SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives". In: *Advances in Neural Information Processing Systems 27*. 2014, pp. 1646–1654.

[37] Ron S Dembo, Stanley C Eisenstat, and Trond Steihaug. "Inexact Newton methods". In: *SIAM Journal on Numerical analysis* 19.2 (1982), pp. 400–408.

[38] Ron S Dembo and Trond Steihaug. "Truncated-Newton algorithms for large-scale unconstrained optimization". In: *Mathematical Programming* 26.2 (1983), pp. 190–212.

[39] Jack B. Dennis and David P. Misunas. "A Preliminary Architecture for a Basic Data-flow Processor". In: *Proceedings of the 2Nd Annual Symposium on Computer Architecture*. ISCA '75. New York, NY, USA: ACM, 1975, pp. 126–132. DOI: 10.1145/642089.642111. URL: http://doi.acm.org/10.1145/642089.642111.

[40] Yan Duan et al. "Benchmarking deep reinforcement learning for continuous control". In: *Proceedings of the 33rd International Conference on Machine Learning (ICML)*. 2016.

[41] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *The Journal of Machine Learning Research* 12 (2011), pp. 2121–2159.

[42] *EC2 Instance Pricing*. `https://aws.amazon.com/ec2/pricing/on-demand/`.

[43] R. Frostig et al. "Competing with the empirical risk minimizer in a single pass". In: *Conference on Learning Theory*. 2015.

[44] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, Sept. 2004, pp. 97–104.

[45] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: *ACM SIGOPS operating systems review*. Vol. 37. 5. ACM. 2003, pp. 29–43.

[46] Joseph E. Gonzalez et al. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 599–613. ISBN: 978-1-931971-16-4.

[47] Joseph E Gonzalez et al. "Graphx: Graph processing in a distributed dataflow framework". In: *Proceedings of OSDI*. 2014, pp. 599–613.

[48] Shixiang Gu* et al. "Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates". In: *IEEE International Conference on Robotics and Automation (ICRA 2017)*. 2017.

[49] Stefan Hadjis et al. "Caffe Con Troll: Shallow Ideas to Speed Up Deep Learning". In: *Proceedings of the Fourth Workshop on Data Analytics in the Cloud*. DanaC'15. Melbourne, VIC, Australia: ACM, 2015, 2:1–2:4. ISBN: 978-1-4503-3724-3. DOI: `10.1145/2799562.2799641`. URL: `http://doi.acm.org/10.1145/2799562.2799641`.

[50] Carl Hewitt, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence". In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI'73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: `http://dl.acm.org/citation.cfm?id=1624775.1624804`.

[51] Benjamin Hindman et al. "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 295–308. URL: `http://dl.acm.org/citation.cfm?id=1972457.1972488`.

[52] Qirong Ho et al. "More effective distributed ML via a stale synchronous parallel parameter server". In: *Advances in Neural Information Processing Systems*. 2013, pp. 1223–1231.

[53] C. A. R. Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (Aug. 1978), pp. 666–677. ISSN: 0001-0782. DOI: 10.1145/359576.359585. URL: http://doi.acm.org/10.1145/359576.359585.

[54] Dan Horgan et al. "Distributed Prioritized Experience Replay". In: *International Conference on Learning Representations* (2018). URL: https://openreview.net/forum?id=H1Dy---0Z.

[55] Forrest N. Iandola et al. "FireCaffe: near-linear acceleration of deep neural network training on compute clusters". In: *CoRR* abs/1511.00175 (2015). arXiv: 1511.00175. URL: http://arxiv.org/abs/1511.00175.

[56] Michael Isard et al. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 59–72. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273005. URL: http://doi.acm.org/10.1145/1272996.1273005.

[57] Michael Isard et al. "Dryad: distributed data-parallel programs from sequential building blocks". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 3. ACM. 2007, pp. 59–72.

[58] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[59] Rie Johnson and Tong Zhang. "Accelerating stochastic gradient descent using predictive variance reduction". In: *Advances in Neural Information Processing Systems*. 2013, pp. 315–323.

[60] Michael I. Jordan and Tom M Mitchell. "Machine learning: Trends, perspectives, and prospects". In: *Science* 349.6245 (2015), pp. 255–260.

[61] Diederik Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *International Conference on Learning Representations*. 2015.

[62] Diego Kreutz et al. "Software-defined networking: A comprehensive survey". In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76.

[63] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. 2012, pp. 1097–1105.

[64] Jonathan Leibiusky, Gabriel Eisbruch, and Dario Simonassi. *Getting Started with Storm*. O'Reilly Media, Inc., 2012. ISBN: 1449324010, 9781449324018.

[65] David D Lewis et al. "RCV1: A new benchmark collection for text categorization research". In: *The Journal of Machine Learning Research* 5 (2004), pp. 361–397.

[66] Mu Li et al. "Scaling Distributed Machine Learning with the Parameter Server". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO, 2014, pp. 583–598. ISBN: 978-1-931971-16-4.

[67] Mu Li et al. "Scaling distributed machine learning with the parameter server". In: *11th USENIX Symposium on Operating Systems Design and Implementation*. 2014, pp. 583–598.

[68] Eric Liang et al. "RLlib: Abstractions for Distributed Reinforcement Learning". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholm Sweden: PMLR, Oct. 2018, pp. 3053–3062. URL: `http://proceedings.mlr.press/v80/liang18b.html`.

[69] Richard Liaw et al. "Tune: A Research Platform for Distributed Model Selection and Training". In: *arXiv preprint arXiv:1807.05118* (2018).

[70] Dong C Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical Programming* 45.1-3 (1989), pp. 503–528.

[71] Moshe Looks et al. "Deep learning with dynamic computation graphs". In: *arXiv preprint arXiv:1702.02181* (2017).

[72] Yucheng Low et al. "GraphLab: A New Framework for Parallel Machine Learning". In: *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*. UAI'10. Catalina Island, CA, 2010, pp. 340–349. ISBN: 978-0-9749039-6-5.

[73] Aurelien Lucchi, Brian McWilliams, and Thomas Hofmann. "A Variance Reduced Stochastic Newton Method". In: *arXiv preprint arXiv:1503.08316* (2015).

[74] Grzegorz Malewicz et al. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146. ISBN: 978-1-4503-0032-2.

[75] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[76] Xiangrui Meng et al. "MLlib: Machine Learning in Apache Spark". In: *Journal of Machine Learning Research* 17.34 (2016), pp. 1–7. URL: `http://jmlr.org/papers/v17/15-237.html`.

[77] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International Conference on Machine Learning*. 2016.

[78] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[79] Aryan Mokhtari and Alejandro Ribeiro. "Global convergence of online limited memory BFGS". In: *arXiv preprint arXiv:1409.2045* (2014).

[80] Aryan Mokhtari and Alejandro Ribeiro. "RES: Regularized stochastic BFGS algorithm". In: *IEEE Transactions on Signal Processing* 62.23 (2014), pp. 6089–6104.

[81] Philipp Moritz, Robert Nishihara, and Michael Jordan. "A Linearly-Convergent Stochastic L-BFGS Algorithm". In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*. Ed. by Arthur Gretton and Christian C. Robert. Vol. 51. Proceedings of Machine Learning Research. Cadiz, Spain: PMLR, Sept. 2016, pp. 249–258. URL: http://proceedings.mlr.press/v51/moritz16.html.

[82] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 561–577. ISBN: 978-1-931971-47-8. URL: http://dl.acm.org/citation.cfm?id=3291168.3291210.

[83] Philipp Moritz et al. "SparkNet: Training Deep Networks in Spark". In: *arXiv e-prints*, arXiv:1511.06051 (Nov. 2015), arXiv:1511.06051. arXiv: 1511.06051 [stat.ML].

[84] Derek G. Murray et al. "CIEL: A Universal Execution Engine for Distributed Data-flow Computing". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 113–126. URL: http://dl.acm.org/citation.cfm?id=1972457.1972470.

[85] Derek G. Murray et al. "CIEL: A Universal Execution Engine for Distributed Data-flow Computing". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 113–126. URL: http://dl.acm.org/citation.cfm?id=1972457.1972470.

[86] Derek G. Murray et al. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 439–455. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522738. URL: http://doi.acm.org/10.1145/2517349.2522738.

[87] Derek G. Murray et al. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 439–455. ISBN: 978-1-4503-2388-8.

[88] Derek G Murray et al. "Naiad: a timely dataflow system". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 439–455.

[89] D.G. Murray. *A Distributed Execution Engine Supporting Data-dependent Control Flow*. University of Cambridge, 2012. URL: https://books.google.com/books?id=ZebqoQEACAAJ.

[90] Arun Nair et al. *Massively Parallel Methods for Deep Reinforcement Learning*. 2015.

[91] Yurii Nesterov. "Primal-dual subgradient methods for convex problems". In: *Mathematical Programming* 120.1 (2009), pp. 221–259.

[92] Andrew Ng et al. "Autonomous inverted helicopter flight via reinforcement learning". In: *Experimental Robotics IX* (2006), pp. 363–372.

[93] Robert Nishihara et al. "Real-Time Machine Learning: The Missing Pieces". In: *Workshop on Hot Topics in Operating Systems*. 2017.

[94] Jorge Nocedal and Stephen J Wright. *Numerical Optimization*. Springer, 2006.

[95] Cyprien Noel, Jun Shi, and Andy Feng. *Large Scale Distributed Deep Learning on Hadoop Clusters*. 2015. URL: http://yahoohadoop.tumblr.com/post/129872361846/large-scale-distributed-deep-learning-on-hadoop.

[96] OpenAI. *OpenAI Dota 2 1v1 bot*. https://openai.com/the-international/. 2017.

[97] *OpenAI Baselines: high-quality implementations of reinforcement learning algorithms*. https://github.com/openai/baselines.

[98] Kay Ousterhout et al. "Sparrow: Distributed, Low Latency Scheduling". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 69–84. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522716. URL: http://doi.acm.org/10.1145/2517349.2522716.

[99] Barak A Pearlmutter. "Fast Exact Multiplication by the Hessian". In: *Neural Computation* 6.1 (1994), pp. 147–160.

[100] Devin Petersohn and Anthony D. Joseph. *Scaling Interactive Data Science Transparently with Modin*. Tech. rep. Electrical Engineering and Computer Sciences, University of California at Berkeley, 2018.

[101] *PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration*. http://pytorch.org/.

[102] Hang Qu et al. "Canary: A Scheduling Architecture for High Performance Cloud Computing". In: *arXiv preprint arXiv:1602.01412* (2016).

[103] Benjamin Recht and Christopher Ré. "Parallel stochastic gradient algorithms for large-scale matrix completion". In: *Mathematical Programming Computation* 5.2 (2013), pp. 201–226.

[104] Robbert van Renesse and Fred B. Schneider. "Chain Replication for Supporting High Throughput and Availability". In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004.

[105] Herbert Robbins and Sutton Monro. "A stochastic approximation method". In: *The Annals of Mathematical Statistics* (1951), pp. 400–407.

[106] Matthew Rocklin. "Dask: Parallel Computation with Blocked algorithms and Task Scheduling". In: *Proceedings of the 14th Python in Science Conference*. Ed. by Kathryn Huff and James Bergstra. 2015, pp. 130–136.

[107] Nicolas L. Roux, Mark Schmidt, and Francis R. Bach. "A Stochastic Gradient Method with an Exponential Convergence Rate for Finite Training Sets". In: *Advances in Neural Information Processing Systems*. 2012, pp. 2663–2671.

[108] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* (2015), pp. 1–42.

[109] Tim Salimans et al. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning". In: *arXiv preprint arXiv:1703.03864* (2017).

[110] Salvatore Sanfilippo. *Redis: An open source, in-memory data structure store*. `https://redis.io/`. 2009.

[111] Nicol N Schraudolph, Jin Yu, and Simon Günter. "A Stochastic Quasi-Newton Method for Online Convex Optimization". In: *International Conference on Artificial Intelligence and Statistics*. 2007, pp. 436–443.

[112] John Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *International Conference on Learning Representations (ICLR)* (2016).

[113] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[114] John Schulman et al. "Trust Region Policy Optimization". In: *ICML*. 2015, pp. 1889–1897.

[115] Malte Schwarzkopf et al. "Omega: Flexible, Scalable Schedulers for Large Compute Clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 351–364. ISBN: 978-1-4503-1994-2. DOI: `10.1145/2465351.2465386`. URL: `http://doi.acm.org/10.1145/2465351.2465386`.

[116] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *arXiv preprint arXiv:1802.05799* (2018).

[117] Shai Shalev-Shwartz and Tong Zhang. "Stochastic dual coordinate ascent methods for regularized loss". In: *The Journal of Machine Learning Research* 14.1 (2013), pp. 567–599.

[118] David Silver et al. "Deterministic policy gradient algorithms". In: *ICML*. 2014.

[119] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[120] Jascha Sohl-Dickstein, Ben Poole, and Surya Ganguli. "Fast large-scale optimization by unifying stochastic gradient and quasi-Newton methods". In: *International Conference on Machine Learning*. 2014.

[121] Evan R. Sparks et al. "KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics". In: *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. 2017, pp. 535–546. DOI: 10.1109/ICDE.2017.109. URL: https://doi.org/10.1109/ICDE.2017.109.

[122] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *International Conference on Machine Learning*. 2013, pp. 1139–1147.

[123] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[124] Christian Szegedy et al. "Going Deeper with Convolutions". In: *Computer Vision and Pattern Recognition (CVPR)*. 2015. URL: http://arxiv.org/abs/1409.4842.

[125] *TensorFlow Serving*. https://www.tensorflow.org/serving/.

[126] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH". In: *The International Journal of High Performance Computing Applications* 19.1 (2005), pp. 49–66.

[127] Yuandong Tian et al. "ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games". In: *Advances in Neural Information Processing Systems (NIPS)* (2017).

[128] Emanuel Todorov, Tom Erez, and Yuval Tassa. "Mujoco: A physics engine for model-based control". In: *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE. 2012, pp. 5026–5033.

[129] Leslie G Valiant. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (1990), pp. 103–111.

[130] Jur Van Den Berg et al. "Superhuman performance of surgical tasks by robots using iterative learning from human-guided demonstrations". In: *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE. 2010, pp. 2074–2081.

[131] Shivaram Venkataraman et al. "Drizzle: Fast and Adaptable Stream Processing at Scale". In: *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017.

[132] Chong Wang et al. "Variance reduction for stochastic gradient optimization". In: *Advances in Neural Information Processing Systems*. 2013, pp. 181–189.

[133] Xiao Wang, Shiqian Ma, and Wei Liu. "Stochastic Quasi-Newton Methods for Nonconvex Stochastic Optimization". In: *arXiv preprint arXiv:1412.1196* (2014).

[134] David Wentzlaff and Anant Agarwal. "Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores". In: *SIGOPS Oper. Syst. Rev.* 43.2 (Apr. 2009), pp. 76–85. ISSN: 0163-5980. DOI: 10.1145/1531793.1531805. URL: http://doi.acm.org/10.1145/1531793.1531805.

[135] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[136] Cathy Wu et al. "Flow: Architecture and Benchmarking for Reinforcement Learning in Traffic Control". In: *CoRR* abs/1710.05465 (2017). arXiv: 1710.05465. URL: http://arxiv.org/abs/1710.05465.

[137] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: http://doi.acm.org/10.1145/2934664.

[138] Matei Zaharia et al. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737. URL: http://doi.acm.org/10.1145/2517349.2522737.

[139] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[140] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[141] Matei Zaharia et al. "Spark: cluster computing with working sets". In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. Vol. 10. 2010, p. 10.

[142] Sixin Zhang, Anna Choromanska, and Yann LeCun. "Deep Learning with Elastic Averaging SGD". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 685–693. URL: http://dl.acm.org/citation.cfm?id=2969239.2969316.

[143] Yuchen Zhang and Michael I Jordan. "Splash: User-friendly Programming Interface for Parallelizing Stochastic Algorithms". In: *arXiv preprint arXiv:1506.07552* (2015).

[144] Martin Zinkevich et al. "Parallelized stochastic gradient descent". In: *Advances in Neural Information Processing Systems*. 2010, pp. 2595–2603.