On Systems and Algorithms for Distributed Machine Learning

Robert Nishihara

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-30
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-30.html

May 10, 2019

On Systems and Algorithms for Distributed Machine Learning

by

Robert K Nishihara

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
Assistant Professor Jonathan Ragan-Kelley

Spring 2019

On Systems and Algorithms for Distributed Machine Learning

Abstract

On Systems and Algorithms for Distributed Machine Learning

by

Robert K Nishihara

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael I. Jordan, Chair

The advent of algorithms capable of leveraging vast quantities of data and computational resources has led to the proliferation of systems and tools aimed to facilitate the development and usage of these algorithms. Hardware trends, including the end of Moore's Law and the maturation of cloud computing, have placed a premium on the development of scalable algorithms designed for parallel architectures. The combination of these factors has made distributed computing an integral part of machine learning in practice.

This thesis examines the design of systems and algorithms to support machine learning in the distributed setting. The distributed computing landscape today consists of many domain-specific tools. We argue that these tools underestimate the generality of many modern machine learning applications and hence struggle to support them. We examine the requirements of a system capable of supporting modern machine learning workloads and present a general-purpose distributed system architecture for doing so. In addition, we examine several examples of specific distributed learning algorithms. We explore the theoretical properties of these algorithms and see how they can leverage such a system.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis describes work I have had the opportunity to do at Berkeley, and there are many people I want to thank for their invaluable contributions.

First, my advisor Michael Jordan, for drawing me to Berkeley and encouraging me throughout the whole journey. His guidance has been indispensable, and his creation of such an extraordinary research environment gave me the opportunity to work with many other stellar peers and colleagues.

Ion Stoica, whom I am fortunate to have worked with closely. I have learned so much from his unparalleled drive and focus on trailblazing research. His intense focus on long-term and real-world impact has shaped how I approach my own work.

Philipp Moritz, who has been a friend and inspiration since the start of my PhD, and has deeply influenced my trajectory in grad school. He is someone who does not see a lot of obstacles and that has helped me rethink what is possible.

And everyone on 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 uniquely fruitful collaboration. You have taught me so much, and I have greatly enjoyed the camaraderie and teamwork.

I would like to thank my friends and colleagues in my research groups. SAIL has been an incredibly vibrant and enthusiastic group, and I have learned so much from my peers here, including Stefanie Jegelka, Ashia Wilson, Horia Mania, Mitchell Stern, Tamara Broderick, Ahmed El Alaoui, Esther Rolf, Akosua Busia, Chi Jin, Tijana Zrnic, Max Rabinovich, Nilesh Tripuraneni, Karl Krauth, Ryan Giordano, and Nick Boyd.

The RISELab, AMPLab, and BAIR have been nurturing and collaborative research environments as well as marvelous communities of friends. The diversity of research interests and expertise under one roof gave me the freedom to immerse myself in new areas, and that has made all the difference.

I also would like to thank Jonathan Ragan-Kelley and Ken Goldberg for being on my thesis committee and for their insights and advice over the years.

Outside of Berkeley, I am very fortunate to have been supported by many others along this journey. Thanks to Ryan Adams, who introduced me to research in machine learning and provided valuable guidance during my undergraduate years, Oren Rippel and Michael Gelbart, for their delightful friendship and for shaping my early experience in research, David Parkes, who first sparked my interest in machine learning and encouraged me to pursue research, Leon Bottou and Daniel Tarlow, for mentoring me and exposing me to research in the industrial setting, and Greg Brockman, Wojciech Zaremba, David Luan, John Schulman and everyone at OpenAI for the exciting work and welcoming environment.

My entire experience would have been greatly diminished without the companionship of many of my friends including Jeff Mahler, Alyssa Morrow, Yiren Lu, Jacob Andreas, Dan Ranard, Leah Weiss, Eric Larson, Sandy Huang, Sam Melton, Sam Schoenberg, Kirk Benson, Ana Klimovic, Lisa Yao, Judy Savitskaya, Ma'ayan Bresler, Madalina Persu, Chris

Maddison, Nate Sauder, Mehrdad Niknami, Mark Chen, Smitha Milli, Juliana Cherston, Ludwig Schmidt, Yifan Wu, Olivia Angiuli, Yasaman Bahri, Katarina Slama, Richard Shin, Vlad Feinberg, Sarah Bird, David Lopez-Paz, Yufei Zhao, Mattan Mansoor, Sabrina Siu, Lucy Stephenson, Dwight Crow, Marc Khoury, Charles Cary, Michael Webb, Roy Frostig, Lisha Li, and Jacob Steinhardt. Thank you for your presence in my life.

Finally, thanks to my parents, Catherine and Keith for imparting their excitement about the world, for giving me a love for math, and for raising me with their values and their commitment to the overall well-being of the world. They, and my sister Naomi, are a source of constant support and love and feedback. You mean the world to me.

Chapter 1

Introduction

The recent empirical success of machine learning techniques in many problem domains has led to a rapid expansion of the field and a flurry of work in both techniques for and applications of machine learning. The large-scale requirements of these techniques, both in terms of data and computational power, combined with the end of Moore's law, have placed an increased importance on parallel and distributed computing. As such, many tools have been developed to facilitate the development and usage of distributed machine learning applications. These include tools for performing core machine learning tasks like large-scale optimization, rapid gradient computation, and hyperparameter search. These also include tools for plumbing data in and out of machine learning algorithms and managing the surrounding ecosystem such as systems for batch data processing, streaming data processing, and prediction serving.

However, these tools are highly specialized. They handle specific scenarios well, but have been unable to efficiently support the diversity of machine learning applications that we see today. Many applications, including those in reinforcement learning and online learning, are forcing practitioners to abandon existing systems and build new ones. The reason for this is that applications in reinforcement learning, online learning, and many other domains exhibit a variety of computational patterns that span the use cases of many different specialized distributed systems. Neither a streaming system nor a model training system will be a perfect fit for an application that needs to perform both stream processing and model training in a tightly coupled manner. As a consequence, practitioners often find themselves doing one of two things.

- Practitioners will stitch multiple distinct distributed systems together to support their application.
- Practitioners will build new distributed systems from scratch to support their application.

The first approach gives rise to a number of practical challenges. Stitching together multiple distinct systems is not a straightforward exercise in composition. These systems are typically not designed to interface with one another and therefore don't expose clean

boundaries. For example, the systems often have different fault tolerance strategies that must be reconciled. If one system uses lineage-based fault tolerance and another uses a checkpointing strategy, practitioners will have to build additional logic into their applications to reconcile these differences. Furthermore, it can be difficult to move data efficiently across system boundaries, leading to performance challenges. Lastly, the overhead of learning how to use and manage many different distributed systems poses a high barrier to entry and an ongoing maintanence burden.

The second approach of building a new system can be even worse. By building a new application-specific distributed system, the application developer at least has the option of building a tool perfectly suited to his or her problem. However, this approach places substantial engineering burdens on the application developer. Instead of reasoning specifically about the algorithm and application at hand, the developer must solve common problems over and over. These problems include how to handle machine failures, what work to assign to which machine, and how to communicate and send data efficiently between machines. These problems have been repeatedly solved in many different existing systems, and yet their solutions have not been abstracted away and shared between systems.

In this thesis, we aim to design a distributed system capable of supporting modern machine learning applications. Our strategy is to design a system with a sufficient level of generality such that existing special-purpose systems can be expressed as libraries or applications on top of it. Such a system would allow machine learning practitioners to develop their distributed applications at a higher level of abstraction, by reasoning about their application logic and not about low-level systems concerns like failure handling, scheduling, and data movement.

Our approach for achieving generality is to use the same abstractions in the single-threaded and in the parallel settings. This approach allows a natural translation of workloads expressible in the single-threaded setting to workloads expressible in the distributed setting, and makes possible a seamless transition from prototyping on a laptop, to parallelizing an application across multiple cores on a single machine, to distributing the application across a large cluster.

The specialized nature of most distributed systems comes from the concepts that they introduce. For example, a data processing system may introduce a dataset as its core concept. A stream processing system may introduce a stream as its core concept. A hyerparameter search system may introduce a trial or experiment as its core concept. An automatic differentiation framework may introduce a differentiable computation graph as its core concept. From the perspective of generality, these concepts are limiting as different computational patterns must be coerced into unfamiliar abstractions.

Our proposed architecture aims to achieve generality be reusing familiar concepts from single-threaded programming in the distributed setting. In particular, two of the building blocks of single-threaded programs are functions and classes. We propose to provide simple translations of these concepts into the distributed setting as tasks and actors. This means that single-threaded applications that can be expressed using functions and classes can be ported to the distributed setting. Our proposed architecture has the following properties.

- A unified programming model that combines stateless tasks and stateful actors in a single dataflow graph.
- A horizontally scalable architecture that allows each component to be replicated or sharded to remove bottlenecks.
- A fault tolerant backend that can recover transparently from machine failures.

The prospect of a general-purpose system holds great promise, and it must strike a careful balance between low-level and high-level abstractions. By providing sufficiently low-level abstractions, such a system can allow a broad range of applications to be expressed, and by providing sufficiently high-level abstractions, the system can prevent those applications from needing to reason about standard distributed computing problems like scheduling, fault tolerance, and data movement. Additionally, a single system incurs less pedagogical overhead as users have fewer concepts and tools that they must learn to use.

In Chapter 2, we illustrate the properties and requirements that such a general system must satisfy in order to enable modern machine learning applications. In Chapter 3, we propose an architecture achieving these requirements. In Chapter 4, we present a case study showing how a variety of distributed training techniques can be implemented on top of this architecture. In Chapter 5, we examine a distributed algorithm for solving large-scale convex optimization problems and examine its theoretical properties. In Chapter 6, we examine a distributed optimization algorithm for solving submodular function optimization problems and examine its theoretical properties. In Chapter 7, we conclude and discuss future work.

Chapter 2

The Requirements of a System

Machine learning applications are increasingly deployed not only to serve predictions using static models, but also as tightly-integrated components of feedback loops involving dynamic, real-time decision making. These applications pose a new set of requirements, none of which are difficult to achieve in isolation, but the combination of which creates a challenge for existing distributed execution frameworks: computation with millisecond latency at high throughput, adaptive construction of arbitrary task graphs, and execution of heterogeneous kernels over diverse sets of resources. In this chapter, we assert that a new distributed execution framework is needed for such ML applications and propose a candidate approach with a proof-of-concept architecture that achieves a 63x performance improvement over a state-of-the-art execution framework for a representative application.[1]

2.1 Introduction

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 2.1 top), 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 2.1 bottom).

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

[1]Material in this chapter is based adapted from [114].

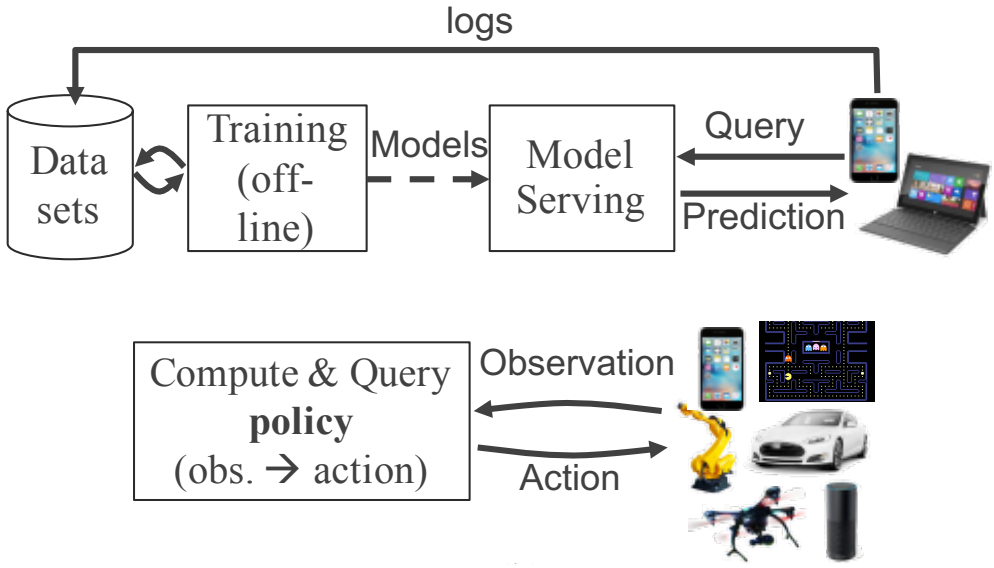


Figure 2.1: Diagrams of machine learning and reinforcement learning pipelines. The top figure shows a traditional ML pipeline (off-line training), and the bottom figure shows an example reinforcement learning pipeline: the system continously interacts with an environment to learn a policy, i.e., a mapping between observations and actions.

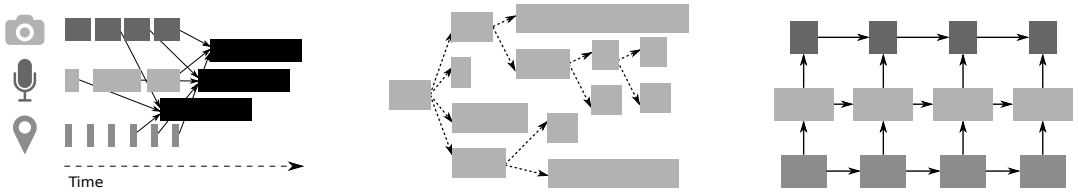


Figure 2.2: Example components of a real-time ML application. The left figure shows online processing of streaming sensory data to model the environment, the middle figure shows dynamic graph construction for Monte Carlo tree search (here tasks are simulations exploring sequences of actions), and the right figure shows heterogeneous tasks in recurrent neural networks. Different shades represent different types of tasks, and the task lengths represent their durations.

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.*

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 [36].
- **R2:** *High throughput.* The volume of micro-simulations required both for training [109] as well as for inference during deployment [135] 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 [40, 155] have gotten great mileage out of identifying and optimizing for common computational patterns, emerging ML applications require far greater flexibility [48].

- **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 2.5). We propose a flexible distributed programming model (Section 3.3) 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 3.4) 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.

2.2 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 (Figure 2.2 left). The robot is then controlled in real time using actions informed by a recurrent neural network (RNN) *policy* (Figure 2.2 right), as well as by Monte Carlo tree search (MCTS) and other online planning algorithms (Figure 2.2 middle). 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.

2.3 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 3.5).

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

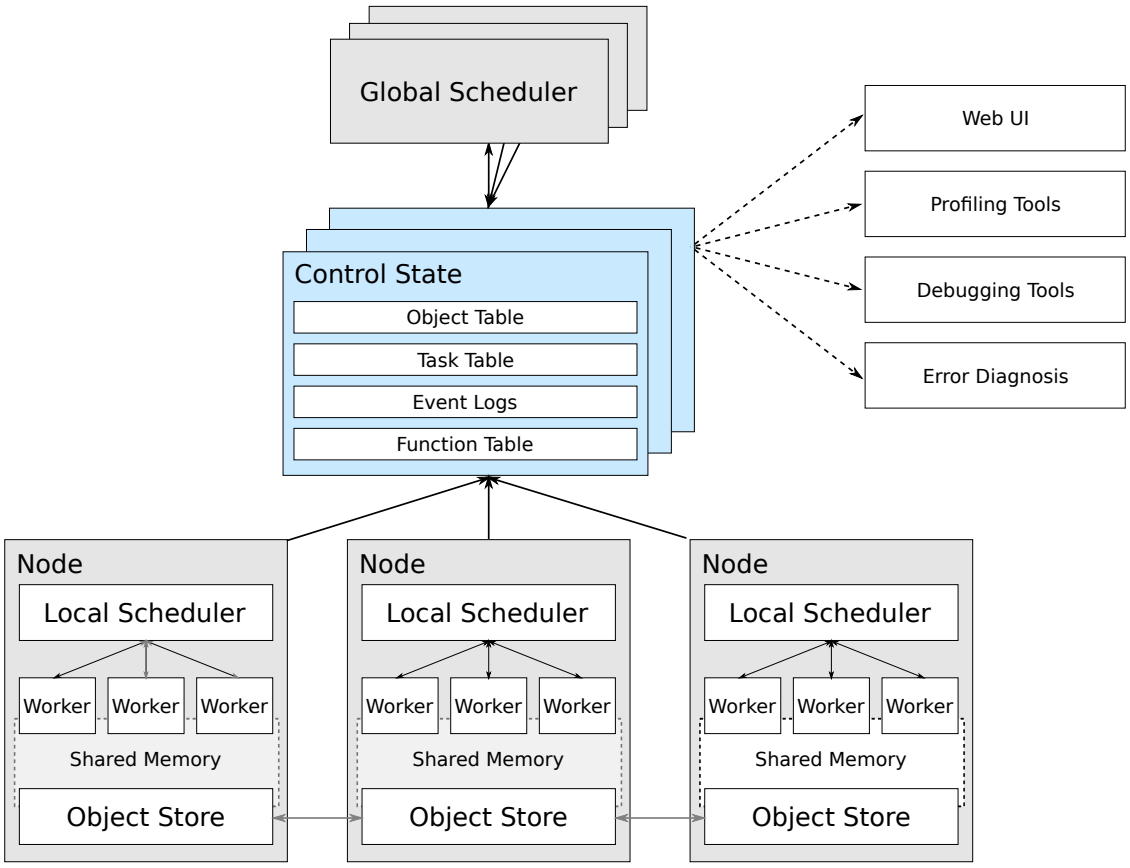


Figure 2.3: Proposed Architecture, with hybrid scheduling (Section 2.3) and a centralized control plane (Section 2.3).

Centralized Control State

As shown in Figure 3.5, our architecture relies on a logically-centralized control plane [90]. 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 [155]. 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

[2]In our implementation we employ Redis [130], although many other fault-tolerant key-value stores could be used.

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.

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 [152].

2.4 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 2.3.

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 GPU, or run the entire workload nested within a larger adaptive hyperparameter search. These changes are all straightforward using the API described in Section 3.3 and involve a few extra lines of code.

2.5 Related Work

Static dataflow systems [40, 155, 77, 108] 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 [40] and Spark [155], emphasize BSP execution, while others, like Dryad [77] and Naiad [108], support complex dependency structures (**R5**). Others, such as TensorFlow [1] and MXNet [32], 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 [107] and Dask [127] 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 [58] and actor-model variants Orleans [28] and Erlang [8] 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 [34, 5], 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.

[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.

2.6 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 3

Ray: A System for Machine Learning

The next generation of AI applications will continuously interact with the environment and learn from these interactions. These applications impose new and demanding systems requirements, both in terms of performance and flexibility. In this chapter, we consider these requirements and present Ray—a distributed system to address them. Ray implements a unified interface that can express both task-parallel and actor-based computations, supported by a single dynamic execution engine. To meet the performance requirements, Ray employs a distributed scheduler and a distributed and fault-tolerant store to manage the system's control state. In our experiments, we demonstrate scaling beyond 1.8 million tasks per second and better performance than existing specialized systems for several challenging reinforcement learning applications.[1]

3.1 Introduction

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 [40, 156, 77], streaming [**storm**, 29, 108], and graph [99, 100, 65] 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 [84]. 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

[1]Material in this chapter is based adapted from [105].

training time in a batch setting. Examples include TensorFlow [1], MXNet [32], and PyTorch [120].

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, 114]. 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 [138]. RL-based systems have already yielded remarkable results, such as Google's AlphaGo beating a human world champion [135], and are beginning to find their way into dialogue systems, UAVs [111], and robotic manipulation [69, 145].

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 [40], Apache Spark [156], and Dryad [77] do not support fine-grained simulation or policy serving. Task-parallel systems such as CIEL [107] and Dask [127] provide little support for distributed training and serving. The same is true for streaming systems such as Naiad [108] and Storm [**storm**]. Distributed deep-learning frameworks such as TensorFlow [1] and MXNet [32] do not naturally support simulation and serving. Finally, model-serving systems such as TensorFlow Serving [140] and Clipper [35] 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 [134] for distributed training, Clipper [35] for serving, and CIEL [107] 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 [142, 109, 135, 115, 129, 116]. 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 chapter, 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 [156, 77, 107]: (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 [35] and TensorFlow Serving [140], 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 [156], 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.

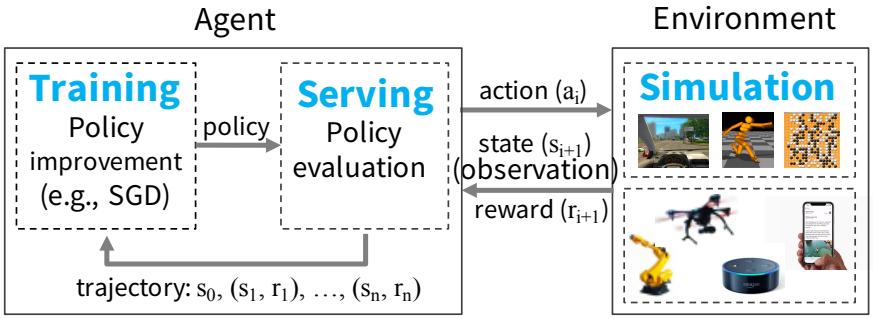


Figure 3.1: Example of an RL system.

3.2 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 3.1, 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.

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 3.2 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 3.1). 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 [93].

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

```
// 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 3.2: Typical RL pseudocode for learning a policy.

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

Name	Description
$futures = \textbf{f.remote}(args)$	Execute function f remotely. **f.remote**() can take objects or futures as inputs and returns one or more futures. This is non-blocking.
$objects = \textbf{ray.get}(futures)$	Return the values associated with one or more futures. This is blocking.
$ready\_futures = \textbf{ray.wait}(futures, k, timeout)$	Return the futures whose corresponding tasks have completed as soon as either k have completed or the timeout expires.
$actor = \textbf{Class.remote}(args)$ $futures = actor.\textbf{method.remote}(args)$	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 3.1: Ray API

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 3.2: Tasks vs. actors tradeoffs.

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 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,

[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.

```
@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 3.3: Python code implementing the example in Figure 3.2 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.

it should enable seamless integration with existing simulators [25, 18, 143] and deep learning frameworks [1, 32, 120, 83].

3.3 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

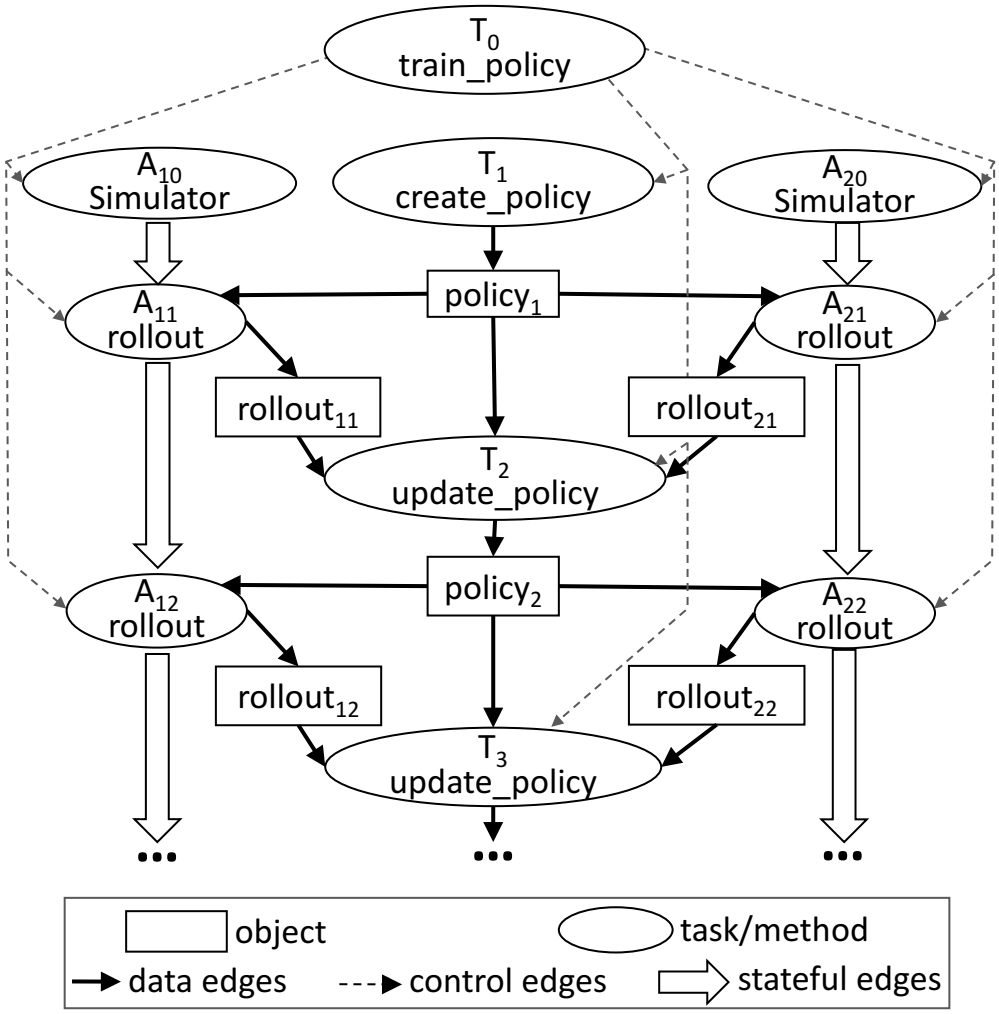


Figure 3.4: The task graph corresponding to an invocation of train_policy.remote() in Figure 3.3. Remote function calls and the actor method calls correspond to tasks in the task graph. The figure shows two actors. The method invocations for each actor (the tasks labeled A_{1i} and A_{2i}) have stateful 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.

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 [28] or Akka [3], 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 3.1 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 3.2 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 [93] 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 3.2), 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 3.4), as it enables multiple processes to invoke remote functions in a distributed fashion.

Computation Model

Ray employs a dynamic task graph computation model [42], 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 3.4) is constructed from a user program (Figure 3.3). This program uses the API in Table 3.1 to implement the pseudocode from Figure 3.2.

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 3.3): 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 3.4). 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 [156], 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 3.4).

3.4 Architecture

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

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.

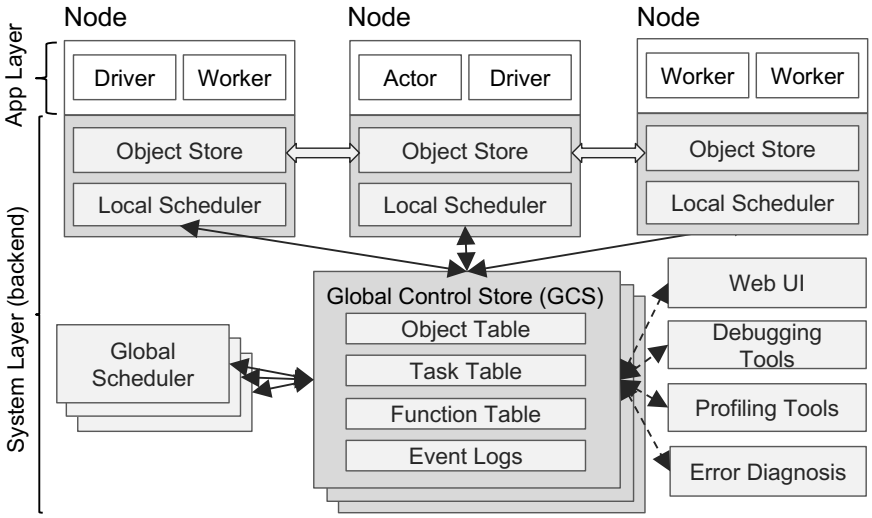


Figure 3.5: Ray's architecture consists of two parts: an *application* layer and a *system* layer. The application layer implements the API and the computation model described in Section 3.3, the system layer implements task scheduling and data management to satisfy the performance and fault-tolerance requirements.

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 [125] 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 [156, 153, 107, 77] 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 [156, 107, 127] couple these by storing object locations and sizes in a centralized scheduler, a natural design when the sched-

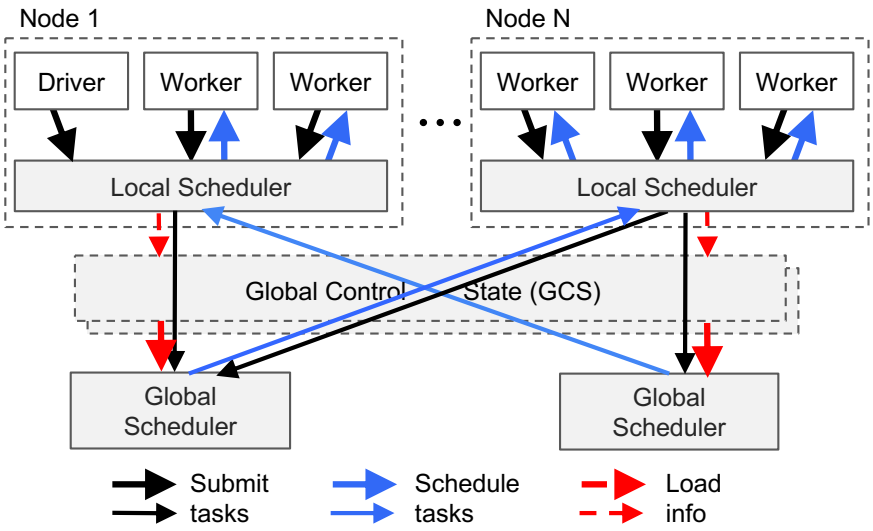


Figure 3.6: Bottom-up distributed scheduler. Tasks are submitted bottom-up, from drivers and workers to a local scheduler and forwarded to the global scheduler only if needed (Section 3.4). The thickness of each arrow is proportional to its request rate.

uler 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 3.2, 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 [156], CIEL [107], and Dryad [77] implement a centralized scheduler, which can provide locality but at latencies in the tens of ms. Distributed schedulers such as work stealing [23], Sparrow [118] and Canary [124] can achieve high scale, but they either don't consider data locality [23], or assume tasks belong to independent jobs [118], or assume the computation graph is known [124].

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 task's 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 [7].

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 [156], and Dryad [77], 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 [130] 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 [125] for fault tolerance. We 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 3.7 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 the top of Figure 3.7).

The top figure in Figure 3.7 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).

The bottom figure in Figure 3.7 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.

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

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

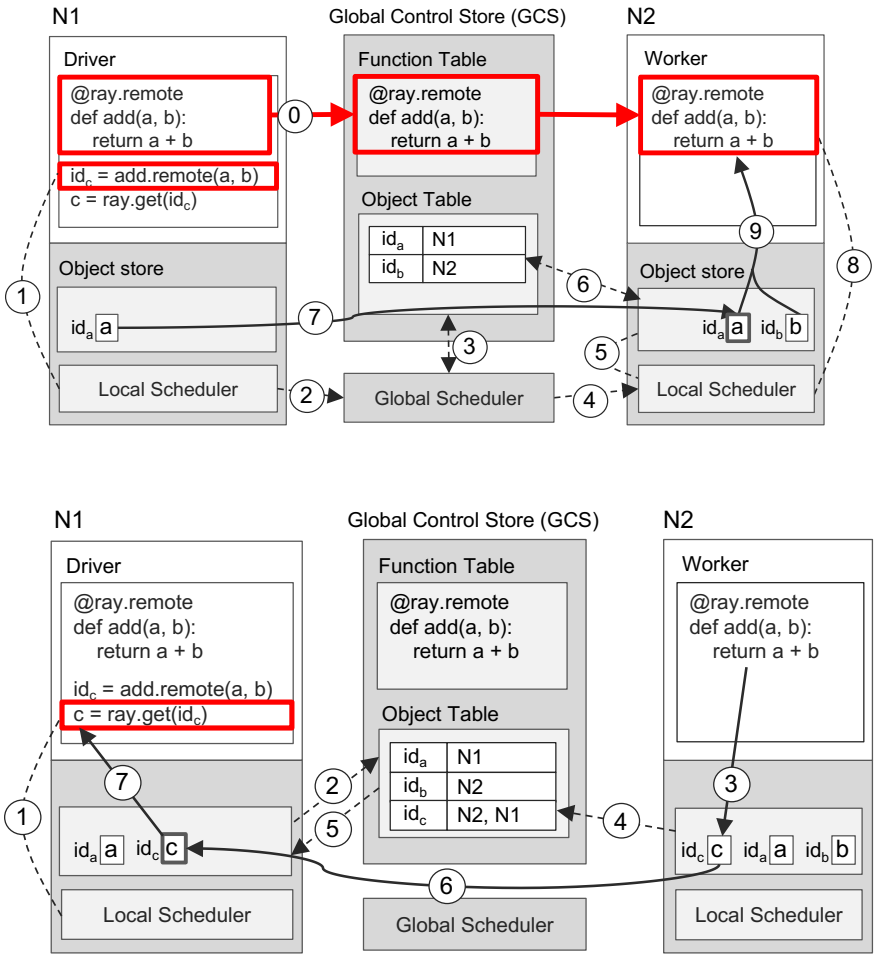


Figure 3.7: An end-to-end example that adds a and b and returns c. Solid lines are data plane operations and dotted lines are control plane operations. In the top figure, the function **add()** is registered with the GCS by node 1 ($N1$), invoked on $N1$, and executed on $N2$. In the bottom figure, $N1$ 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 $N1$.

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.

3.5 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 3.2? (Section 3.5)

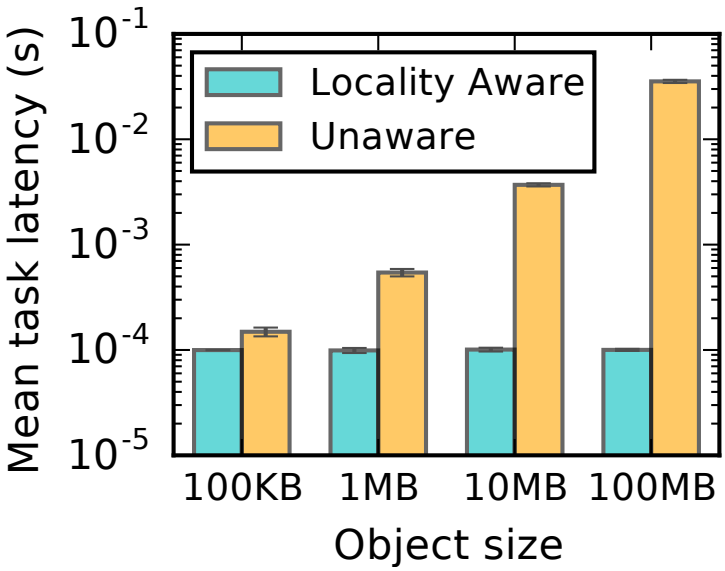


Figure 3.8: Tasks leverage locality-aware placement. 1000 tasks with a random object dependency are scheduled onto one of two nodes. With locality-aware policy, task latency remains independent of the size of task inputs instead of growing by 1-2 orders of magnitude.

2. What overheads are imposed on distributed primitives (e.g., allreduce) written using Ray's API? (Section 3.5)
3. In the context of RL workloads, how does Ray compare against specialized systems for training, serving, and simulation? (Section 3.5)
4. What advantages does Ray provide for RL applications, compared to custom systems? (Section 3.5)

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 3.8, 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 3.9, we evaluate this ability on an embarrassingly parallel

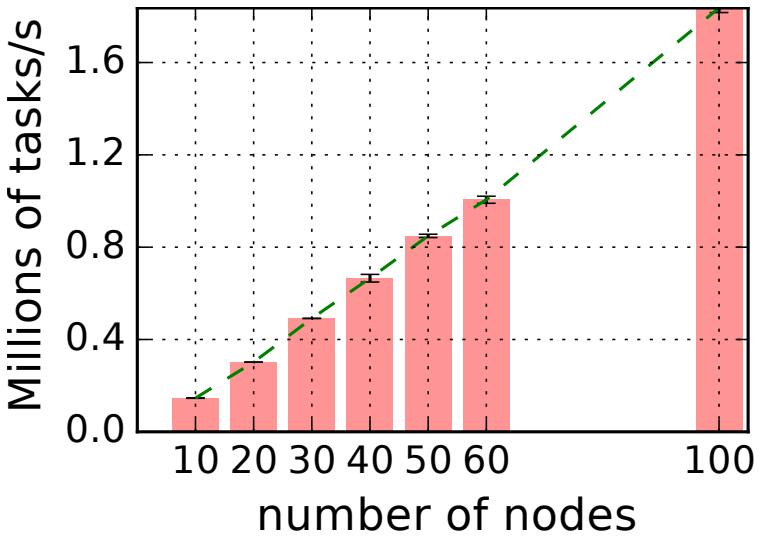


Figure 3.9: Near-linear scalability leveraging the GCS and bottom-up distributed scheduler. Ray reaches 1 million tasks per second throughput with 60 nodes. $x \in \{70, 80, 90\}$ omitted due to cost.

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.

Object store performance. To evaluate the performance of the object store (Section 3.4), we track two metrics: IOPS (for small objects) and write throughput (for large objects). In Figure 3.10, 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 [125] layer on top of Redis. Figure 3.11a 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.

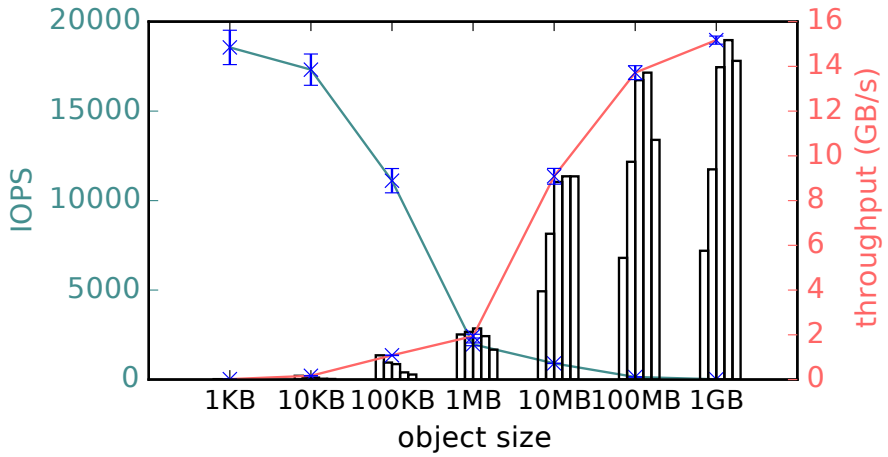


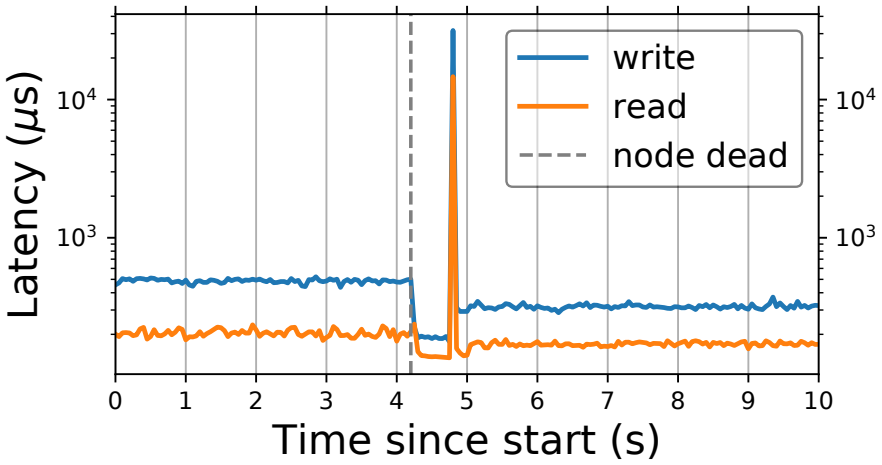
Figure 3.10: Object store write throughput and IOPS. From a single client, throughput exceeds 15GB/s (red) for large objects and 18K IOPS (cyan) for small objects on a 16 core instance (m4.4xlarge). It uses 8 threads to copy objects larger than 0.5MB and 1 thread for small objects. Bar plots report throughput with 1, 2, 4, 8, 16 threads. Results are averaged over 5 runs.

In Figure 3.11b 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.

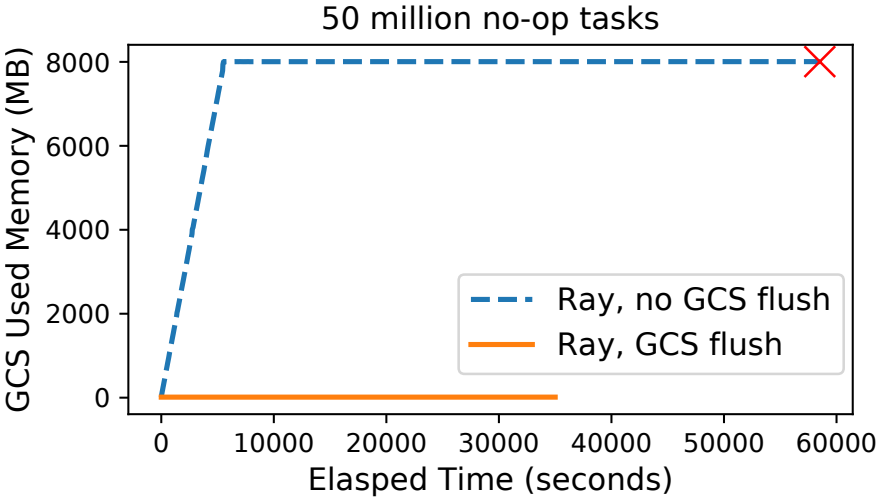
Recovering from task failures. In Figure 3.12a, we demonstrate Ray's ability to transparently recover from worker node failures and elastically scale, using the durable GCS lineage storage. The workload, run on m4.xlarge instances, consists of linear chains of 100ms tasks submitted by the driver. As nodes are removed (at 25s, 50s, 100s), the local schedulers 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 3.12a to provide transparent fault tolerance for *stateful computation*. Ray additionally leverages user-defined checkpoint functions to bound the reconstruction time for actors (Figure 3.12b). 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 do not mutate state.

Allreduce. Allreduce is a distributed communication primitive important to many ma-



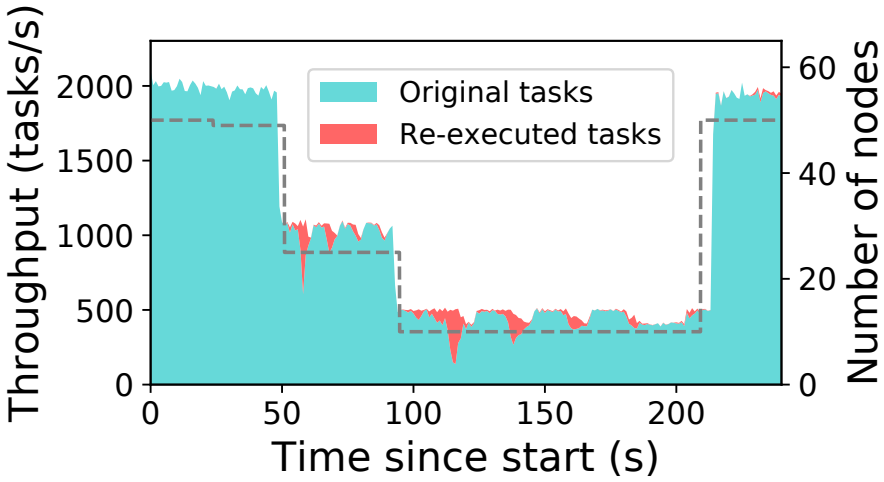
(a) A timeline for GCS read and write latencies as viewed from a client submitting tasks. The chain starts with 2 replicas. We manually trigger reconfiguration as follows. At $t \approx 4.2$s, a chain member is killed; immediately after, a new chain member joins, initiates state transfer, and restores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



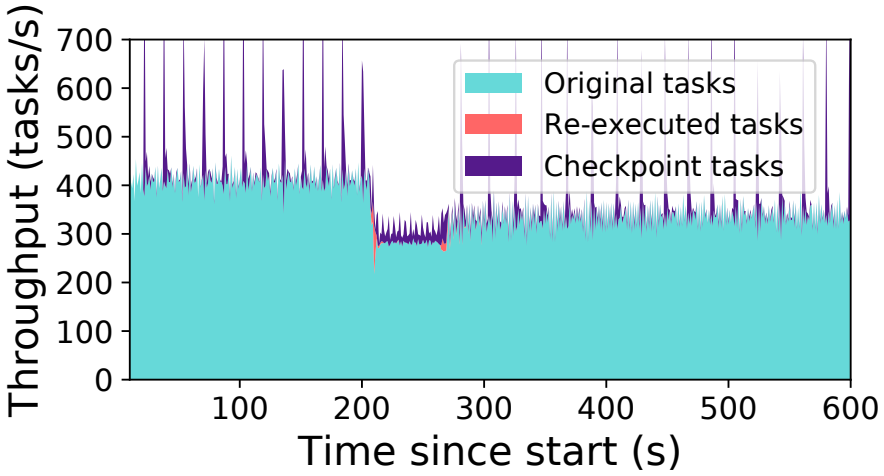
(b) The Ray GCS maintains a constant memory footprint with GCS flushing. Without GCS flushing, the memory footprint reaches a maximum capacity and the workload fails to complete within a predetermined duration (indicated by the red cross).

Figure 3.11: Ray GCS fault tolerance and flushing.

chine learning workloads. Here, we evaluate whether Ray can natively support a ring allreduce [141] implementation with low enough overhead to match existing implementations [134]. 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 3.13a). We attribute Ray's performance



(a) Task reconstruction

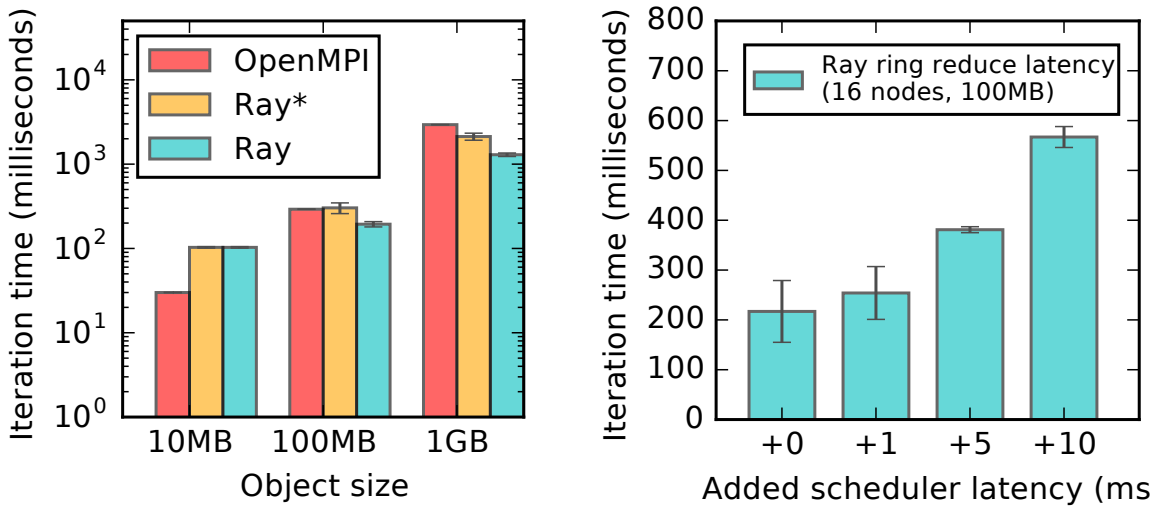


(b) Actor reconstruction

Figure 3.12: 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).

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 [58]. 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 3.13b, we inject artificial task execution delays and show that performance drops



(a) Ray vs OpenMPI

(b) Ray scheduler ablation

Figure 3.13: (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.

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 [146, 106], making such 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 [135]) 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 (3.5) or parameter server, both implemented on top of the Ray API.

In Figure 3.14, we evaluate the performance of the Ray (synchronous) parameter-server SGD implementation against state-of-the-art implementations [134], 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

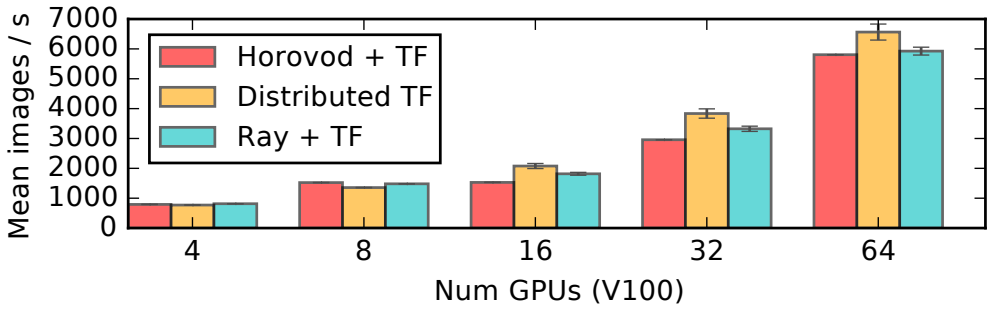


Figure 3.14: Images per second reached when distributing the training of a ResNet-101 TensorFlow model (from the official TF benchmark). All experiments were run on p3.16xl instances connected by 25Gbps Ethernet, and workers allocated 4 GPUs per node as done in Horovod [134]. We note some measurement 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.

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 3.14 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 [35] focus on serving predictions to external clients.

In this setting, low latency is critical for achieving high utilization. To show this, in Table 3.3 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.

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 3.3: Throughput comparisons for Clipper [35], 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.

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 3.4: Timesteps per second for the Pendulum-v0 simulator in OpenAI Gym [25]. Ray allows for better utilization when running heterogeneous simulations at scale.

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 heterogeneity 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 3.4 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.

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

[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.

require substantial engineering effort to port to custom-built systems, but are transparently supported by Ray's dynamic task graph execution engine.

Evolution Strategies

To evaluate Ray on large-scale RL workloads, we implement the evolution strategies (ES) algorithm and compare to the reference implementation [129]—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).

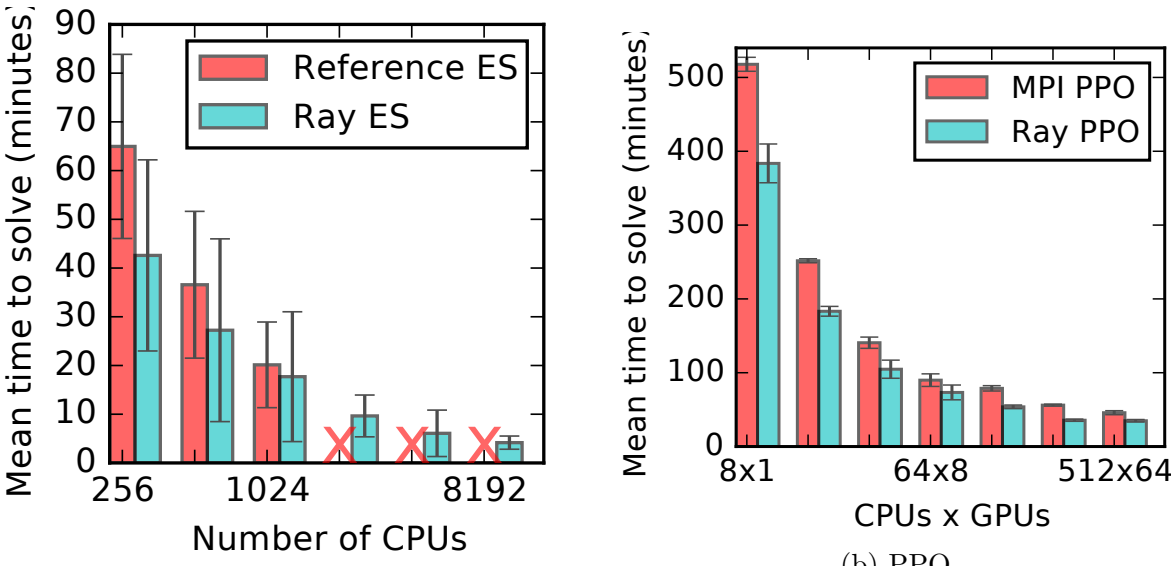
As shown in Figure 3.15a, an implementation on Ray scales to 8192 cores. Doubling the cores available yields an average completion time speedup of 1.6×. Conversely, the special-purpose system fails to complete at 2048 cores, where the work in the system exceeds the processing capacity of the application 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) [131] in Ray and compare to a highly-optimized reference implementation [116] 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 3.15b, 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, [116] includes two custom implementations of PPO, one using MPI for large clusters and one that is optimized for



(a) Evolution Strategies

(b) PPO

Figure 3.15: Time to reach a score of 6000 in the Humanoid-v1 task [25]. **(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 [116] 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).

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 [49], 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×.*

3.6 Related Work

Dynamic task graphs. Ray is closely related to CIEL [107] and Dask [127]. 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 3.5) submits 32 rounds of 16 tasks in 200ms. Meanwhile, Dask reports a maximum scheduler throughput of 3k tasks/s on 512 cores [126]. 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 3.13b). 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 [40], Spark [155], and Dryad [77] 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 [108] is a dataflow system that provides improved scalability for some workloads, but only supports static task graphs.

Machine learning frameworks. TensorFlow [1] and MXNet [32] 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 [98] 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 [58] 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 [28] and Akka [3] 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 [8] and C++ Actor Framework [31], 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) [30], distributed file systems (e.g., GFS [60]), resource management (e.g., Omega [132]), and distributed frameworks (e.g., MapReduce [40], BOOM [4]), 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 [40, 156, 107] use a centralized scheduler architecture. While Sparrow [118] 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 [72] implements a two-level hierarchical scheduler, but its top-level scheduler manages frameworks, not individual tasks. Canary [124] achieves impressive performance by having each scheduler instance handle a portion of the task graph, but does not handle dynamic computation graphs.

Cilk [23] 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.

3.7 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.

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 [104], PPO [131], DQN [103], ES [129], DDPG [136], and Ape-X [76]. 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 4.4, 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.

3.8 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 4

Case Study: Distributed Training

Distributed computation is increasingly important for deep learning, and many deep learning frameworks provide built-in support for distributed training. This results in a tight coupling between the neural network computation and the underlying distributed execution, which poses a challenge for the implementation of new communication and aggregation strategies. We argue that *decoupling the deep learning framework from the distributed execution framework* enables the flexible development of new communication and aggregation strategies. Furthermore, we argue that the Ray architecture described in Chapter 3 provides a flexible set of distributed computing primitives that, when used in conjunction with modern deep learning libraries, enable the implementation of a wide range of gradient aggregation strategies appropriate for different computing environments. We show how these primitives can be used to address common problems, and demonstrate the performance benefits empirically.[1]

4.1 Introduction

Given the importance of distributed computation in scaling up deep learning training, many of today's deep learning frameworks provide built-in support for distributed training [1, 32]. However, as a result, the training algorithms are often tightly coupled to the underlying distributed infrastructure. The resulting communication primitives are often difficult to modify and customize at the application level (without modifying the deep learning framework itself).

As practitioners seek to make distributed training practical in increasingly varied environments such as public clouds where individual machines may be preempted or may fail or where networks exhibit variable performance between machines, we will need training algorithms capable of adapting to sporadic failures and slow machines. At the same time, our algorithms should be able to take advantage of highly reliable computing environments such as supercomputers when such environments are available.

[1] Material in this chapter is based adapted from [26].

One of the most important components of data-parallel training, is the ability to rapidly aggregate gradients that have been computed on different machines and devices (e.g., GPUs). Aggregation is often performed by summation, and the aggregation techniques include allreduce algorithms [61] or parameter server approaches [93]. Within these approaches, there are many variants designed to address different bottlenecks in practice. In an idealized setting (e.g., a supercomputer), straightforward synchronous stochastic gradient descent (SGD) works well and has been used very effectively [66]. In a setting with slow machines, stragglers may become a problem, and techniques like backup workers [119], asynchronous SGD [39] or the stale synchronous parameter server [73] can be used to address this bottleneck.

A more limited set of techniques have been proposed to address the problem of slow parameter servers [154].

We will show that the primitives provided by Ray, though not specifically designed for gradient aggregation, can be used to implement all of these different schemes.

In addition, separating the distributed execution layer from the deep learning framework allows Ray-based implementations to swap in different deep learning frameworks within the same application and leaves open the option of implementing different workers using different deep learning libraries.

4.2 Ray Primitives

Ray (described in Chapter 3) is a high-performance distributed execution framework targeted at supporting AI applications and machine learning in dynamic environments [114]. The underlying system is capable of executing tasks with millisecond latencies at throughputs of millions of tasks per second. Ray also uses a shared-memory object store in addition to zero-copy serialization through Apache Arrow [7] to provide efficient handling of numerical data. Ray's API is designed for general purpose distributed computing, which is precisely why it provides the flexibility needed to implement diverse training strategies.

Several components of Ray's API make it well-suited for implementing the communication strategies underlying distributed training. First, Ray achieves parallelism through *fine-grained dynamic tasks*. A task may consist of a single gradient computation or a full training run. As a result, one task (e.g., a training task) may spawn many more tasks as it executes (e.g., gradient computation tasks). Parallelism is achieved by executing multiple tasks at the same time on different workers or actors. Second, Ray encapsulates stateful computation with *actors*. An actor is a stateful service whose method invocations are executed as tasks on the actor. These tasks may trigger the submission of additional tasks or may depend on other tasks in complex ways.

A parameter server is a natural example of a Ray actor. It may expose a method for *getting* its parameters and a method for *updating* its parameters. These methods could be invoked by any number of parameter server clients, which themselves could be implemented as actors or as long-running non-actor tasks.

At a programming level, Ray tasks (including actor method invocations) return *object IDs* (similar to futures). An application can choose to fetch the values corresponding to a given set of object IDs by blocking until the corresponding tasks have completed. Crucially, Ray includes a primitive *wait* which allows applications to wait for a subset of tasks to complete or for a timeout to expire. *This primitive gives applications great flexibility in determining their execution and control flow as a function of runtime performance characteristics*, and it is critical for the implementation of strategies like backup workers for synchronous SGD [119] or partial pulling [154].

By separating the neural network graph from the communication strategy, Ray makes it easy to experiment with a wide range of gradient aggregation strategies without changing the underlying neural network computation or modifying the deep learning framework.

4.3 Examples

To illustrate the diversity of training strategies that can be implemented using Ray's primitives, we implement the following examples. Each of these is around one to ten extra lines of code on top of the basic synchronous and asynchronous sharded parameter server training applications, which themselves are around a hundred lines of code.[2]

Vanilla Synchronous Parameter Server: In this basic scheme [93], the neural network weights are divided evenly between a number of parameter servers. Each parameter server occupies a different machine. A number of workers processes occupy a different set of machines and do the following in lockstep: retrieve the latest parameters from all of the parameter servers, compute a gradient update using some training data, and push the different portions of the gradient update to the relevant parameter servers. Synchronous schemes are often preferred to asynchronous ones because they offer more predictable training behavior and are less dependent on hardware or runtime characteristics.

Synchronous Parameter Server with Backup Workers: This scheme [119] is similar to the regular synchronous scheme except that a few extra workers are kept around. If a worker is slow and falls behind, it's gradient update is not used and results from a backup worker are used instead.

Asynchronous Parameter Server: The asynchronous scheme [39] is the same as the synchronous scheme except that the workers no longer operate in lockstep. If a worker is slow, that worker may fall behind and updates from that worker may be received even after the parameter server has performed a large number of updates, but other workers are not blocked from making progress by a slow worker.

Bounded Staleness: The bounded staleness scheme [73, 12], similar to the stale-synchronous parallel scheme, deals with slow workers by allowing workers to proceed asynchronously within a certain bound. If a worker falls too far behind, then either the faster workers will wait for it, or its updates will simply not be used.

[2]Several code examples are available at `https://github.com/ray-project/ray/tree/master/examples/parameter_server`.

Partial Pulling: The partial pulling scheme [154] is an approach for dealing with slow parameter servers as opposed to slow workers. It allows workers to proceed with a given gradient computation without waiting to receive parameters from every parameter server. If too much time has passed and parameters have not arrived from a given parameter server, the worker will simply reuse the previous parameter values from that parameter server.

Implementing these communication strategies within a deep learning framework such as TensorFlow would require deep integration within the framework itself. By providing distributed computing primitives outside of a deep learning framework, Ray enables these custom communication strategies to be implemented easily at the application level.

4.4 Experiments

As a proof of concept, we implemented the five aggregation strategies from Section 4.3 and integrated them with the TensorFlow CIFAR-10 Resnet implementation provided as part of the official distributed TensorFlow example [139].

In Figure 4.1, we compare the synchronous parameter server throughput of our Ray plus TensorFlow implementation to the throughput of the pure TensorFlow version. The performance results are largely similar despite the lack of tuning of our implementation. The results are slightly worse in the case of 8 workers, for reasons which we are still investigating. In terms of complexity, implementing this approach and the four other aggregation strategies from Section 4.3 took a couple days, which included the time required to integrate with TensorFlow. In contrast, the synchronous parameter server implementation in TensorFlow took months of engineering time.

Our distributed training experiments used between two and twelve g3.4xlarge worker instances on Amazon Web Services. Each worker and parameter server ran on a dedicated instance and a separate instance was used to host a TensorBoard visualization job. We used the nexus-scheduler framework for orchestrating training runs and tracking results, which we plan to open-source soon.

In Section 4.5, we implement a partial-pull aggregation scheme and demonstrate that the Ray implementation is robust to slowdowns in individual parameter server shards. This experiment is of interest because none of the existing deep learning frameworks are robust to slowdowns of individual parameter server shards.

4.5 Parameter Server Slowdowns

In the experiment below we evaluate the sensitivity of distributed training to periodic parameter server slowdowns. This example was of interest because none of the existing deep learning frameworks are robust to slowdowns of individual parameter server shards.

For the baseline experiment, we launched two gradient workers and 2 parameter servers using TensorFlow's asynchronous parameter server from the official CIFAR-10 Estimator

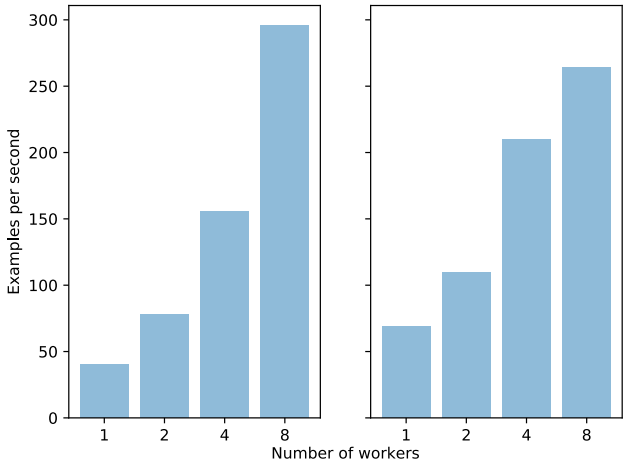


Figure 4.1: Left: Synchronous parameter server throughput for the pure TensorFlow implementation. Right: throughput of the Ray plus TensorFlow implementation. In both cases, the number of parameter servers is half the number of workers (rounded up).

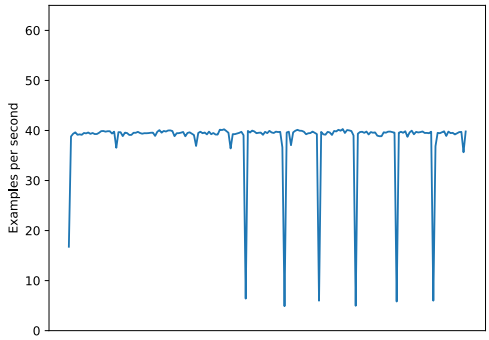


Figure 4.2: Training throughput of the pure TensorFlow asynchronous parameter server implementation in the presence of periodic slowdowns of a single parameter server shard. Throughput consistently decreases when one of the parameter servers slows down.

implementation and inserted a 10 second pauses in one of the parameter server shards roughly every 60 seconds.

TensorFlow training predictably paused whenever any of the parameter server shards paused. The Ray implementation used a partial-pull aggregation strategy [154] which allowed training to continue during pauses at the cost of increased staleness for some of the parameters. The results can be seen in Figure 4.2 and Figure 4.3.

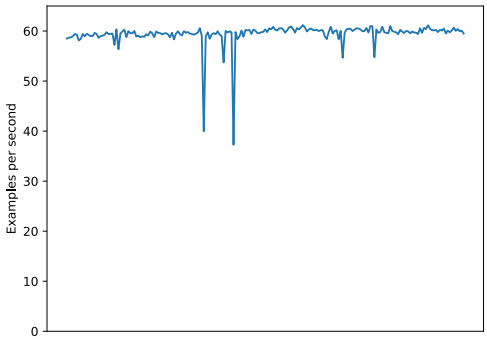


Figure 4.3: Training throughput of a Ray-based implementation of partial pulling in the presence of periodic slowdowns of a single parameter server shard. This particular aggregation strategy allows workers to avoid waiting for slow parameter servers and hence throughput suffers very little compared with the pure TensorFlow implementation.

4.6 Conclusion

Model training in the form of stochastic gradient descent is at the heart of many machine learning applications, and algorithm speed is a primary concern for many practitioners and researchers. Some important computational patterns have been established, such as the need for data parallelism in which a single neural network architecture is replicated on multiple devices and machines to compute gradients with large batches quickly. Well-established patterns also include the need to shard the parameters of a single large matrix such as an embedding matrix across multiple machines due to the sheer size of the matrix. Supporting these use cases well is critical. In addition, there are patterns of distributed training that are very much in flux. For example, the general form of model parallelism in which different layers of a single model are split across multiple devices or in which the different models in an ensemble exist on different machines. In these cases, the flexibility needed by different applications and different approaches requires the underlying distributed computing framework to allow a great deal of generality in the patterns that it supports. We showed that the architecture proposed in Chapter 3 allows for the flexible expression of many different patterns of distributed training in a performant manner.

Chapter 5

Case Study: Distributed Optimization with ADMM

In this chapter, we consider the alternating direction method of multipliers (ADMM) algorithm for solving an important class of convex optimization problems. One of the primary advantages of ADMM over comparable algorithms is that, in many settings, the algorithm updates can be made massively parallel. Therefore, ADMM is particularly well suited for machine learning problems with large numbers of features or large datasets and can leverage a distributed architecture like the one proposed in Chapter 3.

We provide a new proof of the linear convergence of ADMM when one of the objective terms is strongly convex. Our proof is based on a framework for analyzing optimization algorithms introduced in [91], reducing algorithm convergence to verifying the stability of a dynamical system. This approach generalizes a number of existing results and obviates any assumptions about specific choices of algorithm parameters. On a numerical example, we demonstrate that minimizing the derived bound on the convergence rate provides a practical approach to selecting algorithm parameters for particular ADMM instances. We complement our upper bound by constructing a nearly-matching lower bound on the worst-case rate of convergence.[1]

5.1 Introduction

The alternating direction method of multipliers (ADMM) seeks to solve the problem

$$\begin{array}{ll} \text{minimize} & f(x) + g(z) \\ \text{subject to} & Ax + Bz = c, \end{array} \tag{5.1}$$

with variables $x \in \mathbb{R}^p$ and $z \in \mathbb{R}^q$ and constants $A \in \mathbb{R}^{r \times p}$, $B \in \mathbb{R}^{r \times q}$, and $c \in \mathbb{R}^r$. ADMM was introduced in [63] and [57]. More recently, it has found applications in a variety of

[1] Material in this chapter is based adapted from [113].

distributed settings such as model fitting, resource allocation, and classification. A partial list of examples includes [21, 149, 22, 55, 133, 92, 150, 157, 102, 151, 9, 56, 128, 20, 158]. See [24] for an overview.

Part of the appeal of ADMM is the fact that, in many settings, the algorithm updates are massively parallel and lend themselves to distributed settings. The algorithm itself is given in Algorithm 1. We refer to $\rho > 0$ as the step-size parameter.

Algorithm 1 Alternating Direction Method of Multipliers

Input: functions f and g, matrices A and B, vector c, parameter ρ
Initialize x_0, z_0, u_0
repeat
  $x_{k+1} = \arg\min_x f(x) + \frac{\rho}{2}\|Ax + Bz_k - c + u_k\|^2$
  $z_{k+1} = \arg\min_z g(z) + \frac{\rho}{2}\|Ax_{k+1} + Bz - c + u_k\|^2$
  $u_{k+1} = u_k + Ax_{k+1} + Bz_{k+1} - c$.
until meet stopping criterion

A popular variant of Algorithm 1 is over-relaxed ADMM, which introduces an additional parameter α and replaces each instance of Ax_{k+1} in the z and u updates in Algorithm 1 with

$$\alpha Ax_{k+1} - (1-\alpha)(Bz_k - c).$$

The parameter α is typically chosen to lie in the interval $(0, 2]$, but we demonstrate in Section 5.8 that a larger set of choices can lead to convergence. Over-relaxed ADMM is described in Algorithm 2. Note that when $\alpha = 1$, Algorithm 2 and Algorithm 1 coincide. We will analyze Algorithm 2.

Algorithm 2 Over-Relaxed Alternating Direction Method of Multipliers

Input: functions f and g, matrices A and B, vector c, parameters ρ and α
Initialize x_0, z_0, u_0
repeat
  $x_{k+1} = \arg\min_x f(x) + \frac{\rho}{2}\|Ax + Bz_k - c + u_k\|^2$
  $z_{k+1} = \arg\min_z g(z) + \frac{\rho}{2}\|\alpha Ax_{k+1} - (1-\alpha)Bz_k + Bz - \alpha c + u_k\|^2$
  $u_{k+1} = u_k + \alpha Ax_{k+1} - (1-\alpha)Bz_k + Bz_{k+1} - \alpha c$
until meet stopping criterion

The conventional wisdom that ADMM works well without any tuning [24], for instance by setting $\rho = 1$, is often not borne out in practice. Algorithm 1 can be challenging to tune, and Algorithm 2 is even harder. We use the machinery developed in this chapter to make reasonable recommendations for setting ρ and α when some information about f is available (Section 5.8).

In this chapter, we give an upper bound on the linear rate of convergence of Algorithm 2 for all ρ and α (Theorem 7), and we give a nearly-matching lower bound (Theorem 8).

Importantly, we show that we can prove convergence rates for Algorithm 2 by numerically solving a 4×4 semidefinite program (Theorem 6). When we change the parameters of Algorithm 2, the semidefinite program changes. Whereas prior work requires a new proof of convergence for every change to the algorithm, our work automates that process.

Our work builds on the integral quadratic constraint framework introduced in [91], which uses ideas from robust control to analyze optimization algorithms that can be cast as discrete-time linear dynamical systems. Our work provides a flexible framework for analyzing variants of Algorithm 1, including those like Algorithm 2 created by the introduction of additional parameters. In Section 5.7, we compare our results to prior work.

We note that aside from assuming basic facts about matrices and convex functions, our presentation is completely self-contained.

5.2 Preliminaries and Notation

Let $\overline{\mathbb{R}}$ denote the extended real numbers $\mathbb{R} \cup \{+\infty\}$. Suppose that $f \colon \mathbb{R}^d \to \overline{\mathbb{R}}$ is convex and differentiable, and let ∇f denote the gradient of f. We say that f is strongly convex with parameter $m > 0$ if for all $x, y \in \mathbb{R}^d$, we have

$$f(x) \geq f(y) + \nabla f(y)^\top (x - y) + \tfrac{m}{2}\|x - y\|^2.$$

When ∇f is Lipschitz continuous with parameter L, it follows that

$$f(x) \leq f(y) + \nabla f(y)^\top (x - y) + \tfrac{L}{2}\|x - y\|^2.$$

For $0 < m \leq L < \infty$, let $S_d(m, L)$ denote the set of differentiable convex functions $f \colon \mathbb{R}^d \to \overline{\mathbb{R}}$ that are strongly convex with parameter m and whose gradients are Lipschitz continuous with parameter L. We let $S_d(0, \infty)$ denote the set of convex functions $\mathbb{R}^d \to \overline{\mathbb{R}}$. In general, we let ∂f denote the subdifferential of f. We denote the d-dimensional identity matrix by I_d and the d-dimensional zero matrix by 0_d. We will make use of the following results.

Lemma 1. *Suppose that $f \in S_d(m, L)$, where $0 < m \leq L < \infty$. Suppose that $b_1 = \nabla f(a_1)$ and $b_2 = \nabla f(a_2)$. Then*

$$\begin{bmatrix} a_1 - a_2 \\ b_1 - b_2 \end{bmatrix}^\top \begin{bmatrix} -2mLI_d & (m+L)I_d \\ (m+L)I_d & -2I_d \end{bmatrix} \begin{bmatrix} a_1 - a_2 \\ b_1 - b_2 \end{bmatrix} \geq 0.$$

Proof. The Lipschitz continuity of ∇f implies the co-coercivity of ∇f, that is

$$(a_1 - a_2)^\top (b_1 - b_2) \geq \tfrac{1}{L}\|b_1 - b_2\|^2.$$

Note that $f(x) - \frac{m}{2}\|x\|^2$ is convex and its gradient is Lipschitz continuous with parameter $L - m$. Applying the co-coercivity condition to this function and rearranging gives

$$(m + L)(a_1 - a_2)^\top (b_1 - b_2) \geq mL\|a_1 - a_2\|^2 + \|b_1 - b_2\|^2,$$

which can be rearranged in matrix form to complete the proof. □

Lemma 2. *Suppose that $f \in S_d(0,\infty)$, and suppose that $b_1 \in \partial f(a_1)$ and $b_2 \in \partial f(a_2)$. Then*

$$\begin{bmatrix} a_1 - a_2 \\ b_1 - b_2 \end{bmatrix}^\top \begin{bmatrix} 0_d & I_d \\ I_d & 0_d \end{bmatrix} \begin{bmatrix} a_1 - a_2 \\ b_1 - b_2 \end{bmatrix} \geq 0.$$

Lemma 2 is simply the statement that the subdifferential of a convex function is a monotone operator.

When M is a matrix, we use κ_M to denote the condition number of M. For example, $\kappa_A = \sigma_1(A)/\sigma_p(A)$, where $\sigma_1(A)$ and $\sigma_p(A)$ denote the largest and smallest singular values of the matrix A. When $f \in S_d(m, L)$, we let $\kappa_f = \frac{L}{m}$ denote the condition number of the function f. We let $M \otimes N$ denote the Kronecker product of matrices M and N.

5.3 ADMM as a Dynamical System

We group our assumptions together in Assumption 3.

Assumption 3. *We assume that f and g are convex, closed, and proper. We assume that for some $0 < m \leq L < \infty$, we have $f \in S_p(m, L)$ and $g \in S_q(0,\infty)$. We assume that A is invertible and that B has full column rank.*

The assumption that f and g are closed (their sublevel sets are closed) and proper (they neither take on the value $-\infty$ nor are they uniformly equal to $+\infty$) is standard. Similar rank assumptions on A and B are standard as well [41, 75, 62].

We begin by casting over-relaxed ADMM as a discrete-time dynamical system with state sequence (ξ_k), input sequence (ν_k), and output sequences (w_k^1) and (w_k^2) satisfying the recursions

$$\xi_{k+1} = (\hat{A} \otimes I_r)\xi_k + (\hat{B} \otimes I_r)\nu_k \tag{5.2a}$$
$$w_k^1 = (\hat{C}^1 \otimes I_r)\xi_k + (\hat{D}^1 \otimes I_r)\nu_k \tag{5.2b}$$
$$w_k^2 = (\hat{C}^2 \otimes I_r)\xi_k + (\hat{D}^2 \otimes I_r)\nu_k \tag{5.2c}$$

for particular matrices $\hat{A}$, $\hat{B}$, $\hat{C}^1$, $\hat{D}^1$, $\hat{C}^2$, and $\hat{D}^2$ (whose dimensions do not depend on any problem parameters).

First define the functions $\hat{f}, \hat{g} \colon \mathbb{R}^r \to \overline{\mathbb{R}}$ via

$$\begin{aligned} \hat{f} &= (\rho^{-1} f) \circ A^{-1} \\ \hat{g} &= (\rho^{-1} g) \circ B^\dagger + \mathbb{I}_{\operatorname{im} B}, \end{aligned} \tag{5.3}$$

where $B^\dagger$ is any left inverse of B and where $\mathbb{I}_{\operatorname{im} B}$ is the indicator function of the image of B. For future reference, we define $\kappa = \kappa_f \kappa_A^2$ and to normalize, we define

$$\hat{m} = \frac{m}{\sigma_1^2(A)} \quad \hat{L} = \frac{L}{\sigma_p^2(A)} \quad \rho = (\hat{m}\hat{L})^{\frac{1}{2}}\rho_0. \tag{5.4}$$

Note that under Assumption 3,

$$\hat{f} \in S_p(\rho_0^{-1}\kappa^{-\frac{1}{2}}, \rho_0^{-1}\kappa^{\frac{1}{2}}) \tag{5.5a}$$
$$\hat{g} \in S_q(0, \infty). \tag{5.5b}$$

To define the relevant sequences, let the sequences (x_k), (z_k), and (u_k) be generated by Algorithm 2 with parameters α and ρ. Define the sequences (r_k) and (s_k) by $r_k = Ax_k$ and $s_k = Bz_k$, and define the sequence (ξ_k) by

$$\xi_k = \begin{bmatrix} s_k \\ u_k \end{bmatrix}.$$

We define the sequence (ν_k) as in Proposition 4.

Proposition 4. *There exist sequences (β_k) and (γ_k) with $\beta_k = \nabla \hat{f}(r_k)$ and $\gamma_k \in \partial \hat{g}(s_k)$ such that when we define the sequence (ν_k) by*

$$\nu_k = \begin{bmatrix} \beta_{k+1} \\ \gamma_{k+1} \end{bmatrix},$$

then the sequences (ξ_k) and (ν_k) satisfy Equation 5.2a with the matrices

$$\hat{A} = \begin{bmatrix} 1 & \alpha - 1 \\ 0 & 0 \end{bmatrix} \qquad \hat{B} = \begin{bmatrix} \alpha & -1 \\ 0 & -1 \end{bmatrix}. \tag{5.6}$$

Proof. Using the fact that A has full rank, we rewrite the update rule for x from Algorithm 2 as

$$x_{k+1} = A^{-1} \arg\min_r f(A^{-1}r) + \tfrac{\rho}{2}\|r + s_k - c + u_k\|^2.$$

Multiplying through by A, we can write

$$r_{k+1} = \arg\min_r \hat{f}(r) + \tfrac{1}{2}\|r + s_k - c + u_k\|^2.$$

This implies that

$$0 = \nabla \hat{f}(r_{k+1}) + r_{k+1} + s_k - c + u_k,$$

and so

$$r_{k+1} = -s_k - u_k + c - \beta_{k+1}, \tag{5.7}$$

where $\beta_{k+1} = \nabla \hat{f}(r_{k+1})$. In the same spirit, we rewrite the update rule for z as

$$s_{k+1} = \arg\min_s \hat{g}(s) + \tfrac{1}{2}\|\alpha r_{k+1} - (1-\alpha)s_k + s - \alpha c + u_k\|^2.$$

It follows that there exists some $\gamma_{k+1} \in \partial \hat{g}(s_{k+1})$ such that

$$0 = \gamma_{k+1} + \alpha r_{k+1} - (1-\alpha)s_k + s_{k+1} - \alpha c + u_k.$$

It follows then that

$$\begin{aligned} s_{k+1} &= -\alpha r_{k+1} + (1-\alpha)s_k + \alpha c - u_k - \gamma_{k+1} \\ &= s_k - (1-\alpha)u_k + \alpha\beta_{k+1} - \gamma_{k+1}, \end{aligned} \tag{5.8}$$

where the second equality follows by substituting in Equation 5.7. Combining Equation 5.7 and Equation 5.8 to simplify the u update, we have

$$\begin{aligned} u_{k+1} &= u_k + \alpha r_{k+1} - (1-\alpha)s_k + s_{k+1} - \alpha c \\ &= -\gamma_{k+1}. \end{aligned} \tag{5.9}$$

Together, Equation 5.8 and Equation 5.9 confirm the relation in Equation 5.2a. □

Corollary 5. *Define the sequences (β_k) and (γ_k) as in Proposition 4. Define the sequences (w_k^1) and (w_k^2) via*

$$w_k^1 = \begin{bmatrix} r_{k+1} - c \\ \beta_{k+1} \end{bmatrix} \qquad w_k^2 = \begin{bmatrix} s_{k+1} \\ \gamma_{k+1} \end{bmatrix}.$$

Then the sequences (ξ_k), (ν_k), (w_k^1), and (w_k^2) satisfy Equation 5.2b and Equation 5.2c with the matrices

$$\hat{C}^1 = \begin{bmatrix} -1 & -1 \\ 0 & 0 \end{bmatrix} \quad \hat{D}^1 = \begin{bmatrix} -1 & 0 \\ 1 & 0 \end{bmatrix} \quad \hat{C}^2 = \begin{bmatrix} 1 & \alpha - 1 \\ 0 & 0 \end{bmatrix} \quad \hat{D}^2 = \begin{bmatrix} \alpha & -1 \\ 0 & 1 \end{bmatrix}. \tag{5.10}$$

5.4 Convergence Rates from Semidefinite Programming

Now, in Theorem 6, we make use of the perspective developed in Section 5.3 to obtain convergence rates for Algorithm 2. This is essentially the same as the main result of [91], and we include it because it is simple and self-contained.

Theorem 6. *Suppose that Assumption 3 holds. Let the sequences (x_k), (z_k), and (u_k) be generated by running Algorithm 2 with step size $\rho = (\hat{m}\hat{L})^{\frac{1}{2}}\rho_0$ and with over-relaxation parameter α. Suppose that (x_*, z_*, u_*) is a fixed point of Algorithm 2, and define*

$$\varphi_k = \begin{bmatrix} z_k \\ u_k \end{bmatrix} \qquad \varphi_* = \begin{bmatrix} z_* \\ u_* \end{bmatrix}.$$

Fix $0 < \tau < 1$, and suppose that there exist a 2×2 positive definite matrix $P \succ 0$ and nonnegative constants $\lambda^1, \lambda^2 \geq 0$ such that the 4×4 linear matrix inequality

$$0 \preceq \begin{bmatrix} \hat{A}^\top P \hat{A} - \tau^2 P & \hat{A}^\top P \hat{B} \\ \hat{B}^\top P \hat{A} & \hat{B}^\top P \hat{B} \end{bmatrix} + \begin{bmatrix} \hat{C}^1 & \hat{D}^1 \\ \hat{C}^2 & \hat{D}^2 \end{bmatrix}^\top \begin{bmatrix} \lambda^1 M^1 & 0 \\ 0 & \lambda^2 M^2 \end{bmatrix} \begin{bmatrix} \hat{C}^1 & \hat{D}^1 \\ \hat{C}^2 & \hat{D}^2 \end{bmatrix} \tag{5.11}$$

is satisfied, where $\hat{A}$ and $\hat{B}$ are defined in Equation 5.6, where $\hat{C}^1$, $\hat{D}^1$, $\hat{C}^2$, and $\hat{D}^2$ are defined in Equation 5.10, and where M^1 and M^2 are given by

$$M^1 = \begin{bmatrix} -2\rho_0^{-2} & \rho_0^{-1}(\kappa^{-\frac{1}{2}} + \kappa^{\frac{1}{2}}) \\ \rho_0^{-1}(\kappa^{-\frac{1}{2}} + \kappa^{\frac{1}{2}}) & -2 \end{bmatrix} \qquad M^2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Then for all $k \geq 0$, we have

$$\|\varphi_k - \varphi_*\| \leq \kappa_B \sqrt{\kappa_P} \|\varphi_0 - \varphi_*\| \tau^k.$$

Proof. Define r_k, s_k, β_k, γ_k, ξ_k, ν_k, w_k^1, and w_k^2 as before. Choose $r_* = Ax_*$, $s_* = Bz_*$, and

$$w_*^1 = \begin{bmatrix} r_* - c \\ \beta_* \end{bmatrix} \quad w_*^2 = \begin{bmatrix} s_* \\ \gamma_* \end{bmatrix} \quad \xi_* = \begin{bmatrix} s_* \\ u_* \end{bmatrix} \quad \nu_* = \begin{bmatrix} \beta_* \\ \gamma_* \end{bmatrix}$$

such that $(\xi_*, \nu_*, w_*^1, w_*^2)$ is a fixed point of the dynamics of Equation 5.2 and satisfying $\beta_* = \nabla \hat{f}(r_*)$, $\gamma_* \in \partial \hat{g}(s_*)$. Now, consider the Kronecker product of the right hand side of Equation 5.11 and I_r. Multiplying this on the left and on the right by $\begin{bmatrix} (\xi_j - \xi_*)^\top & (\nu_j - \nu_*)^\top \end{bmatrix}$ and its transpose, respectively, we find

$$\begin{aligned} 0 \geq\ & (\xi_{j+1} - \xi_*)^\top P (\xi_{j+1} - \xi_*) \\ & - \tau^2 (\xi_j - \xi_*)^\top P (\xi_j - \xi_*) \\ & + \lambda^1 (w_j^1 - w_*^1)^\top M^1 (w_j^1 - w_*^1) \\ & + \lambda^2 (w_j^2 - w_*^2)^\top M^2 (w_j^2 - w_*^2). \end{aligned} \tag{5.12}$$

Lemma 1 and Equation 5.5a show that the third term on the right hand side of Equation 5.12 is nonnegative. Lemma 2 and Equation 5.5b show that the fourth term on the right hand side of Equation 5.12 is nonnegative. It follows that

$$(\xi_{j+1} - \xi_*)^\top P (\xi_{j+1} - \xi_*) \leq \tau^2 (\xi_j - \xi_*)^\top P (\xi_j - \xi_*).$$

Inducting from $j = 0$ to $k - 1$, we see that

$$(\xi_k - \xi_*)^\top P (\xi_k - \xi_*) \leq \tau^{2k} (\xi_0 - \xi_*)^\top P (\xi_0 - \xi_*),$$

for all k. It follows that

$$\|\xi_k - \xi_*\| \leq \sqrt{\kappa_P} \|\xi_0 - \xi_*\| \tau^k.$$

From this, we may conclude that

$$\|\varphi_k - \varphi_*\| \leq \kappa_B \sqrt{\kappa_P} \|\varphi_0 - \varphi_*\| \tau^k$$

as desired. □

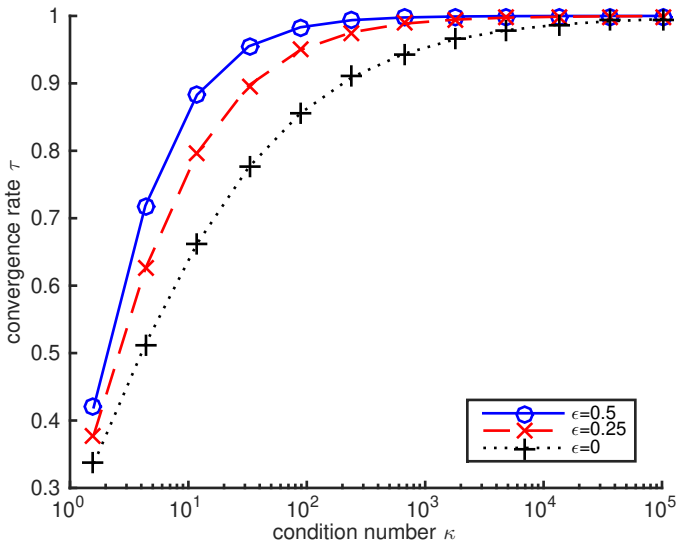


Figure 5.1: For $\alpha = 1.5$ and for several choices of ϵ in $\rho_0 = \kappa^\epsilon$, we plot the minimal rate τ for which the linear matrix inequality in Equation 5.11 is satisfied as a function of κ.

For fixed values of α, ρ_0, $\hat{m}$, $\hat{L}$, and τ, the feasibility of Equation 5.11 is a semidefinite program with variables P, λ^1, and λ^2. We perform a binary search over τ to find the minimal rate τ such that the linear matrix inequality in Equation 5.11 is satisfied. The results are shown in Figure 5.1 for a wide range of condition numbers κ, for $\alpha = 1.5$, and for several choices of step size ρ_0. In Figure 5.2, we plot the values $-1/\log\tau$ to show the number of iterations required to achieve a desired accuracy.

Note that when we choose $\rho_0 = \kappa^\epsilon$, then the matrix M^1 is given by

$$M^1 = \begin{bmatrix} -2\kappa^{-2\epsilon} & \kappa^{-\frac{1}{2}-\epsilon} + \kappa^{\frac{1}{2}-\epsilon} \\ \kappa^{-\frac{1}{2}-\epsilon} + \kappa^{\frac{1}{2}-\epsilon} & -2 \end{bmatrix},$$

and so the linear matrix inequality in Equation 5.11 depends only on κ and not on $\hat{m}$ and $\hat{L}$. Therefore, we will consider step sizes of this form (recall from Equation 5.4 that $\rho = (\hat{m}\hat{L})^{\frac{1}{2}}\rho_0$). The choice $\epsilon = 0$ is common in the literature [62], but requires the user to know the strong-convexity parameter $\hat{m}$. We also consider the choice $\epsilon = 0.5$, which produces worse guarantees, but does not require knowledge of $\hat{m}$.

One weakness of Theorem 6 is the fact that the rate we produce is not given as a function of κ. To use Theorem 6 as stated, we first specify the condition number (for example, $\kappa = 1000$). Then we search for the minimal τ such that Equation 5.11 is feasible. This produces an upper bound on the convergence rate of Algorithm 2 (for example, $\tau = 0.9$). To remedy this problem, in Section 5.5, we demonstrate how Theorem 6 can be used to obtain the convergence rate of Algorithm 2 as a symbolic function of the step size ρ and the over-relaxation parameter α.

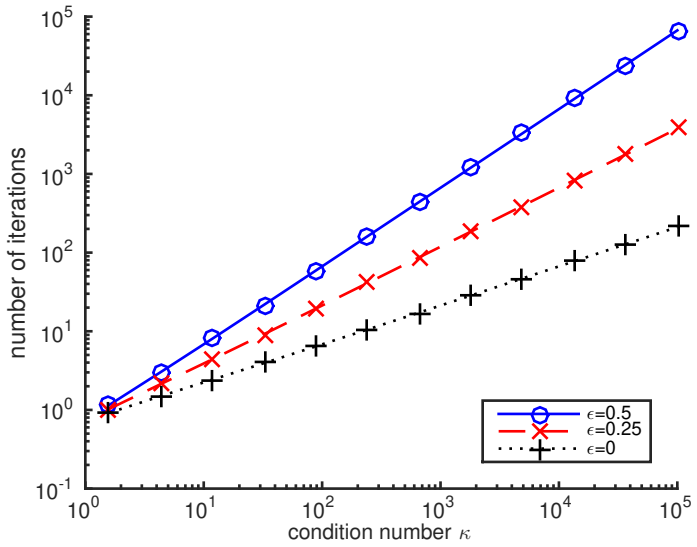


Figure 5.2: For $\alpha = 1.5$ and for several choices of ϵ in $\rho_0 = \kappa^\epsilon$, we compute the minimal rate τ such that the linear matrix inequality in Equation 5.11 is satisfied, and we plot $-1/\log\tau$ as a function of κ.

5.5 Symbolic Rates for Various ρ and α

In Section 5.4, we demonstrated how to use semidefinite programming to produce numerical convergence rates. That is, given a choice of algorithm parameters and the condition number κ, we could determine the convergence rate of Algorithm 2. In this section, we show how Theorem 6 can be used to prove symbolic convergence rates. That is, we describe the convergence rate of Algorithm 2 as a function of ρ, α, and κ. In Theorem 7, we prove the linear convergence of Algorithm 2 for all choices $\alpha \in (0, 2)$ and $\rho = (\hat{m}\hat{L})^{\frac{1}{2}}\kappa^\epsilon$, with $\epsilon \in (-\infty, \infty)$. This result generalizes a number of results in the literature. As two examples, [62] consider the case $\epsilon = 0$ and [41] consider the case $\alpha = 1$ and $\epsilon = 0.5$.

The rate given in Theorem 7 is loose by a factor of four relative to the lower bound given in Theorem 8. However, weakening the rate by a constant factor eases the proof by making it easier to find a certificate for use in Equation 5.11.

Theorem 7. *Suppose that Assumption 3 holds. Let the sequences (x_k), (z_k), and (u_k) be generated by running Algorithm 2 with parameter $\alpha \in (0, 2)$ and with step size $\rho = (\hat{m}\hat{L})^{\frac{1}{2}}\kappa^\epsilon$, where $\epsilon \in (-\infty, \infty)$. Define x_*, z_*, u_*, φ_k, and φ_* as in Theorem 6. Then for all sufficiently large κ, we have*

$$\|\varphi_k - \varphi_*\| \leq C\|\varphi_0 - \varphi_*\| \left(1 - \frac{\alpha}{2\kappa^{0.5+|\epsilon|}}\right)^k,$$

where

$$C = \kappa_B \sqrt{\max\left\{\tfrac{\alpha}{2-\alpha}, \tfrac{2-\alpha}{\alpha}\right\}}.$$

Proof. We claim that for all sufficiently large κ, the linear matrix inequality in Equation 5.11 is satisfied with the rate $\tau = 1 - \frac{\alpha}{2\kappa^{0.5+|\epsilon|}}$ and with certificate

$$\lambda^1 = \alpha\kappa^{\epsilon-0.5} \quad \lambda^2 = \alpha \quad P = \begin{bmatrix} 1 & \alpha - 1 \\ \alpha - 1 & 1 \end{bmatrix}.$$

The matrix on the right hand side of Equation 5.11 can be expressed as $-\frac{1}{4}\alpha\kappa^{-2}M$, where M is a symmetric 4×4 matrix whose last row and column consist of zeros. We wish to prove that M is positive semidefinite for all sufficiently large κ. To do so, we consider the cases $\epsilon \geq 0$ and $\epsilon < 0$ separately, though the two cases will be nearly identical. First suppose that $\epsilon \geq 0$. In this case, the nonzero entries of M are specified by

$$\begin{aligned}
M_{11} &= \alpha\kappa^{1-2\epsilon} + 4\kappa^{\frac{3}{2}-\epsilon} \\
M_{12} &= \alpha^2\kappa^{1-2\epsilon} - \alpha\kappa^{1-2\epsilon} + 12\kappa^{\frac{3}{2}-\epsilon} - 4\alpha\kappa^{\frac{3}{2}-\epsilon} \\
M_{13} &= 4\kappa + 8\kappa^{\frac{3}{2}-\epsilon} \\
M_{22} &= 8\kappa^2 - 4\alpha\kappa^2 + \alpha\kappa^{1-2\epsilon} + 4\kappa^{\frac{3}{2}-\epsilon} \\
M_{23} &= 4\kappa + 8\kappa^2 - 4\alpha\kappa^2 + 8\kappa^{\frac{3}{2}-\epsilon} \\
M_{33} &= 8\kappa + 8\kappa^2 - 4\alpha\kappa^2 + 8\kappa^{\frac{3}{2}-\epsilon} + 8\kappa^{\frac{3}{2}+\epsilon}.
\end{aligned}$$

We show that each of the first three leading principal minors of M is positive for sufficiently large κ. To understand the behavior of the leading principal minors, it suffices to look at their leading terms. For large κ, the first leading principal minor (which is simple M_{11}) is dominated by the term $4\kappa^{\frac{3}{2}-\epsilon}$, which is positive. Similarly, the second leading principal minor is dominated by the term $16(2-\alpha)\kappa^{\frac{7}{2}-\epsilon}$, which is positive. When $\epsilon > 0$, the third leading principal minor is dominated by the term $128(2-\alpha)\kappa^5$, which is positive. When $\epsilon = 0$, the third leading principal minor is dominated by the term $64\alpha(2-\alpha)^2\kappa^5$, which is positive. Since these leading coefficients are all positive, it follows that for all sufficiently large κ, the matrix M is positive semidefinite.

Now suppose that $\epsilon < 0$. In this case, the nonzero entries of M are specified by

$$\begin{aligned}
M_{11} &= 8\kappa^{\frac{3}{2}-\epsilon} - 4\kappa^{\frac{3}{2}+\epsilon} + \alpha\kappa^{1+2\epsilon} \\
M_{12} &= 8\kappa^{\frac{3}{2}-\epsilon} + 4\kappa^{\frac{3}{2}+\epsilon} - 4\alpha\kappa^{\frac{3}{2}+\epsilon} - \alpha\kappa^{1+2\epsilon} + \alpha^2\kappa^{1+2\epsilon} \\
M_{13} &= 4\kappa + 8\kappa^{\frac{3}{2}-\epsilon} \\
M_{22} &= 8\kappa^2 - 4\alpha\kappa^2 + 8\kappa^{\frac{3}{2}-\epsilon} - 4\kappa^{\frac{3}{2}+\epsilon} + \alpha\kappa^{1+2\epsilon} \\
M_{23} &= 4\kappa + 8\kappa^2 - 4\alpha\kappa^2 + 8\kappa^{\frac{3}{2}-\epsilon} \\
M_{33} &= 8\kappa + 8\kappa^2 - 4\alpha\kappa^2 + 8\kappa^{\frac{3}{2}-\epsilon} + 8\kappa^{\frac{3}{2}+\epsilon}.
\end{aligned}$$

As before, we show that each of the first three leading principal minors of M is positive. For large κ, the first leading principal minor (which is simple M_{11}) is dominated by the

term $8\kappa^{\frac{3}{2}-\epsilon}$, which is positive. Similarly, the second leading principal minor is dominated by the term $32(2-\alpha)\kappa^{\frac{7}{2}-\epsilon}$, which is positive. The third leading principal minor is dominated by the term $128(2-\alpha)\kappa^5$, which is positive. Since these leading coefficients are all positive, it follows that for all sufficiently large κ, the matrix M is positive semidefinite.

The result now follows from Theorem 6 by noting that P has eigenvalues α and $2-\alpha$. □

Note that since the matrix P doesn't depend on ρ, the proof holds even when the step size changes at each iteration.

5.6 Lower Bounds

In this section, we probe the tightness of the upper bounds on the convergence rate of Algorithm 2 given by Theorem 6. The construction of the lower bound in this section is similar to a construction given in [59].

Let Q be a d-dimensional symmetric positive-definite matrix whose largest and smallest eigenvalues are L and m respectively. Let $f(x) = \frac{1}{2}x^\top Qx$ be a quadratic and let $g(z) = \frac{\delta}{2}\|z\|^2$ for some $\delta \geq 0$. Let $A = I_d$, $B = -I_d$, and $c = 0$. With these definitions, the optimization problem in Equation 5.1 is solved by $x = z = 0$. The updates for Algorithm 2 are given by

$$x_{k+1} = \rho(Q + \rho I)^{-1}(z_k - u_k) \tag{5.13a}$$

$$z_{k+1} = \frac{\rho}{\delta + \rho}(\alpha x_{k+1} + (1-\alpha)z_k + u_k) \tag{5.13b}$$

$$u_{k+1} = u_k + \alpha x_{k+1} + (1-\alpha)z_k - z_{k+1}. \tag{5.13c}$$

Solving for z_k in Equation 5.13b and substituting the result into Equation 5.13c gives $u_{k+1} = \frac{\delta}{\rho}z_{k+1}$. Then eliminating x_{k+1} and u_k from Equation 5.13b using Equation 5.13a and the fact that $u_k = \frac{\delta}{\rho}z_k$ allows us to express the update rule purely in terms of z as

$$z_{k+1} = \underbrace{\left(\frac{\alpha\rho(\rho-\delta)}{\rho+\delta}(Q+\rho I)^{-1} + \frac{\rho - \alpha\rho + \delta}{\rho+\delta}I\right)}_{T} z_k.$$

Note that the eigenvalues of T are given by

$$1 - \frac{\alpha\rho(\lambda+\delta)}{(\rho+\delta)(\lambda+\rho)}, \tag{5.14}$$

where λ is an eigenvalue of Q. We will use this setup to construct a lower bound on the worst-case convergence rate of Algorithm 2 in Theorem 8.

Theorem 8. *Suppose that Assumption 3 holds. The worst-case convergence rate of Algorithm 2, when run with step size $\rho = (\hat{m}\hat{L})^{\frac{1}{2}}\kappa^\epsilon$ and over-relaxation parameter α, is lower-bounded by*

$$1 - \frac{2\alpha}{1 + \kappa^{0.5+|\epsilon|}}. \tag{5.15}$$

Proof. First consider the case $\epsilon \geq 0$. Choosing $\delta = 0$ and $\lambda = m$, from Equation 5.14, we see that T has eigenvalue

$$1 - \frac{\alpha}{1 + \kappa^{0.5+\epsilon}}. \tag{5.16}$$

When initialized with z as the eigenvector corresponding to this eigenvalue, Algorithm 2 will converge linearly with rate given exactly by Equation 5.16, which is lower bounded by the expression in Equation 5.15 when $\epsilon \geq 0$.

Now suppose that $\epsilon < 0$. Choosing $\delta = L$ and $\lambda = L$, after multiplying the numerator and denominator of Equation 5.14 by $\kappa^{0.5-\epsilon}$, we see that T has eigenvalue

$$1 - \frac{2\alpha}{(1 + \kappa^{0.5-\epsilon})(\kappa^{-0.5+\epsilon} + 1)} \geq 1 - \frac{2\alpha}{1 + \kappa^{0.5-\epsilon}}. \tag{5.17}$$

When initialized with z as the eigenvector corresponding to this eigenvalue, Algorithm 2 will converge linearly with rate given exactly by the left hand side of Equation 5.17, which is lower bounded by the expression in Equation 5.15 when $\epsilon < 0$. □

Figure 5.3 compares the lower bounds given by Equation 5.16 with the upper bounds given by Theorem 6 for $\alpha = 1.5$ and for several choices of $\rho = (\hat{m}\hat{L})^{\frac{1}{2}}\kappa^{\epsilon}$ satisfying $\epsilon \geq 0$. The upper and lower bounds agree visually on the range of choices ϵ depicted, demonstrating the practical tightness of the upper bounds given by Theorem 6 for a large range of choices of parameter values.

5.7 Related Work

Several recent papers have studied the linear convergence of Algorithm 1 but do not extend to Algorithm 2. [41] prove a linear rate of convergence for ADMM in the strongly convex case. [78] prove the linear convergence of a specialization of ADMM to a class of distributed optimization problems under a local strong-convexity condition. [75] prove the linear convergence of a generalization of ADMM to a multiterm objective in the setting where each term can be decomposed as a strictly convex function and a polyhedral function. In particular, this result does not require the terms to be strongly convex.

More generally, there are a number of results for operator splitting methods in the literature. [97] and [52] analyze the convergence of several operator splitting schemes. More recently, [121, 122] prove the equivalence of forward-backward splitting and Douglas–Rachford splitting with a scaled version of the gradient method applied to unconstrained nonconvex surrogate functions (called the forward-backward envelope and the Douglas–Rachford envelope respectively). [64] propose an accelerated version of ADMM in the spirit of Nesterov, and prove a $O(1/k^2)$ convergence rate in the case where f and g are both strongly convex and g is quadratic.

The theory of over-relaxed ADMM is more limited. [51] prove the convergence of over-relaxed ADMM but do not give a rate. More recently, [37, 38] analyze the convergence

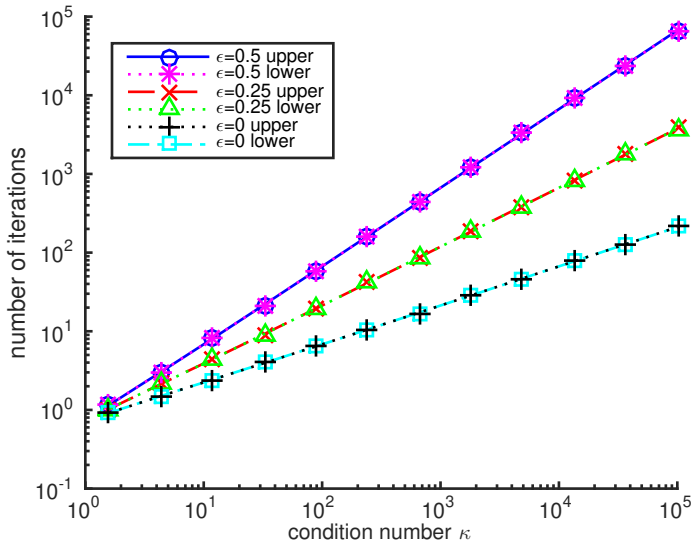


Figure 5.3: For $\alpha = 1.5$ and for several choices ϵ in $\rho_0 = \kappa^\epsilon$, we plot $-1/\log\tau$ as a function of κ, both for the lower bound on τ given by Equation 5.16 and the upper bound on τ given by Theorem 6. For each choice of ϵ in $\{0.5, 0.25, 0\}$, the lower and upper bounds agree visually. This agreement demonstrates the practical tightness of the upper bounds given by Theorem 6 for a large range of choices of parameter values.

rates of ADMM in a variety of settings. [62] prove the linear convergence of Douglas–Rachford splitting in the strongly-convex setting. They use the fact that ADMM is Douglas–Rachford splitting applied to the dual problem [51] to derive a linear convergence rate for over-relaxed ADMM with a specific choice of step size ρ. [50] gives convergence results for several specializations of ADMM, and found that over-relaxation with $\alpha = 1.5$ empirically sped up convergence. [59] give some guidance on tuning over-relaxed ADMM in the quadratic case.

Unlike prior work, our framework requires no assumptions on the parameter choices in Algorithm 2. For example, Theorem 6 certifies the linear convergence of Algorithm 2 even for values $\alpha > 2$. In our framework, certifying a convergence rate for an arbitrary choice of parameters amounts to checking the feasibility of a 4×4 semidefinite program, which is essentially instantaneous, as opposed to formulating a proof.

5.8 Selecting Algorithm Parameters

In this section, we show how to use the results of Section 5.4 to select the parameters α and ρ in Algorithm 2 and we show the effect on a numerical example.

Recall that given a choice of parameters α and ρ and given the condition number κ,

Theorem 6 gives an upper bound on the convergence rate of Algorithm 2. Therefore, one approach to parameter selection is to do a grid search over the space of parameters for the choice that minimizes the upper bound provided by Theorem 6. We demonstrate this approach numerically for a distributed Lasso problem, but first we demonstrate that the usual range of $(0, 2)$ for the over-relaxation parameter α is too limited, that more choices of α lead to linear convergence. In Figure 5.4, we plot the largest value of α found through binary search such that Equation 5.11 is satisfied for some $\tau < 1$ as a function of κ. Proof techniques in prior work do not extend as easily to values of $\alpha > 2$. In our framework, we simply change some constants in a small semidefinite program.

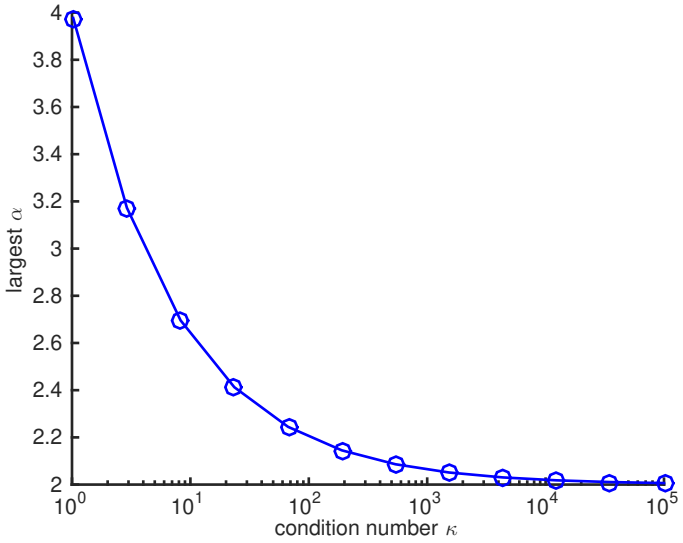


Figure 5.4: As a function of κ, we plot the largest value of α such that Equation 5.11 is satisfied for some $\tau < 1$. In this figure, we set $\epsilon = 0$ in $\rho_0 = \kappa^\epsilon$.

Distributed Lasso

Following [41], we give a numerical demonstration with a distributed Lasso problem of the form

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} \frac{1}{2\mu} \|A_i x_i - b_i\|^2 + \|z\|_1 \\ \text{subject to} \quad & x_i - z = 0 \quad \text{for all} \quad i = 1, \ldots, N. \end{aligned}$$

Each A_i is a tall matrix with full column rank, and so the first term in the objective will be strongly convex and its gradient will be Lipschitz continuous. As in [41], we choose $N = 5$ and $\mu = 0.1$. Each A_i is generated by populating a 600×500 matrix with independent

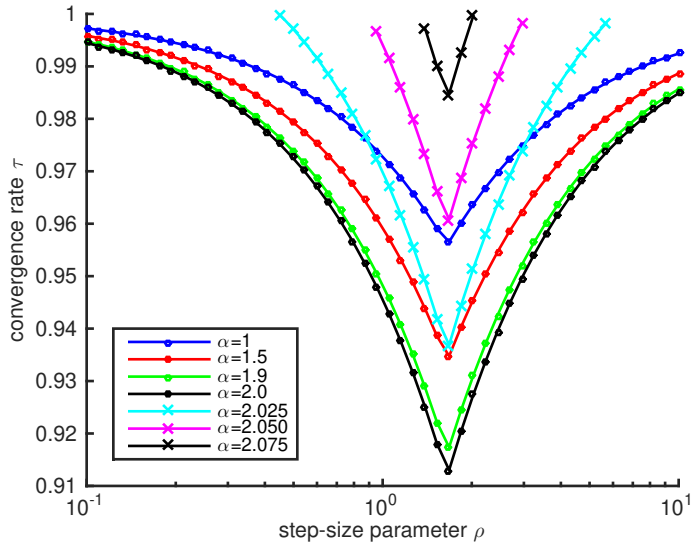


Figure 5.5: We compute the upper bounds on the convergence rate given by Theorem 6 for a grid of eighty-five values of α evenly spaced between 0.1 and 2.2 and a grid of fifty values of ρ geometrically spaced between 0.1 and 10. Each line corresponds to a fixed choice of α, and we plot only a subset of the values of α to keep the plot manageable. We omit points corresponding to parameter values for which Equation 5.11 is not feasible for any value of $\tau < 1$. This analysis suggests choosing $\alpha = 2.0$ and $\rho = 1.7$.

standard normal entries and normalizing the columns. We generate each b_i via $b_i = A_i x^0 + \epsilon_i$, where x^0 is a sparse 500-dimensional vector with 250 independent standard normal entries, and $\epsilon_i \sim \mathcal{N}(0, 10^{-3}I)$.

In Figure 5.5, we compute the upper bounds on the convergence rate given by Theorem 6 for a grid of values of α and ρ. Each line corresponds to a fixed choice of α, and we plot only a subset of the values of α to keep the plot manageable. We omit points corresponding to parameter values for which the linear matrix inequality in Equation 5.11 was not feasible for any value of $\tau < 1$.

In Figure 5.6, we run Algorithm 2 for the same values of α and ρ. We then plot the number of iterations needed for z_k to reach within 10^{-6} of a precomputed reference solution. We plot lines corresponding to only a subset of the values of α to keep the plot manageable, and we omit points corresponding to parameter values for which Algorithm 2 exceeded 1000 iterations. For the most part, the performance of Algorithm 2 as a function of ρ closely tracked the performance predicted by the upper bounds in Figure 5.5. Notably, smaller values of α seem more robust to poor choices of ρ. The parameters suggested by our analysis perform close to the best of any parameter choices.

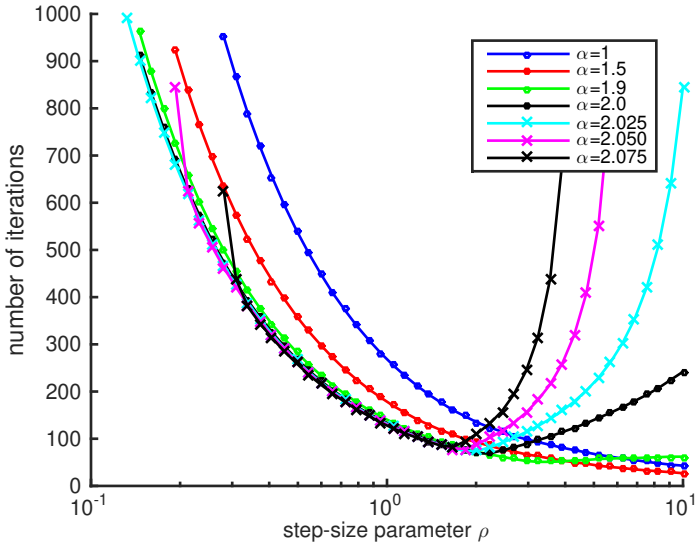


Figure 5.6: We run Algorithm 2 for up to 1000 iterations for a grid of eighty-five values of α evenly spaced between 0.1 and 2.2 and a grid of fifty value of ρ geometrically spaced between 0.1 and 10. We plot the number of iterations required for z_k to reach within 10^{-6} of a precomputed reference solution. We plot lines corresponding to only a subset of the values of α to keep the plot manageable. We omit points corresponding to parameter values for which Algorithm 2 exceeded 1000 iterations.

5.9 Conclusion

We showed that a framework based on semidefinite programming can be used to prove convergence rates for the alternating direction method of multipliers and allows a unified treatment of the algorithm's many variants, which arise through the introduction of additional parameters. We showed how to use this framework for establishing convergence rates, as in Theorem 6 and Theorem 7, and how to use this framework for parameter selection in practice, as in Section 5.8. The potential uses are numerous. This framework makes it straightforward to propose new algorithmic variants, for example, by introducing new parameters into Algorithm 2 and using Theorem 6 to see if various settings of these new parameters give rise to improved guarantees. In the case that Assumption 3 does not hold, the most likely cause is that we lack the strong convexity of f. One approach to handling this is to run Algorithm 2 on the modified function $f(x) + \frac{\delta}{2}\|x\|^2$. By completing the square in the x update, we see that this amounts to an extremely minor algorithmic modification (it only affects the x update).

It should be clear that other operator splitting methods such as Douglas–Rachford splitting and forward-backward splitting can be cast in this framework and analyzed using the tools presented here.

Chapter 6

Case Study: Distributed Submodular Function Optimization

Submodular functions describe a variety of discrete problems in machine learning, signal processing, and computer vision. However, minimizing submodular functions poses a number of algorithmic challenges. Recent work introduced an easy-to-use, parallelizable algorithm for minimizing submodular functions that decompose as the sum of "simple" submodular functions. Because this algorithm decomposes the solution to each step of the larger optimization problem into the solutions to many independent smaller optimization problems, this algorithm is particularly amenable to parallel execution and can leverage distributed architectures like the one described in Chapter 3. Empirically, this algorithm performs extremely well, but no theoretical analysis was given. In this chapter, we show that the algorithm converges linearly, and we provide upper and lower bounds on the rate of convergence. Our proof relies on the geometry of submodular polyhedra and draws on results from spectral graph theory.[1]

6.1 Introduction

A large body of recent work demonstrates that many discrete problems in machine learning can be phrased as the optimization of a submodular set function [11, 89]. A set function $F\colon 2^V \to \mathbb{R}$ over a ground set V of N elements is *submodular* if the inequality $F(A)+F(B) \geq F(A \cup B)+F(A \cap B)$ holds for all subsets $A, B \subseteq V$. Problems like clustering [110], structured sparse variable selection [10], MAP inference with higher-order potentials [86], and corpus extraction problems [96] can be reduced to the problem of submodular function minimization (SFM), that is

$$\min_{A \subseteq V} F(A). \tag{P1}$$

[1] Material in this chapter is based adapted from [112].

Although SFM is solvable in polynomial time, existing algorithms are often inefficient on large-scale problems. For this reason, the development of scalable, parallelizable algorithms has been an active area of research [80, 81, 87, 137]. Approaches to solving Problem Equation P1 are either based on combinatorial optimization or on convex optimization via the *Lovász extension*.

Functions that occur in practice are usually not arbitrary and frequently possess additional exploitable structure. For example, a number of submodular functions admit specialized algorithms that solve Problem Equation P1 very quickly. Examples include cut functions on certain kinds of graphs, concave functions of the cardinality $|A|$, and functions counting joint ancestors in trees. We will use the term *simple* to refer to functions F for which we have a fast subroutine for minimizing $F+s$, where $s \in \mathbb{R}^N$ is any modular function. We treat these subroutines as black boxes. Many commonly occuring submodular functions (for example, graph cuts, hypergraph cuts, MAP inference with higher-order potentials [54, 86, 147], co-segmentation [74], certain structured-sparsity inducing functions [82], covering functions [137], and combinations thereof) can be expressed as a sum

$$F(A) = \sum_{r=1}^{R} F_r(A) \tag{6.1}$$

of simple submodular functions. Recent work demonstrates that this structure offers important practical benefits [81, 87, 137]. For instance, it admits iterative algorithms that minimize each F_r separately and combine the results in a straightforward manner (for example, dual decomposition).

In particular, it has been shown that the minimization of decomposable functions can be rephrased as a *best-approximation problem*, the problem of finding the closest points in two convex sets [81]. This formulation brings together SFM and classical projection methods and yields empirically fast, parallel, and easy-to-implement algorithms. In these cases, the performance of projection methods depends heavily on the specific geometry of the problem at hand and is not well understood in general. Indeed, while [81] show good empirical results, the analysis of this alternative approach to SFM was left as an open problem.

Contributions. In this work, we study the geometry of the submodular best-approximation problem and ground the prior empirical results in theoretical guarantees. We show that SFM via alternating projections, or block coordinate descent, converges at a *linear rate*. We show that this rate holds for the best-approximation problem, relaxations of SFM, and the original discrete problem. More importantly, we prove upper and lower bounds on the worst-case rate of convergence. Our proof relies on analyzing angles between the polyhedra associated with submodular functions and draws on results from spectral graph theory. It offers insight into the geometry of submodular polyhedra that may be beneficial beyond the analysis of projection algorithms.

Submodular minimization. The first polynomial-time algorithm for minimizing arbitrary submodular functions was a consequence of the ellipsoid method [68]. Strongly and weakly polynomial-time combinatorial algorithms followed [101]. The current fastest

running times are $O(N^5\tau_1 + N^6)$ [117] in general and $O((N^4\tau_1 + N^5)\log F_{\max})$ for integer-valued functions [79], where $F_{\max} = \max_A |F(A)|$ and τ_1 is the time required to evaluate F. Some work has addressed decomposable functions [81, 87, 137]. The running times in [87] apply to integer-valued functions and range from $O((N+R)^2 \log F_{\max})$ for cuts to $O((N+Q^2R)(N+Q^2R+QR\tau_2)\log F_{\max})$, where $Q \leq N$ is the maximal cardinality of the support of any F_r, and τ_2 is the time required to minimize a simple function. [137] use a convex optimization approach based on Nesterov's smoothing technique. They achieve a (sublinear) convergence rate of $O(1/k)$ for the discrete SFM problem. Their results and our results do not rely on the function being integral.

Projection methods. Algorithms based on alternating projections between convex sets (and related methods such as the Douglas-Rachford algorithm) have been studied extensively for solving convex-feasibility and best-approximation problems [15, 16, 19, 43, 46, 70, 71, 144, 148]. See [44] for a survey of applications. In the simple case of subspaces, the convergence of alternating projections has been characterized in terms of the Friedrichs angle c_F between the subspaces [16, 14]. There are often good ways to compute c_F (see Lemma 14), which allow us to obtain concrete linear rates of convergence for subspaces. The general case of alternating projections between arbitrary convex sets is less well understood. [17] give a general condition for the linear convergence of alternating projections in terms of the value κ_* (defined in Section 6.3). However, except in very limited cases, it is unclear how to compute or even bound κ_*. While it is known that $\kappa_* < \infty$ for polyhedra [16, Corollary 5.26], the rate may be arbitrarily slow, and the challenge is to bound the linear rate away from one. We are able to give a specific *uniform* linear rate for the submodular polyhedra that arise in SFM.

Although both κ_* and c_F are useful quantities for understanding the convergence of projection methods, they largely have been studied independently of one another. In this work, we relate these two quantities for polyhedra, thereby obtaining some of the generality of κ_* along with the computability of c_F. To our knowledge, we are the first to relate κ_* and c_F outside the case of subspaces. We feel that this connection may be useful beyond the context of submodular polyhedra.

Background

Throughout this chapter, we assume that F is a sum of simple submodular functions $F_1, \ldots, F_R$ and that $F(\emptyset) = 0$. Points $s \in \mathbb{R}^N$ can be identified with (modular) set functions via $s(A) = \sum_{n \in A} s_n$. The *base polytope* of F is defined as the set of all modular functions that are dominated by F and that sum to $F(V)$,

$$B(F) = \{s \in \mathbb{R}^N \mid s(A) \leq F(A) \text{ for all } A \subseteq V \text{ and } s(V) = F(V)\}.$$

The *Lovász extension* $f\colon \mathbb{R}^N \to \mathbb{R}$ of F can be written as the support function of the base polytope, that is $f(x) = \max_{s \in B(F)} s^\top x$. Even though $B(F)$ may have exponentially many faces, the extension f can be evaluated in $O(N \log N)$ time [53]. The discrete SFM problem

in Problem Equation P1 can be relaxed to the non-smooth convex optimization problem

$$\min_{x\in[0,1]^N} f(x) \quad \equiv \quad \min_{x\in[0,1]^N} \sum_{r=1}^{R} f_r(x), \tag{P2}$$

where f_r is the Lovász extension of F_r. This relaxation is exact – rounding an optimal continuous solution yields the indicator vector of an optimal discrete solution. The formulation in Problem Equation P2 is amenable to dual decomposition [88] and smoothing techniques [137], but suffers from the non-smoothness of f [81]. Alternatively, we can formulate a proximal version of the problem

$$\min_{x\in\mathbb{R}^N} f(x) + \tfrac{1}{2}\|x\|^2 \quad \equiv \quad \min_{x\in\mathbb{R}^N} \sum_{r=1}^{R} (f_r(x) + \tfrac{1}{2R}\|x\|^2). \tag{P3}$$

By thresholding the optimal solution of Problem Equation P3 at zero, we also recover the indicator vector of an optimal discrete solution [11, Proposition 8.4].

Lemma 9. *The dual of the right-hand version of Problem Equation P3 is the best-approximation problem*

$$\min \ \|a-b\|^2 \quad a\in\mathcal{A}, \ \ b\in\mathcal{B}, \tag{P4}$$

where $\mathcal{A} = \{(a_1,\dots,a_R)\in\mathbb{R}^{NR} \mid \sum_{r=1}^{R} a_r = 0\}$ *and* $\mathcal{B} = B(F_1)\times\cdots\times B(F_R)$.

Lemma 9 is shown in [81]. It implies that we can minimize a decomposable submodular function by solving Problem Equation P4, which means finding the closest points between the subspace $\mathcal{A}$ and the product $\mathcal{B}$ of base polytopes. Projecting onto $\mathcal{A}$ is straightforward because $\mathcal{A}$ is a subspace. Projecting onto $\mathcal{B}$ amounts to projecting onto each $B(F_r)$ separately. The projection $\Pi_{B(F_r)}z$ of a point z onto $B(F_r)$ may be solved by minimizing $F_r - z$ [81]. We can compute these projections easily because each F_r is simple.

Throughout this chapter, we use $\mathcal{A}$ and $\mathcal{B}$ to refer to the specific polyhedra defined in Lemma 9 (which live in $\mathbb{R}^{NR}$) and P and Q to refer to general polyhedra (sometimes arbitrary convex sets) in $\mathbb{R}^D$. Note that the polyhedron $\mathcal{B}$ depends on the submodular functions $F_1,\dots,F_R$, but we omit the dependence to simplify our notation. Our bound will be uniform over all submodular functions.

6.2 Algorithm and Idea of Analysis

A popular class of algorithms for solving best-approximation problems are projection methods [16]. The most straightforward approach uses alternating projections (AP) or block coordinate descent. Start with any point $a_0\in\mathcal{A}$, and inductively generate two sequences via $b_k = \Pi_{\mathcal{B}}a_k$ and $a_{k+1} = \Pi_{\mathcal{A}}b_k$. Given the nature of $\mathcal{A}$ and $\mathcal{B}$, this algorithm is easy to implement and use in our setting. [81] propose to solve Problem P4 with AP between $\mathcal{A}$ and $\mathcal{B}$. This is the algorithm that we will analyze.

The sequence (a_k, b_k) will eventually converge to an optimal pair (a_*, b_*). We say that AP converges linearly with rate $\alpha < 1$ if $\|a_k - a_*\| \leq C_1\alpha^k$ and $\|b_k - b_*\| \leq C_2\alpha^k$ for all k and for some constants C_1 and C_2. Smaller values of α are better.

Analysis: Intuition. We will provide a detailed analysis of the convergence of AP for the polyhedra $\mathcal{A}$ and $\mathcal{B}$. To motivate our approach, we first provide some intuition with the following much-simplified setup. Let U and V be one-dimensional subspaces spanned by the unit vectors u and v respectively. In this case, it is known that AP converges linearly with rate $\cos^2\theta$, where $\theta \in [0, \frac{\pi}{2}]$ is the angle such that $\cos\theta = u^\top v$. The smaller the angle, the slower the rate of convergence. For subspaces U and V of higher dimension, the relevant generalization of the "angle" between the subspaces is the *Friedrichs angle* [43, Definition 9.4], whose cosine is given by

$$c_F(U, V) = \sup\left\{u^\top v \,|\, u \in U \cap (U \cap V)^\perp, v \in V \cap (U \cap V)^\perp, \|u\| \leq 1, \|v\| \leq 1\right\}. \tag{6.2}$$

In finite dimensions, $c_F(U, V) < 1$. In general, when U and V are subspaces of arbitrary dimension, AP will converge linearly with rate $c_F(U, V)^2$ [43, Theorem 9.8]. If U and V are *affine spaces*, AP still converges linearly with rate $c_F(U-u, V-v)^2$, where $u \in U$ and $v \in V$.

We are interested in rates for *polyhedra* P and Q, which we define as the intersection of finitely many halfspaces. We generalize the preceding results by considering all pairs (P_x, Q_y) of faces of P and Q and showing that the convergence rate of AP between P and Q is at worst $\max_{x,y} c_F(\text{aff}_0(P_x), \text{aff}_0(Q_y))^2$, where $\text{aff}(C)$ is the affine hull of C and $\text{aff}_0(C) = \text{aff}(C) - c$ for some $c \in C$. The faces $\{P_x\}_{x \in \mathbb{R}^D}$ of P are defined as the maximizers of linear functions over P, that is

$$P_x = \arg\max_{p \in P} x^\top p. \tag{6.3}$$

While we look at angles between pairs of faces, we remark that [45] consider a different generalization of the "angle" between arbitrary convex sets.

Roadmap of the Analysis. Our analysis has two main parts. First, we relate the convergence rate of AP between polyhedra P and Q to the angles between the faces of P and Q. To do so, we give a general condition under which AP converges linearly (Theorem 10), which we show depends on the angles between the faces of P and Q (Corollary 13) in the polyhedral case. Second, we specialize to the polyhedra $\mathcal{A}$ and $\mathcal{B}$, and we equate the angles with eigenvalues of certain matrices and use tools from spectral graph theory to bound the relevant eigenvalues in terms of the conductance of a specific graph. This yields a worst-case bound of $1 - \frac{1}{N^2R^2}$ on the rate, stated in Theorem 20.

In Theorem 22, we show a lower bound of $1 - \frac{2\pi^2}{N^2R}$ on the worst-case convergence rate.

6.3 The Upper Bound

We first derive an upper bound on the rate of convergence of AP between the polyhedra $\mathcal{A}$ and $\mathcal{B}$.

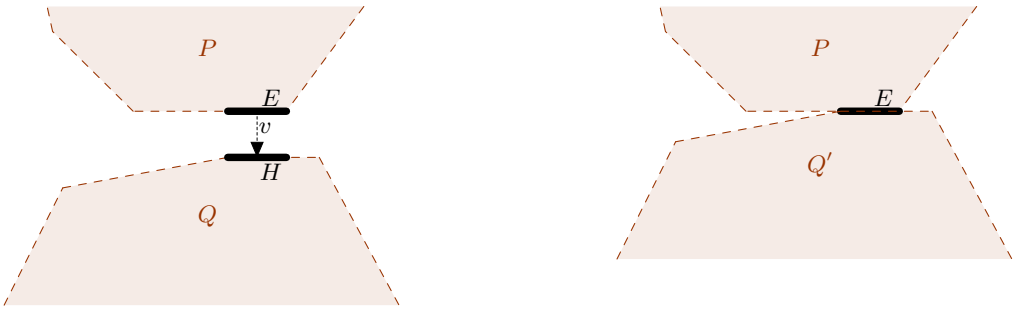


Figure 6.1: The optimal sets E, H in Equation Equation 6.4, the vector v, and the shifted polyhedron Q'.

A Condition for Linear Convergence

We begin with a condition under which AP between two closed convex sets P and Q converges linearly. This result is similar to that of [17, Corollary 3.14], but the rate we achieve is twice as fast and relies on slightly weaker assumptions.

We will need a few definitions from [17]. Let $d(K_1, K_2) = \inf\{\|k_1 - k_2\| : k_1 \in K_1, k_2 \in K_2\}$ be the distance between sets K_1 and K_2. Define the sets of "closest points" as

$$E = \{p \in P \,|\, d(p, Q) = d(P, Q)\} \qquad H = \{q \in Q \,|\, d(q, P) = d(Q, P)\}, \tag{6.4}$$

and let $v = \Pi_{Q-P}0$ (see Figure 6.1). Note that $H = E + v$, and when $P \cap Q \neq \emptyset$ we have $v = 0$ and $E = H = P \cap Q$. Therefore, we can think of the pair (E, H) as a generalization of the intersection $P \cap Q$ to the setting where P and Q do not intersect. Pairs of points $(e, e + v) \in E \times H$ are solutions to the best-approximation problem between P and Q. In our analysis, we will mostly study the translated version $Q' = Q - v$ of Q that intersects P at E.

For $x \in \mathbb{R}^D \backslash E$, the function κ relates the distance to E with the distances to P and Q',

$$\kappa(x) = \frac{d(x, E)}{\max\{d(x, P), d(x, Q')\}}.$$

If κ is bounded, then whenever x is close to both P and Q', it must also be close to their intersection. If, for example, $D \geq 2$ and P and Q are balls of radius one whose centers are separated by distance exactly two, then κ is unbounded. The maximum $\kappa_* = \sup_{x \in (P \cup Q') \backslash E} \kappa(x)$ is intimately related to the convergence rate.

Theorem 10. *Let P and Q be convex sets, and suppose that $\kappa_* < \infty$. Then AP between P and Q converges linearly with rate $1 - \frac{1}{\kappa_*^2}$. Specifically,*

$$\|p_k - p_*\| \leq 2\|p_0 - p_*\|(1 - \tfrac{1}{\kappa_*^2})^k \quad \text{and} \quad \|q_k - q_*\| \leq 2\|q_0 - q_*\|(1 - \tfrac{1}{\kappa_*^2})^k.$$

We prove Theorem 10 in Section 6.6.

Relating κ_* to the Angles Between Faces of the Polyhedra

In this section, we consider the case of polyhedra P and Q, and we bound κ_* in terms of the angles between pairs of their faces. In Lemma 11, we show that κ is nondecreasing along the sequence of points generated by AP between P and Q'. We treat points p for which $\kappa(p) = 1$ separately because those are the points from which AP between P and Q' converges in one step. This lemma enables us to bound $\kappa(p)$ by initializing AP at p and bounding κ at some later point in the resulting sequence.

Lemma 11. *For any $p \in P \backslash E$, either $\kappa(p) = 1$ or $1 < \kappa(p) \leq \kappa(\Pi_{Q'} p)$. Similarly, for any $q \in Q' \backslash E$, either $\kappa(q) = 1$ or $1 < \kappa(q) \leq \kappa(\Pi_P q)$.*

We prove Lemma 11 in Section 6.6. We can now bound κ by the angles between the faces of P and Q.

Proposition 12. *If P and Q are polyhedra and $p \in P \backslash E$, then there exist faces P_x and Q_y such that*

$$1 - \frac{1}{\kappa(p)^2} \leq c_F(\text{aff}_0(P_x), \text{aff}_0(Q_y))^2.$$

The analogous statement holds when we replace $p \in P \backslash E$ with $q \in Q' \backslash E$.

Note that $\text{aff}_0(Q_y) = \text{aff}_0(Q'_y)$. We prove Proposition 12 in Section 6.6. Proposition 12 immediately gives us the following corollary.

Corollary 13. *If P and Q are polyhedra, then*

$$1 - \frac{1}{\kappa_*^2} \leq \max_{x,y \in \mathbb{R}^D} c_F(\text{aff}_0(P_x), \text{aff}_0(Q_y))^2.$$

Angles Between Subspaces and Singular Values

Corollary 13 leaves us with the task of bounding the Friedrichs angle. To do so, we first relate the Friedrichs angle to the singular values of certain matrices in Lemma 14. We then specialize this to base polyhedra of submodular functions. For convenience, we prove Lemma 14 in Section 6.6, though this result is implicit in the characterization of principal angles between subspaces given in [85, Section 1]. Ideas connecting angles between subspaces and eigenvalues are also used by [47].

Lemma 14. *Let S and T be matrices with orthonormal rows and with equal numbers of columns. If all of the singular values of $ST^\top$ equal one, then $c_F(\text{null}(S), \text{null}(T)) = 0$. Otherwise, $c_F(\text{null}(S), \text{null}(T))$ is equal to the largest singular value of $ST^\top$ that is less than one.*

Faces of relevant polyhedra. Let $\mathcal{A}_x$ and $\mathcal{B}_y$ be faces of the polyhedra $\mathcal{A}$ and $\mathcal{B}$ from Lemma 9. Since $\mathcal{A}$ is a vector space, its only nonempty face is $\mathcal{A}_x = \mathcal{A}$. Hence, $\mathcal{A}_x = \text{null}(S)$, where S is the $N \times NR$ matrix

$$S = \frac{1}{\sqrt{R}} \left(\underbrace{I_N \quad \cdots \quad I_N}_{\text{repeated } R \text{ times}} \right). \tag{6.5}$$

Here, I_N denotes the $N \times N$ identity matrix. The matrix for $\text{aff}_0(\mathcal{B}_y)$ requires a bit more elaboration. Since $\mathcal{B}$ is a Cartesian product, we have $\mathcal{B}_y = B(F_1)_{y_1} \times \cdots \times B(F_R)_{y_R}$, where $y = (y_1, \ldots, y_R)$ and $B(F_r)_{y_r}$ is a face of $B(F_r)$. To proceed, we use the following characterization of faces of base polytopes [11, Proposition 4.7].

Proposition 15. *Let F be a submodular function, and let $B(F)_x$ be a face of $B(F)$. Then there exists a partition of V into disjoint sets $A_1, \ldots, A_M$ such that*

$$\text{aff}(B(F)_x) = \bigcap_{m=1}^{M} \{s \in \mathbb{R}^N \mid s(A_1 \cup \cdots \cup A_m) = F(A_1 \cup \cdots \cup A_m)\}.$$

The following corollary is immediate.

Corollary 16. *Define F, $B(F)_x$, and $A_1, \ldots, A_M$ as in Proposition 15. Then*

$$\text{aff}_0(B(F)_x) = \bigcap_{m=1}^{M} \{s \in \mathbb{R}^N \mid s(A_1 \cup \cdots \cup A_m) = 0\}.$$

Corollary 16 implies that, for each F_r, there exists a partition of V into disjoint sets $A_{r1}, \ldots, A_{rM_r}$ such that

$$\text{aff}_0(\mathcal{B}_y) = \bigcap_{r=1}^{R} \bigcap_{m=1}^{M_r} \{(s_1, \ldots, s_R) \in \mathbb{R}^{NR} \mid s_r(A_{r1} \cup \cdots \cup A_{rm}) = 0\}. \tag{6.6}$$

In other words, we can write $\text{aff}_0(\mathcal{B}_y)$ as the nullspace of either of the matrices

$$T' = \begin{pmatrix} 1_{A_{11}}^\top & & & \\ \vdots & & & \\ 1_{A_{11} \cup \cdots \cup A_{1M_1}}^\top & & & \\ & \ddots & & \\ & & 1_{A_{R1}}^\top & \\ & & \vdots & \\ & & 1_{A_{R1} \cup \cdots \cup A_{RM_R}}^\top & \end{pmatrix} \text{ or } T = \begin{pmatrix} \frac{1_{A_{11}}^\top}{\sqrt{|A_{11}|}} & & & \\ \vdots & & & \\ \frac{1_{A_{1M_1}}^\top}{\sqrt{|A_{1M_1}|}} & & & \\ & \ddots & & \\ & & \frac{1_{A_{R1}}^\top}{\sqrt{|A_{R1}|}} & \\ & & \vdots & \\ & & \frac{1_{A_{RM_R}}^\top}{\sqrt{|A_{RM_R}|}} & \end{pmatrix},$$

where 1_A is the indicator vector of $A \subseteq V$. For T', this follows directly from Equation Equation 6.6. T can be obtained from T' via left multiplication by an invertible matrix, so T and T' have the same nullspace. Lemma 14 then implies that $c_F(\mathrm{aff}(\mathcal{A}_x), \mathrm{aff}_0(\mathcal{B}_y))$ equals the largest singular value of

$$ST^\top = \frac{1}{\sqrt{R}} \begin{pmatrix} \frac{1_{A_{11}}}{\sqrt{|A_{11}|}} & \cdots & \frac{1_{A_{1M_1}}}{\sqrt{|A_{1M_1}|}} & \cdots & \frac{1_{A_{R1}}}{\sqrt{|A_{R1}|}} & \cdots & \frac{1_{A_{RM_R}}}{\sqrt{|A_{RM_R}|}} \end{pmatrix}$$

that is less than one. We rephrase this conclusion in the following remark

Remark 17. *The largest eigenvalue of $(ST^\top)^\top(ST^\top)$ less than one equals $c_F(\mathrm{aff}(\mathcal{A}_x), \mathrm{aff}_0(\mathcal{B}_y))^2$.*

Let $M_{\text{all}} = M_1 + \cdots + M_R$. Then $(ST^\top)^\top(ST^\top)$ is the $M_{\text{all}} \times M_{\text{all}}$ square matrix whose rows and columns are indexed by (r, m) with $1 \leq r \leq R$ and $1 \leq m \leq M_r$ and whose entry corresponding to row (r_1, m_1) and column (r_2, m_2) equals

$$\frac{1}{R} \frac{1^\top_{A_{r_1 m_1}} 1_{A_{r_2 m_2}}}{\sqrt{|A_{r_1 m_1}||A_{r_2 m_2}|}} = \frac{1}{R} \frac{|A_{r_1 m_1} \cap A_{r_2 m_2}|}{\sqrt{|A_{r_1 m_1}||A_{r_2 m_2}|}}.$$

Bounding the Relevant Eigenvalues

It remains to bound the largest eigenvalue of $(ST^\top)^\top(ST^\top)$ that is less than one. To do so, we view the matrix as the symmetric normalized Laplacian of a particular weighted graph.

Let G be the graph whose vertices are indexed by (r, m) with $1 \leq r \leq R$ and $1 \leq m \leq M_r$. Let the edge between vertices (r_1, m_1) and (r_2, m_2) have weight $|A_{r_1 m_1} \cap A_{r_2 m_2}|$. We may assume that G is connected (the analysis in this case subsumes the analysis in the general case). The symmetric normalized Laplacian $\mathcal{L}$ of this graph is closely related to our matrix of interest,

$$(ST^\top)^\top(ST^\top) = I - \tfrac{R-1}{R}\mathcal{L}. \tag{6.7}$$

Hence, the largest eigenvalue of $(ST^\top)^\top(ST^\top)$ that is less than one can be determined from the smallest nonzero eigenvalue $\lambda_2(\mathcal{L})$ of $\mathcal{L}$. We bound $\lambda_2(\mathcal{L})$ via Cheeger's inequality (stated in Section 6.6) by bounding the Cheeger constant h_G of G.

Lemma 18. *For $R \geq 2$, we have $h_G \geq \frac{2}{NR}$ and hence $\lambda_2^2(\mathcal{L}) \geq \frac{2}{N^2R^2}$.*

We prove Lemma 18 in Section 6.6. Combining Remark 17, Equation Equation 6.7, and Lemma 18, we obtain the following bound on the Friedrichs angle.

Proposition 19. *Assuming that $R \geq 2$, we have*

$$c_F(\mathrm{aff}(\mathcal{A}_x), \mathrm{aff}_0(\mathcal{B}_y))^2 \leq 1 - \tfrac{R-1}{R}\tfrac{2}{N^2R^2} \leq 1 - \tfrac{1}{N^2R^2}.$$

Together with Theorem 10 and Corollary 13, Proposition 19 implies the final bound on the rate.

Theorem 20. *The AP algorithm for Problem Equation P4 converges linearly with rate $1 - \frac{1}{N^2R^2}$.*

6.4 A Lower Bound

To probe the tightness of Theorem 20, we construct a submodular function and a decomposition that leads to a slow rate. We make use of Lemma 27, which describes the exact rate in the subspace case, and Corollary 28, which allows us to reduce our example to the subspace case. Both results are shown in Section 6.7. For our worst-case example below, the empirical convergence rate matches the theoretical lower bound of Theorem 22 (Section 6.7).

For each $x, y \in V$, define the submodular function G_{xy} on V to be the cut function on the graph with the single edge (x, y)

$$G_{xy} = \begin{cases} 1 & \text{if } |A \cap \{x, y\}| = 1 \\ 0 & \text{otherwise .} \end{cases}$$

Now, let N be even and $R \geq 2$ and define the submodular function $F^{\text{lb}} = F_1^{\text{lb}} + \cdots + F_R^{\text{lb}}$, where

$$F_1^{\text{lb}} = G_{12} + G_{34} + \cdots + G_{(N-1)N} \qquad F_2^{\text{lb}} = G_{23} + G_{45} + \cdots + G_{N1}$$

and $F_r^{\text{lb}} = 0$ for all $r \geq 3$. If we restrict the support of G_{xy} to $\{x, y\}$, then $B(G_{xy}) = \{(s, -s) \,|\, s \in [-1, 1]\}$. Therefore, the product $\mathcal{B}$ of base polytopes for this problem, which we refer to as $\mathcal{B}^{\text{lb}}$, is

$$\mathcal{B}^{\text{lb}} = \{(\underbrace{s_1, -s_1, \ldots, s_{\frac{N}{2}}, -s_{\frac{N}{2}}}_{B(F_1^{\text{lb}})}, \underbrace{-t_{\frac{N}{2}}, t_1, -t_1, \ldots, t_{\frac{N}{2}}}_{B(F_2^{\text{lb}})}, \underbrace{0, \ldots, 0}_{B(F_3^{\text{lb}})}, \ldots, \underbrace{0, \ldots, 0}_{B(F_R^{\text{lb}})}) \,|\, s_i, t_j \in [-1, 1]\}.$$

In Figure 6.3, we illustrate several runs of AP between $\mathcal{A}$ and $\mathcal{B}^{\text{lb}}$.

Lemma 21. *The Friedrichs angle between $\mathcal{A}$ and $\text{aff}(\mathcal{B}^{lb})$ satisfies*

$$c_F(\mathcal{A}, \text{aff}(\mathcal{B}^{lb}))^2 = 1 - \tfrac{1}{R}\left(1 - \cos\left(\tfrac{2\pi}{N}\right)\right).$$

Lemma 21, proved in Section 6.7, leads to the following lower bound on the worst-case rate.

Theorem 22. *The worst-case convergence rate of AP between the polyhedra $\mathcal{A}$ and $\mathcal{B}$ from Lemma 9 is at least $1 - \frac{2\pi^2}{N^2 R}$.*

Proof. From Corollary 28 and Lemma 21, we see that there is some nonzero $a_0 \in \mathcal{A}$ such that the sequences $\{a_k\}_{k\geq 0}$ and $\{b_k\}_{k\geq 0}$ generated by AP between $\mathcal{A}$ and $\mathcal{B}^{\text{lb}}$ satisfy

$$\|a_k\| = (1 - \tfrac{1}{R}(1 - \cos(\tfrac{2\pi}{N})))^k \|a_0\| \quad \text{and} \quad \|b_k\| = (1 - \tfrac{1}{R}(1 - \cos(\tfrac{2\pi}{N})))^k \|b_0\|.$$

Lastly, the inequality $1 - \cos(x) \leq \frac{1}{2}x^2$ gives the lower bound. □

6.5 Convergence of the Primal Objective

We have shown that AP produces a sequence of points $\{a_k\}_{k\geq 0}$ and $\{b_k\}_{k\geq 0}$ in $\mathbb{R}^{NR}$ such that $(a_k, b_k) \to (a_*, b_*)$ linearly, where (a_*, b_*) minimizes the objective in Problem Equation P4. In this section, we show that this result also implies the linear convergence of the objective in Problem Equation P3 and of the original discrete objective in Problem Equation P1.

Define the matrix $\Gamma = -R^{1/2}S$, where S is the matrix defined in Equation Equation 6.5. Multiplication by Γ maps a vector $(w_1, \ldots, w_R)$ to $-\sum_r w_r$, where $w_r \in \mathbb{R}^N$ for each r. Set $x_k = \Gamma b_k$ and $x_* = \Gamma b_*$. As shown in [81], Problem Equation P3 is minimized by x_*.

Proposition 23. *We have $f(x_k) + \frac{1}{2}\|x_k\|^2 \to f(x_*) + \frac{1}{2}\|x_*\|^2$ linearly with rate $1 - \frac{1}{N^2R^2}$.*

We prove Proposition 23 in Section 6.8. This linear rate of convergence translates into the following linear rate for the original discrete problem. We prove Theorem 24 in Section 6.8.

Theorem 24. *Choose $A_* \in \arg\min_{A\subseteq V} F(A)$. Let A_k be the suplevel set of x_k with smallest value of F. Then $F(A_k) \to F(A_*)$ linearly with rate $1 - \frac{1}{2N^2R^2}$.*

6.6 Upper Bound Results

Proof of Theorem 10

For the proof of this theorem, we will need the fact that projection maps are firmly nonexpansive, that is, for a closed convex nonempty subset $C \subseteq \mathbb{R}^D$, we have

$$\|\Pi_C x - \Pi_C y\|^2 + \|(x - \Pi_C x) - (y - \Pi_C y)\|^2 \leq \|x - y\|^2$$

for all $x, y \in \mathbb{R}^D$. Now, suppose that $\kappa_* < \infty$. Let $e = \Pi_E p_k$ and note that $v = \Pi_Q e - e$ and that $\Pi_Q e \in H$. We have

$$\begin{aligned}
\kappa_*^{-2} d(p_k, E)^2 &\leq d(p_k, Q')^2 \\
&\leq \|p_k - \Pi_Q p_k + v\|^2 \\
&\leq \|(p_k - \Pi_Q p_k) - (e - \Pi_Q e)\|^2 \\
&\leq \|p_k - e\|^2 - \|\Pi_Q p_k - \Pi_Q e\|^2 \\
&\leq d(p_k, E)^2 - d(q_k, H)^2.
\end{aligned}$$

It follows that $d(q_k, H) \leq (1 - \kappa_*^{-2})^{1/2} d(p_k, E)$. Similarly, we have $d(p_{k+1}, E) \leq (1 - \kappa_*^{-2})^{1/2} d(q_k, H)$. Combining these and inducting, we see that

$$\begin{aligned}
d(p_k, E) &\leq (1 - \kappa_*^{-2})^k d(p_0, E) \\
d(q_k, H) &\leq (1 - \kappa_*^{-2})^k d(q_0, H).
\end{aligned}$$

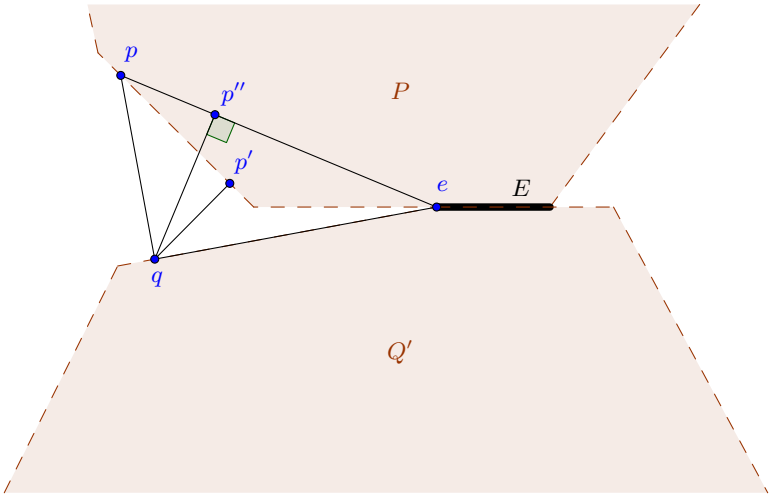


Figure 6.2: Illustration of the proof of Lemma 11.

As shown in [17, Theorem 3.3], the above implies that $p_k \to p_* \in E$ and $q_k \to q_* \in H$ and that

$$\|p_k - p_*\| \leq 2\|p_0 - p_*\|(1 - \kappa_*^{-2})^k$$
$$\|q_k - q_*\| \leq 2\|q_0 - q_*\|(1 - \kappa_*^{-2})^k.$$

Connection Between κ and c_F in the Subspace Case

In this section, we introduce a simple lemma connecting κ and c_F in the case of subspaces U and V. We will use this lemma in several subsequent proofs.

Lemma 25. *Let U and V be subspaces and suppose $u \in U \cap (U \cap V)^\perp$ and that $u \neq 0$. Then*

(a) $\|\Pi_V u\| \leq c_F(U, V)\|u\|$
(b) $\kappa(u) \leq (1 - c_F(U, V)^2)^{-1/2}$
(c) $\kappa(u) = (1 - c_F(U, V)^2)^{-1/2}$ *if and only if* $\|\Pi_V u\| = c_F(U, V)\|u\|$.

Proof. Part (a) follows from the definition of c_F. Indeed,

$$c_F(U, V) \geq \frac{u^\top(\Pi_V u)}{\|u\|\|\Pi_V u\|} = \frac{\|\Pi_V u\|^2}{\|u\|\|\Pi_V u\|} = \frac{\|\Pi_V u\|}{\|u\|}.$$

Part (b) follows from Part (a) and the observation that $\kappa(u) = (1 - \|\Pi_V u\|^2/\|u\|^2)^{-1/2}$. Part (c) follows from the same observation. □

Proof of Lemma 11

It suffices to prove the statement for $p \in P \backslash E$. For $p \in P \backslash E$, define $q = \Pi_{Q'} p$, $e = \Pi_E q$, and $p'' = \Pi_{[p,e]} q$, where $[p, e]$ denotes the line segment between p and e (which is contained in P by convexity). See Figure 6.2 for a graphical depiction. If $q \in E$, then $\kappa(p) = 1$. So we may assume that $q \notin E$ which also implies that $d(p'', E) > 0$ and $d(\Pi_P q, E) > 0$. We have

$$\kappa(p) = \frac{d(p, E)}{d(p, Q')} \leq \frac{\|p - e\|}{\|p - q\|} \leq \frac{\|q - e\|}{\|q - p''\|} \leq \frac{d(q, E)}{d(q, P)} = \kappa(q). \tag{6.8}$$

The first inequality holds because $d(p, E) \leq \|p - e\|$ and $d(p, Q') = \|p - q\|$. The middle inequality holds because the area of the triangle with vertices p, q, and e can be expressed as both $\frac{1}{2}\|p - e\|\|q - p''\|$ and $\frac{1}{2}\|p - q\|\|q - e\| \sin\theta$, where θ is the angle between vectors $p - q$ and $e - q$, so

$$\|p - e\|\|q - p''\| = \|p - q\|\|q - e\| \sin\theta \leq \|p - q\|\|q - e\|.$$

The third inequality holds because $\|q - e\| = d(q, E)$ and $\|q - p''\| \geq d(q, P)$. The chain of inequalities in Equation Equation 6.8 prove the lemma.

Proof of Proposition 12

Suppose that $p \in P \backslash E$ (the case $q \in Q' \backslash E$ is the same), and let $e = \Pi_E p$. If $\kappa(p) = 1$, the statement is evident, so we may assume that $\kappa(p) > 1$. We will construct sequences of polyhedra

$$\begin{array}{ccccccc} P & \supseteq & P_1 & \supseteq & \cdots & \supseteq & P_J \\ Q' & \supseteq & Q'_1 & \supseteq & \cdots & \supseteq & Q'_J \end{array},$$

where P_{j+1} is a face of P_j and Q'_{j+1} is a face of Q'_j for $1 \leq j \leq J-1$. Either $\dim(\text{aff}(P_{j+1})) < \dim(\text{aff}(P_j))$ or $\dim(\text{aff}(Q'_{j+1})) < \dim(\text{aff}(Q'_j))$ will hold. We will further define $E_j = P_j \cap Q'_j$, which will contain e, so that we can define

$$\kappa_j(x) = \frac{d(x, E_j)}{\max\{d(x, P_j), d(x, Q'_j)\}}$$

for $x \in \mathbb{R}^D \backslash E_j$ (this is just the function κ defined for the polyhedra P_j and Q'_j). Our construction will yield points $p_j \in P_j$, and $q_j \in Q'_j$ such that $p_j \in \text{relint}(P_j) \backslash E_j$, $q_j \in \text{relint}(Q'_j) \backslash E_j$, and $q_j = \Pi_{Q'_j} p_j$ for each j. Furthermore, we will have

$$\kappa(p) \leq \kappa_1(p_1) \leq \cdots \leq \kappa_J(p_J). \tag{6.9}$$

Now we describe the construction. For any $t \in [0, 1]$, define $p^t = (1 - t)p + te$ to be the point obtained by moving p by the appropriate amount toward e. Note that $t \mapsto \kappa(p^t)$ is a nondecreasing function on the interval $[0, 1)$. Choose $\epsilon > 0$ sufficiently small so that every

face of either P or Q' that intersects $B_\epsilon(e)$, the ball of radius ϵ centered on e, necessarily contains e. Now choose $0 \leq t_0 < 1$ sufficiently close to 1 so that $\|p^{t_0} - e\| < \epsilon$. It follows that e is contained in the face of P whose relative interior contains p^{t_0}. It further follows that e is contained in the face of Q' whose relative interior contains $\Pi_{Q'} p^{t_0}$ because

$$\|\Pi_{Q'} p^{t_0} - e\| = \|\Pi_{Q'} p^{t_0} - \Pi_{Q'} e\| \leq \|p^{t_0} - e\| < \epsilon.$$

To initialize the construction, set

$$\begin{aligned} p_1 &= p^{t_0} \\ q_1 &= \Pi_{Q'} p^{t_0}, \end{aligned}$$

and let P_1 and Q'_1 be the unique faces of P and Q' respectively such that $p_1 \in \text{relint}(P_1)$ and $q_1 \in \text{relint}(Q'_1)$ (the relative interiors of the faces of a polyhedron partition that polyhedron [27, Theorem 2.2]). Note that $q_1 \notin E$ because $\kappa(p_1) \geq \kappa(p) > 1$. Note that $e \in E_1 = P_1 \cap Q'_1$ so that

$$\kappa(p) \leq \kappa(p_1) = \frac{d(p_1, E)}{d(p_1, Q')} = \frac{\|p_1 - e\|}{\|p_1 - q_1\|} = \frac{d(p_1, E_1)}{d(p_1, Q'_1)} = \kappa_1(p_1).$$

Now, inductively assume that we have defined P_j, Q'_j, p_j, and q_j satisfying the stated properties. Generate the sequences $\{x_k\}_{k\geq 0}$ and $\{y_k\}_{k\geq 0}$ with $x_k \in P_j$ and $y_k \in Q'_j$ by running AP between the polyhedra P_j and Q'_j initialized with $x_0 = p_j$. There are two possibilities, either $x_k \in \text{relint}(P_j)$ and $y_k \in \text{relint}(Q'_j)$ for every k, or there is some k for which either $x_k \notin \text{relint}(P_j)$ or $y_k \notin \text{relint}(Q'_j)$. Note that AP will not terminate after a finite number of steps.

Suppose that $x_k \in \text{relint}(P_j)$ and $y_k \in \text{relint}(Q'_j)$ for every k. Then set $J = j$ and terminate the procedure. Otherwise, choose k' such that either $x_{k'} \notin \text{relint}(P_j)$ or $y_{k'} \notin \text{relint}(Q'_j)$. Now set $p_{j+1} = x_{k'}$ and $q_{j+1} = y_{k'}$. Let P_{j+1} and Q'_{j+1} be the unique faces of P_j and Q'_j respectively such that $p_{j+1} \in \text{relint}(P_{j+1})$ and $q_{j+1} \in \text{relint}(Q'_{j+1})$. Note that $p_{j+1}, q_{j+1} \notin E_{j+1} = P_{j+1} \cap Q'_{j+1}$ and $e \in E_{j+1}$. We have

$$\kappa_j(p_j) < \kappa_j(p_{j+1}) = \frac{d(p_{j+1}, E_j)}{d(p_{j+1}, Q'_j)} = \frac{d(p_{j+1}, E_j)}{\|p_{j+1} - q_{j+1}\|} \leq \frac{d(p_{j+1}, E_{j+1})}{d(p_{j+1}, Q'_{j+1})} = \kappa_{j+1}(p_{j+1}).$$

The preceding work shows the inductive step. Note that if $P_{j+1} \neq P_j$ then $\dim(\text{aff}(P_{j+1})) < \dim(\text{aff}(P_j))$ and if $Q'_{j+1} \neq Q'_j$ then $\dim(\text{aff}(Q'_{j+1})) < \dim(\text{aff}(Q'_j))$. One of these will hold, so the induction will terminate after a finite number of steps.

We have produced the sequence in Equation Equation 6.9 and we have created p_J, P_J, and Q'_J such that AP between P_J and Q'_J, when initialized at p_J, generates the same sequence of points as AP between $\text{aff}(P_J)$ and $\text{aff}(Q'_J)$. Using this fact, along with [46, Theorem 9.3], we see that $\Pi_{\text{aff}(P_J) \cap \text{aff}(Q'_J)} p_J \in E_J$. Using this, along with Lemma 25(b), we see that

$$\kappa_J(p_J) \leq (1 - c_F(\text{aff}_0(P_J), \text{aff}_0(Q'_J))^2)^{-1/2}. \tag{6.10}$$

Equations Equation 6.10 and Equation 6.9 prove the result. Note that P_J and Q'_J are faces of P and Q' respectively. We can switch between faces of Q' and faces of Q because doing so amounts to translating by v which does not affect the angles.

Proof of Lemma 14

We have

$$\begin{aligned} c_F(\text{null}(S), \text{null}(T)) &= c_F(\text{range}(S^\top)^\perp, \text{range}(T^\top)^\perp) \\ &= c_F(\text{range}(S^\top), \text{range}(T^\top)), \end{aligned}$$

where the first equality uses the fact that $\text{null}(W) = \text{range}(W^\top)^\perp$ for matrices W, and the second equality uses the fact that $c_F(U^\perp, V^\perp) = c_F(U, V)$ for subspaces U and V [14, Fact 2.3].

Let $S^\top$ and $T^\top$ have dimensions $D \times J$ and $D \times K$ respectively, and let X and Y be the subspaces spanned by the columns of $S^\top$ and $T^\top$ respectively. Without loss of generality, assume that $J \leq K$. Let $\sigma_1 \geq \cdots \geq \sigma_J$ be the singular values of $ST^\top$ with corresponding left singular vectors $u_1, \ldots, u_J$ and right singular vectors $v_1, \ldots, v_J$. Let $x_j = S^\top u_j$ and let $y_j = T^\top v_j$ for $1 \leq j \leq J$. By definition, we can write

$$\sigma_j = \max_{u,v} \{u^\top ST^\top v \,|\, u \perp \text{span}(u_1, \ldots, u_{j-1}), v \perp \text{span}(v_1, \ldots, v_{j-1}), \|u\| = 1, \|v\| = 1\}.$$

Since the $\{u_j\}_j$ are orthonormal, so are the $\{x_j\}_j$. Similarly, since the $\{v_j\}_j$ are orthonormal, so are the $\{y_j\}_j$. Suppose that all of the singular values of $ST^\top$ equal one. Then we must have $x_j = y_j$ for each j, which implies that $X \subseteq Y$, and so $c_F(X, Y) = 0$.

Now suppose that $\sigma_1 = \cdots = \sigma_\ell = 1$, and $\sigma_{\ell+1} \neq 1$. It follows that

$$X \cap Y = \text{span}(x_1, \ldots, x_\ell) = \text{span}(y_1, \ldots, y_\ell),$$

and so

$$\begin{aligned} \sigma_{\ell+1} &= \sup_{u,v} \{u^\top ST^\top v \,|\, u \in \text{span}(u_1, \ldots, u_\ell)^\perp, v \in \text{span}(v_1, \ldots, v_\ell)^\perp, \|u\| = 1, \|v\| = 1\} \\ &= \sup_{x,y} \{x^\top y \,|\, x \in X \cap (X \cap Y)^\perp, y \in Y \cap (X \cap Y)^\perp, \|x\| = 1, \|y\| = 1\} \\ &= c_F(X, Y). \end{aligned}$$

Cheeger's Inequality

For an overview of spectral graph theory, see [33]. We state Cheeger's inequality below.

Let G be a weighted, connected graph with vertex set V_G and edge weights $(w_{ij})_{i,j \in V_G}$. Define the weighted degree of a vertex i to be $\delta_i = \sum_{j \neq i} w_{ij}$, define the volume of a subset of vertices to be the sum of their weighted degrees, $\text{vol}(U) = \sum_{i \in U} \delta_i$, and define the size of the cut between U and its complement U^c to be the sum of the weights of the edges between U and U^c,

$$|E(U, U^c)| = \sum_{i \in U, j \in U^c} w_{ij}.$$

The Cheeger constant is defined as

$$h_G = \min_{\emptyset \neq U \subsetneq V_G} \frac{|E(U, U^c)|}{\min(\text{vol}(U), \text{vol}(U^c))}.$$

Let L be the unnormalized Laplacian of G, i.e. the $|V_G| \times |V_G|$ matrix whose entries are defined by

$$L_{ij} = \begin{cases} -w_{ij} & i \neq j \\ \delta_i & \text{otherwise} \end{cases}.$$

Let D be the $|V_G| \times |V_G|$ diagonal matrix defined by $D_{ii} = \delta_i$. Then $\mathcal{L} = D^{-1/2} L D^{-1/2}$ is the normalized Laplacian. Let $\lambda_2(\mathcal{L})$ denote the second smallest eigenvalue of $\mathcal{L}$ (since G is connected, there will be exactly one eigenvalue equal to zero).

Theorem 26 (Cheeger's inequality)**.** *We have* $\lambda_2(\mathcal{L}) \geq \frac{h_G^2}{2}$.

Proof of Lemma 18

Proof. We have

$$\begin{aligned} \min(\text{vol}(U), \text{vol}(U^c)) &\leq \frac{1}{2} \text{vol}(V_G) \\ &= \frac{1}{2} \sum_{(r,m)} \left(\sum_{(r',m') \neq (r,m)} |A_{rm} \cap A_{r'm'}| \right) \\ &= \frac{1}{2} \sum_{(r,m)} (R-1)|A_{rm}| \\ &= \frac{1}{2} NR(R-1). \end{aligned}$$

Since G is connected, for any nonempty set $U \subsetneq V_G$, there must be some element $v \in V$ (here V is the ground set of our submodular function F, not the set of vertices V_G) such that $v \in A_{r_1 m_1} \cap A_{r_2 m_2}$ for some $(r_1, m_1) \in U$ and $(r_2, m_2) \in U^c$. Suppose that v appears in k of the subsets of V indexed by elements of U and in $R - k$ of the subsets of V indexed by elements of U^c. Then

$$|E(U, U^c)| \geq k(R-k) \geq R - 1.$$

It follows that

$$h_G \geq \frac{R-1}{\frac{1}{2} NR(R-1)} = \frac{2}{NR}.$$

□

It follows from Theorem 26 that $\lambda_2(\mathcal{L}) \geq \frac{2}{N^2 R^2}$.

6.7 Results for the Lower Bound

Some Helpful Results

In Lemma 27, we show how AP between subspaces U and V can be initialized to exactly achieve the worst-case rate of convergence. Then in Corollary 28, we show that if subsets U' and V' look like subspaces U and V near the origin, we can initialize AP between U' and V' to achieve the same worst-case rate of convergence.

Lemma 27. *Let U and V be subspaces with $U \not\subseteq V$ and $V \not\subseteq U$. Then there exists some nonzero point $u_0 \in U \cap (U \cap V)^\perp$ such that when we initialize AP at u_0, the resulting sequences $\{u_k\}_{k\geq 0}$ and $\{v_k\}_{k\geq 0}$ satisfy*

$$\|u_k\| = c_F(U,V)^{2k}\|u_0\|$$
$$\|v_k\| = c_F(U,V)^{2k}\|v_0\|.$$

Proof. Find $u_* \in U \cap (U \cap V)^\perp$ and $v_* \in V \cap (U \cap V)^\perp$ with $\|u_*\| = 1$ and $\|v_*\| = 1$ such that $u_*^\top v_* = c_F(U,V)$, which we can do by compactness. By Lemma 25(a),

$$c_F(U,V) = v_*^\top u_* = v_*^\top \Pi_V u_* \leq \|\Pi_V u_*\| \leq c_F(U,V).$$

Set $u_0 = u_*$ and generate the sequences $\{u_k\}_{k\geq 0}$ and $\{v_k\}_{k\geq 0}$ via AP. Since $\|\Pi_V u_0\| = c_F(U,V)$, Lemma 25(c) implies that $\kappa(u_0) = (1-c_F(U,V)^2)^{-1/2}$. Since κ attains its maximum at u_0, Lemma 11 implies that κ attains the same value at every element of the sequences $\{u_k\}_{k\geq 0}$ and $\{v_k\}_{k\geq 0}$. Therefore, Lemma 25(c) implies that $\|\Pi_V u_k\| = c_F(U,V)\|u_k\|$ and $\|\Pi_U v_k\| = c_F(U,V)\|v_k\|$ for all k. This proves the lemma. □

Corollary 28. *let U and V be subspaces with $U \not\subseteq V$ and $V \not\subseteq U$. Let $U' \subseteq U$ and $V' \subseteq V$ be subsets such that $U' \cap B_\epsilon(0) = U \cap B_\epsilon(0)$ and $V' \cap B_\epsilon(0) = V \cap B_\epsilon(0)$ for some $\epsilon > 0$. Then there is a point $u_0' \in U'$ such that the sequences $\{u_k'\}_{k\geq 0}$ and $\{v_k'\}_{k\geq 0}$ generated by AP between U' and V' initialized at u_0' satisfy*

$$\|u_k'\| = c_F(U,V)^{2k}\|u_0'\|$$
$$\|v_k'\| = c_F(U,V)^{2k}\|v_0'\|.$$

Proof. Use Lemma 27 to choose some nonzero $u_0 \in U \cap (U \cap V)^\perp$ satisfying this property. Then set $u_0' = \frac{\epsilon}{\|u_0\|}u_0$. □

Proof of Lemma 21

Observe that we can write

$$\text{aff}(\mathcal{B}^{\text{lb}}) = \{(s_1, -s_1, \ldots, s_{\frac{N}{2}}, -s_{\frac{N}{2}}, -t_{\frac{N}{2}}, t_1, -t_1, \ldots, t_{\frac{N}{2}}, 0, \ldots, 0, \ldots, 0, \ldots, 0) \mid s_i, t_j \in \mathbb{R}\}.$$

We can write $\text{aff}(\mathcal{B}^{\text{lb}})$ as the nullspace of the matrix

$$T_{\text{lb}} = \begin{pmatrix} T_{\text{lb},1} & & & & \\ & T_{\text{lb},2} & & & \\ & & I_N & & \\ & & & \ddots & \\ & & & & I_N \end{pmatrix},$$

where the $N \times N$ identity matrix I_N is repeated $R-2$ times and where $T_{\text{lb},1}$ and $T_{\text{lb},2}$ are the $\frac{N}{2} \times N$ matrices

$$T_{\text{lb},1} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & & & & \\ & & 1 & 1 & & \\ & & & & \ddots & \\ & & & & 1 & 1 \end{pmatrix} \qquad T_{\text{lb},2} = \frac{1}{\sqrt{2}} \begin{pmatrix} & 1 & 1 & & & \\ & & & 1 & 1 & \\ & & & & & \ddots \\ 1 & & & & & 1 \end{pmatrix}.$$

Recall that we can write $\mathcal{A}$ as the nullspace of the matrix S defined in Equation Equation 6.5. It follows from Lemma 14 that $c_F(\mathcal{A}, \text{aff}(\mathcal{B}^{\text{lb}}))$ equals the largest singular value of $ST_{\text{lb}}^\top$ that is less than one. We have

$$ST_{\text{lb}}^\top = \tfrac{1}{\sqrt{R}} \begin{pmatrix} T_{\text{lb},1}^\top & T_{\text{lb},2}^\top & I_N & \cdots & I_N \end{pmatrix}.$$

We can permute the columns of $ST_{\text{lb}}^\top$ without changing the singular values, so $c_F(\mathcal{A}, \text{aff}(\mathcal{B}^{\text{lb}}))$ equals the largest singular value of

$$\tfrac{1}{\sqrt{R}} \begin{pmatrix} T_{\text{lb},0}^\top & I_N & \cdots & I_N \end{pmatrix},$$

that is less than one, where $T_{\text{lb},0}$ is the $N \times N$ circulant matrix

$$T_{\text{lb},0} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & & \ddots & \\ & & & 1 & 1 \\ 1 & & & & 1 \end{pmatrix}.$$

Therefore, $c_F(\mathcal{A}, \text{aff}(\mathcal{B}^{\text{lb}}))^2$ equals the largest eigenvalue of

$$\tfrac{1}{R} \begin{pmatrix} T_{\text{lb},0}^\top & I_N & \cdots & I_N \end{pmatrix} \begin{pmatrix} T_{\text{lb},0}^\top & I_N & \cdots & I_N \end{pmatrix}^\top = \tfrac{1}{R} \left(T_{\text{lb},0}^\top T_{\text{lb},0} + (R-2) I_N \right)$$

that is less than one. Therefore, it suffices to examine the $N \times N$ circulant matrix

$$T_{\text{lb},0}^\top T_{\text{lb},0} = \frac{1}{2} \begin{pmatrix} 2 & 1 & & & 1 \\ 1 & 2 & & & \\ & & \ddots & & \\ & & & 2 & 1 \\ 1 & & & 1 & 2 \end{pmatrix}.$$

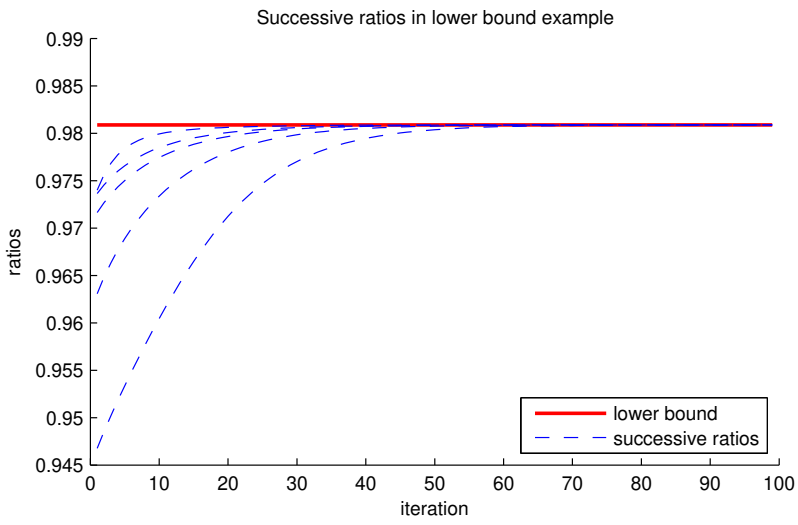


Figure 6.3: We run five trials of AP between $\mathcal{A}$ and $\mathcal{B}^{\text{lb}}$ with random initializations, where $N = 10$ and $R = 10$. For each trial, we plot the ratios $d(a_{k+1}, E)/d(a_k, E)$, where $E = \mathcal{A} \cap \mathcal{B}^{\text{lb}}$ is the optimal set. The red line shows the theoretical lower bound of $1 - \frac{1}{R}(1 - \cos(\frac{2\pi}{N})$ on the worst-case rate of convergence.

The eigenvalues of $T_{\text{lb},0}^{\top} T_{\text{lb},0}$ are given by $\lambda_j = 1 + \cos\left(\frac{2\pi j}{N}\right)$ for $0 \leq j \leq N - 1$ (see [67, Section 3.1] for a derivation). Therefore,

$$c_F(\mathcal{A}, \text{aff}(\mathcal{B}^{\text{lb}}))^2 = 1 - \tfrac{1}{R}(1 - \cos(\tfrac{2\pi}{N})).$$

Lower Bound Illustration

The proof of Theorem 22 shows that there is some $a_0 \in \mathcal{A}$ such that when we initialize AP between $\mathcal{A}$ and $\mathcal{B}^{\text{lb}}$ at a_0, we generate a sequence $\{a_k\}_{k \geq 0}$ satisfying

$$d(a_k, E) = (1 - \tfrac{1}{R}(1 - \cos(\tfrac{2\pi}{N}))^k d(a_0, E),$$

where $E = \mathcal{A} \cap \mathcal{B}^{\text{lb}}$ is the optimal set. In Figure 6.3, we plot the theoretical bound in red, and in blue the successive ratios $d(a_{k+1}, E)/d(a_k, E)$ for five runs of AP between $\mathcal{A}$ and $\mathcal{B}^{\text{lb}}$ with random initializations. Had we initialized AP at a_0, the successive ratios would exactly equal $1 - \frac{1}{R}(1 - \cos(\frac{2\pi}{N}))$. The plot of these ratios would coincide with the red line in Figure 6.3.

Figure 6.3 illustrates that the empirical behavior of AP between $\mathcal{A}$ and $\mathcal{B}^{\text{lb}}$ is often similar to the worst-case behavior, even when the initialization is random. When we initialize AP randomly, the successive ratios appear to increase to the lower bound and then remain constant. Figure 6.3 shows the case $N = 10$ and $R = 10$, but the plot looks similar for other N and R.

We also note that the graph corresponding to our lower bound example actually achieves a Cheeger constant similar to the one used in Lemma 18.

6.8 Results for Convergence of the Primal and Discrete Problems

Proof of Proposition 23

First, suppose that $s \in B(F)$. Let $A = \{n \in V \mid s_n \geq 0\}$ be the set of indices on which s is nonnegative. Then we have

$$\|s\| \leq \|s\|_1 = 2s(A) - s(V) \leq 3F_{\max}. \tag{6.11}$$

Now, we show that $f(x_k)+\frac{1}{2}\|x_k\|^2$ converges to $f(x_*)+\frac{1}{2}\|x_*\|^2$ linearly with rate $1-\frac{1}{N^2R^2}$. We will use Equation Equation 6.11 to bound the norms of x_k and x_*, both of which lie in $-B(F)$. We will also use the fact that $\|x_k - x_*\| \leq \|\Gamma\|\|b_k - b_*\| \leq \sqrt{R}\|b_k - b_*\|$. Finally, we will use the proof of Theorem 20 to bound $\|b_k - b_*\|$. First, we bound the difference between the squared norms using convexity. We have

$$\begin{aligned}
\tfrac{1}{2}\|x_k\|^2 - \tfrac{1}{2}\|x_*\|^2 &\leq x_k^\top (x_k - x_*) \\
&\leq \|x_k\|\|x_k - x_*\| \\
&\leq 3F_{\max}\sqrt{R}\|b_k - b_*\| \\
&\leq 6F_{\max}\sqrt{R}\|b_0 - b_*\|(1 - \tfrac{1}{N^2R^2})^k.
\end{aligned} \tag{6.12}$$

Next, we bound the difference in Lovász extensions. Choose $s \in \arg\max_{s \in B(F)} s^\top x_k$. Then

$$\begin{aligned}
f(x_k) - f(x_*) &\leq s^\top (x_k - x_*) \\
&\leq \|s\|\|x_k - x_*\| \\
&\leq 3F_{\max}\sqrt{R}\|b_k - b_*\| \\
&\leq 6F_{\max}\sqrt{R}\|b_0 - b_*\|(1 - \tfrac{1}{N^2R^2})^k.
\end{aligned} \tag{6.13}$$

Combining the bounds Equation 6.12 and Equation 6.13, we find that

$$(f(x_k) + \tfrac{1}{2}\|x_k\|^2) - (f(x_*) + \tfrac{1}{2}\|x_*\|^2) \leq 12F_{\max}\sqrt{R}\|b_0 - b_*\|(1 - \tfrac{1}{N^2R^2})^k. \tag{6.14}$$

Proof of Theorem 24

By definition, A_k is the set of the form $\{n \in V \mid (x_k)_n \geq c\}$ for some constant c with smallest value of $F(\{n \in V \mid (x_k)_n \geq c\})$.

Let $(w_*, s_*) \in \mathbb{R}^N \times B(F)$ be a primal-dual optimal pair for the left-hand version of Problem Equation P3. The dual of this minimization problem is the projection problem

$\min_{s \in B(F)} \frac{1}{2}\|s\|^2$. From [11, Proposition 10.5], we see that

$$\begin{aligned} F(A_k) - F(A_*) &\leq F(A_k) - (s_*)_-(V) \\ &\leq \sqrt{\tfrac{N}{2}((f(x_k) + \tfrac{1}{2}\|x_k\|^2) - (f(x_*) + \tfrac{1}{2}\|x_*\|^2))} \\ &\leq \sqrt{6F_{\max}NR^{1/2}\|b_0 - b_*\|}\,(1 - \tfrac{1}{N^2R^2})^{k/2} \\ &\leq \sqrt{6F_{\max}NR^{1/2}\|b_0 - b_*\|}(1 - \tfrac{1}{2N^2R^2})^{k}, \end{aligned}$$

where the third inequality uses the proof of Proposition 23. The second inequality relies on [11, Proposition 10.5], which states that a duality gap of ϵ for the left-hand version of Problem Equation P3 turns into a duality gap of $\sqrt{N\epsilon/2}$ for the original discrete problem. If our algorithm converged with rate $\frac{1}{k}$, this would translate to a rate of $\frac{1}{\sqrt{k}}$ for the discrete problem. But fortunately, our algorithm converges linearly, and taking a square root preserves linear convergence.

Running times. Theorem 24 implies that the number of iterations required for an accuracy of ϵ is at most

$$2N^2R^2 \log\left(\frac{\sqrt{6F_{\max}NR^{1/2}\|b_0 - b_*\|}}{\epsilon}\right).$$

Each iteration involves minimizing each of the F_r separately.

6.9 Conclusion

In this work, we analyze projection methods for SFM and give upper and lower bounds on the linear rate of convergence. This means that the number of iterations required for an accuracy of ϵ is logarithmic in $1/\epsilon$, not linear as in previous work [137]. Our rate is uniform over all submodular functions. Moreover, our proof highlights how the number R of components and the facial structure of $\mathcal{B}$ affect the convergence rate. These insights may serve as guidelines when working with projection algorithms and aid in the analysis of special cases. For example, reducing R is often possible. Any collection of F_r that have disjoint support, such as the cut functions corresponding to the rows or columns of a grid graph, can be grouped together as one component without making the projection harder.

Our analysis also shows the effects of additional properties of F. For example, suppose that F is *separable*, that is, $F(V) = F(S) + F(V \backslash S)$ for some nonempty $S \subsetneq V$. Then the subsets $A_{rm} \subseteq V$ defining the relevant faces of $\mathcal{B}$ satisfy either $A_{rm} \subseteq S$ or $A_{rm} \subseteq S^c$ [11]. This makes G in Section 6.3 disconnected, and as a result, the N in Theorem 20 gets replaced by $\max\{|S|, |S^c|\}$ for an improved rate. This applies without the user needing to know S when running the algorithm.

A number of future directions suggest themselves. We addressed AP, but [81] also considered the related Douglas-Rachford (DR) algorithm. DR between subspaces converges

linearly with rate c_F [14], as opposed to c_F^2 for AP. We suspect that our approach may be modified to analyze DR between polyhedra. Further questions include the extension to multiple polyhedra.

Chapter 7

Conclusion

In this thesis, we have described the challenges presented by modern machine learning applications and presented a distributed system capable of supporting these applications. Because machine learning applications themselves are so broad in scope, extending far beyond a single computational pattern such as distributed training or prediction serving, a system capable of supporting machine learning applications must be capable of supporting a variety of workloads including stream processing, data analytics, parallel simulations, distributed training, reinforcement learning, and hyperparameter search.

Many systems introduce new high-level abstractions that can be used to develop distributed applications. For example, many bulk synchronous parallel systems introduce a dataset concept as the core abstraction. Many streaming systems introduce a stream concept as the core abstraction. Hyperparameter search systems may use a trial concept as the core abstraction. The generality of our proposed architecture arises from the choice to not introduce new abstractions. Instead, we find a way to take the abstractions from single-threaded programming, namely functions and classes, and provide analogs in the distributed setting. By keeping the concepts the same, our proposal aims to preserve the generality of single-threaded programming in the distributed setting. In addition, this strategy provides a natural bridge for running the same code in the two settings.

The idea of a single general-purpose distributed system holds great promise. Today's distributed systems typically operate in isolation from one another, except perhaps sharing resources through a cluster manager. Each system separately handles task scheduling, data transfer, and fault tolerance. As a consequence, distributed systems can be challenging to compose together because, as self-contained entities, they are typically not designed to expose clean interfaces. However, our proposal allows new distributed systems to be built not as standalone distributed systems but rather as libraries on top of a general-purpose system. These libraries, because they operate on top of the same underlying distributed system, can easily be used together. This ability to compose libraries can enable substantially greater performance and ease-of-development for applications that integrate many different distributed computational patterns together. These applications include many reinforcement learning applications, which combine training, simulation, and serving patterns together as

well as many online learning applications, which combine streaming, training, and serving patterns together.

Toward this end, an ecosystem has begun to form around our proposal. Sophisticated libraries for reinforcement learning [94], hyperparameter tuning [95], data analytics [123], and other modern AI applications [6] have been developed, and nascent libraries for distributed training, stream processing, and model serving are in progress. Early experience suggests that our proposal has the abstractions needed to support the broader machine learning ecosystem.

Future Work

Much remains to be done in order to realize this vision of a single general-purpose distributed system.

Stringent Heterogeneous Requirements

One challenge with a general system is that it must support the most stringent requirements of each application that runs on top of it. If a streaming application requires transparent fault tolerance but a model training application does not, then the system must provide transparent fault tolerance. Similarly, if a simulation-based workload requires a system throughput of millions of tasks per second, but a hyperparameter tuning application does not, then the system must support millions of tasks per second. Further challenges arise from the fact that individual applications may have heterogeneous requirements. For example, a streaming graph processing application may require transparent fault tolerance for the portion of the application that processes incoming data and builds up the underlying graph, but it may allow tasks that interactively query the graph to fail in the presence of machine failures. Providing a sufficiently rich API that allows applications to specify their heterogeneous requirements is an important and challenging goal.

Guiding Users Toward Best Practices

The more flexible a system is, the more ways applications will be able to implement the same computations. For example a communication primitive such as a ring allreduce between a set of actors can be implemented by having the actors directly invoke methods on one another, or it can be implemented by having an external driver process invoke all of the methods and orchestrate the computation. These choices have implications for the efficiency of the strategy as well as the efficiency of recovery in the event of a failure. Much work remains to be done in order to shepherd users toward best practices and performant implementations.

Composable Libraries for Distributed Applications

Building single-machine applications by composing and using many different single-machine libraries is commonplace. For example, a normal Python script may use libraries for numer-

ical linear algebra, standard data structures, data manipulation, string parsing, and more. Though it is still common to use many single-machine libraries when building distributed applications, it is uncommon to build distributed applications by composing many distributed libraries together. Part of the reason for this is that most distributed applications are built as standalone systems and not as libraries. However, we believe that by providing libraries for common distributed patterns, such as standard distributed data structures, parallel stream and data processing, machine learning training, prediction serving, scalable microservices, and web crawling, that are all easily usable within the same framework, we can enable developers to build distributed applications much more rapidly and with much less code.

Generalizing Serverless Computing

Today, building distributed applications involves reasoning about application logic and managing clusters of machines. Serverless computing is changing that. By automatically managing the details of launching clusters and scaling them up and down, serverless computing allows application developers to reason only about application logic. As the appeal of distributed computing broadens beyond experts, abstracting away the underlying cluster configurations will become more and more important. However, serverless computing today provides a limited programming model and limited performance capabilities. The architecture proposed in this thesis may be well suited to provide the programming model for the next generation of serverless computing frameworks. Coupled together, this architecture and the insights of serverless computing could create a powerful paradigm for building cloud applications.

Bibliography

[1] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 265–283. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[2] Alekh Agarwal et al. "Making contextual decisions with low technical debt". In: *arXiv preprint arXiv:1606.03966* (2016).

[3] *Akka*. https://akka.io/.

[4] 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.

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

[6] Thomas Anthony et al. "Policy Gradient Search: Online Planning and Expert Iteration without Search Trees". In: *arXiv preprint arXiv:1904.03646* (2019).

[7] *Apache Arrow*. https://arrow.apache.org/. 2017.

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

[9] Ö. Aslan et al. "Convex Two-Layer Modeling". In: *Advances in Neural Information Processing Systems*. 2013, pp. 2985–2993.

[10] F. Bach. "Structured sparsity-inducing norms through submodular functions". In: *Advances in Neural Information Processing Systems*. 2011.

[11] Francis Bach. "Learning with Submodular Functions: A Convex Optimization Perspective". In: *Foundations and Trends in Machine Learning* 6.2-3 (2013), pp. 145–373.

[12] Peter Bailis et al. "Probabilistically bounded staleness for practical partial quorums". In: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 776–787.

[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] Heinz H. Bauschke et al. "The rate of linear convergence of the Douglas-Rachford algorithm for subspaces is the cosine of the Friedrichs angle". In: *Journal of Approximation Theory* 185.0 (2014), pp. 63–79.

[15] Heinz H Bauschke and Jonathan M Borwein. "Dykstra's alternating projection algorithm for two sets". In: *Journal of Approximation Theory* 79.3 (1994), pp. 418–443.

[16] Heinz H Bauschke and Jonathan M Borwein. "On Projection Algorithms for Solving Convex Feasibility Problems". In: *SIAM Review* 38.3 (1996), pp. 367–426.

[17] Heinz H Bauschke and Jonathan M Borwein. "On the convergence of von Neumann's alternating projection algorithm for two sets". In: *Set-Valued Analysis* 1.2 (1993), pp. 185–212.

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

[19] Amir Beck and Luba Tetruashvili. "On the Convergence of Block Coordinate Descent Type Methods". In: *SIAM Journal on Optimization* 23.4 (2013), pp. 2037–2060.

[20] B. Behmardi, C. Archambeau, and G. Bouchard. "Overlapping Trace Norms in Multi-View Learning". In: *arXiv preprint arXiv:1404.6163* (2014).

[21] J. M. Bioucas-Dias and M. A. T. Figueiredo. "Alternating direction algorithms for constrained sparse regression: application to hyperspectral unmixing". In: *Hyperspectral Image and Signal Processing: Evolution in Remote Sensing*. IEEE. 2010, pp. 1–4.

[22] S. Bird. "Optimizing Resource Allocations for Dynamic Interactive Applications". PhD thesis. University of California, Berkeley, 2014.

[23] 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.

[24] S. Boyd et al. "Distributed optimization and statistical learning via the alternating direction method of multipliers". In: *Foundations and Trends in Machine Learning* 3.1 (2011), pp. 1–122.

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

[26] Yaroslav Bulatov et al. "Flexible Primitives for Distributed Deep Learning in Ray". In: *SysML Conference*. 2018.

[27] James V Burke and Jorge J Moré. "On the Identification of Active Constraints". In: *SIAM Journal on Numerical Analysis* 25.5 (1988), pp. 1197–1211.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] Fan RK Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.

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

[35] 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.

[36] 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).

[37] D. Davis and W. Yin. "Convergence rate analysis of several splitting schemes". In: *arXiv preprint arXiv:1406.4834* (2014).

[38] D. Davis and W. Yin. "Convergence rates of relaxed Peaceman-Rachford and ADMM under regularity assumptions". In: *arXiv preprint arXiv:1407.5210* (2014).

[39] Jeffrey Dean et al. "Large scale distributed deep networks". In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

[40] 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.

[41] W. Deng and W. Yin. *On the global and linear convergence of the generalized alternating direction method of multipliers*. Tech. rep. DTIC Document, 2012.

[42] 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.

[43] Frank Deutsch. *Best Approximation in Inner Product Spaces*. Vol. 7. Springer, 2001.

[44] Frank Deutsch. "The method of alternating orthogonal projections". In: *Approximation Theory, Spline Functions and Applications*. Springer, 1992, pp. 105–121.

[45] Frank Deutsch and Hein Hundal. "The rate of convergence for the cyclic projections algorithm I: angles between convex sets". In: *Journal of Approximation Theory* 142.1 (2006), pp. 36–55.

[46] Frank Deutsch and Hein Hundal. "The rate of convergence of Dykstra's cyclic projections algorithm: The polyhedral case". In: *Numerical Functional Analysis and Optimization* 15.5-6 (1994), pp. 537–565.

[47] Persi Diaconis, Kshitij Khare, and Laurent Saloff-Coste. "Stochastic Alternating Projections". In: *Illinois Journal of Mathematics* 54.3 (2010), pp. 963–979.

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

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

[50] J. Eckstein. "Parallel alternating direction multiplier decomposition of convex programs". In: *Journal of Optimization Theory and Applications* 80.1 (1994), pp. 39–62.

[51] J. Eckstein and D. P. Bertsekas. "On the Douglas–Rachford splitting method and the proximal point algorithm for maximal monotone operators". In: *Mathematical Programming* 55.1-3 (1992), pp. 293–318.

[52] J. Eckstein and M. C. Ferris. "Operator-splitting methods for monotone affine variational inequalities, with a parallel application to optimal control". In: *INFORMS Journal on Computing* 10.2 (1998), pp. 218–235.

[53] J. Edmonds. "Combinatorial Structures and Their Applications". In: Gordon and Breach, 1970. Chap. Submodular Functions, Matroids and Certain Polyhedra, pp. 69–87.

[54] Alexander Fix et al. "Structured learning of sum-of-submodular higher order energy functions". In: *Int. Conference on Computer Vision (ICCV)*. 2013.

[55] P. A. Forero, A. Cano, and G. B. Giannakis. "Consensus-based distributed support vector machines". In: *The Journal of Machine Learning Research* 11 (2010), pp. 1663–1707.

[56] S. Forouzan and A. Ihler. "Linear Approximation to ADMM for MAP inference". In: *Asian Conference on Machine Learning*. 2013, pp. 48–61.

[57] D. Gabay and B. Mercier. "A dual algorithm for the solution of nonlinear variational problems via finite element approximation". In: *Computers & Mathematics with Applications* 2.1 (1976), pp. 17–40.

[58] 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, 2004, pp. 97–104.

[59] E. Ghadimi et al. "Optimal parameter selection for the alternating direction method of multipliers (ADMM): Quadratic problems". In: *arXiv preprint arXiv:1306.2454* (2014).

[60] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: 37.5 (2003), pp. 29–43.

[61] Andrew Gibiansky. *Bringing HPC Techniques to Deep Learning*. `http://research.baidu.com/bringing-hpc-techniques-deep-learning/`. 2017.

[62] P. Giselsson and S. Boyd. "Diagonal Scaling in Douglas–Rachford Splitting and ADMM". In: *IEEE Conference on Decision and Control*. 2014, pp. 5033–5039.

[63] R. Glowinski and A. Marroco. "Sur l'approximation, par éléments finis d'ordre un, et la résolution, par pénalisation-dualité d'une classe de problémes de Dirichlet non linéaires". In: *ESAIM: Mathematical Modelling and Numerical Analysis-Modélisation Mathématique et Analyse Numérique* 9.R2 (1975), pp. 41–76.

[64] T. Goldstein, B. O'Donoghue, and S. Setzer. "Fast alternating direction optimization methods". In: *CAM report* (2012), pp. 12–35.

[65] 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.

[66] Priya Goyal et al. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *arXiv preprint arXiv:1706.02677* (2017).

[67] Robert M Gray. "Toeplitz and Circulant Matrices: A Review". In: *Foundations and Trends in Communications and Information Theory* 2.3 (2006), pp. 155–239.

[68] Martin Grötschel, László Lovász, and Alexander Schrijver. "The Ellipsoid Method and its Consequences in Combinatorial Optimization". In: *Combinatorica* 1.2 (1981), pp. 169–197.

[69] 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.

[70] LG Gubin, BT Polyak, and EV Raik. "The method of projections for finding the common point of convex sets". In: *USSR Computational Mathematics and Mathematical Physics* 7.6 (1967), pp. 1–24.

[71] Israel Halperin. "The Product of Projection Operators". In: *Acta Sci. Math. (Szeged)* 23 (1962), pp. 96–99.

[72] 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.

[73] 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.

[74] Dorit Hochbaum and Vikas Singh. "An efficient algorithm for co-segmentation". In: *Int. Conference on Computer Vision (ICCV)*. 2009.

[75] M. Hong and Z.-Q. Luo. "On the linear convergence of the alternating direction method of multipliers". In: *arXiv preprint arXiv:1208.3922* (2012).

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

[77] 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.

[78] F. Iutzeler et al. "Linear Convergence Rate for Distributed Optimization with the Alternating Direction Method of Multipliers". In: *IEEE Conference on Decision and Control*. 2014, pp. 5046–5051.

[79] Satoru Iwata. "A Faster Scaling Algorithm for Minimizing Submodular Functions". In: *SIAM J. on Computing* 32 (2003), pp. 833–840.

[80] S. Jegelka, H. Lin, and J. Bilmes. "On fast approximate sumodular minimization". In: *Advances in Neural Information Processing Systems*. 2011.

[81] Stefanie Jegelka, Francis Bach, and Suvrit Sra. "Reflection methods for user-friendly submodular optimization". In: *Advances in Neural Information Processing Systems*. 2013, pp. 1313–1321.

[82] Rodolphe Jenatton et al. "Proximal methods for hierarchical sparse coding". In: *JMLR* (2011), pp. 2297–2334.

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

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

[85] Andrew V Knyazev and Merico E Argentati. "Principal angles between subspaces in an A-based scalar product: algorithms and perturbation estimates". In: *SIAM Journal on Scientific Computing* 23.6 (2002), pp. 2008–2040.

[86] P. Kohli, L. Ladický, and P. Torr. "Robust higher order potentials for enforcing label consistency". In: *Int. Journal of Computer Vision* 82 (2009).

[87] V. Kolmogorov. "Minimizing a sum of submodular functions". In: *Discrete Applied Mathematics* 160.15 (2012), pp. 2246–2258.

[88] N. Komodakis, N. Paragios, and G. Tziritas. "MRF energy minimization and beyond via dual decomposition". In: *IEEE Trans. Pattern Analysis and Machine Intelligence* (2011).

[89] A. Krause and S. Jegelka. *Submodularity in Machine Learning – New Directions*. ICML Tutorial. 2013.

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

[91] L. Lessard, B. Recht, and A. Packard. "Analysis and Design of Optimization Algorithms via Integral Quadratic Constraints". In: *SIAM Journal on Optimization* (2016).

[92] Mu Li et al. "Communication Efficient Distributed Machine Learning with the Parameter Server". In: *Advances in Neural Information Processing Systems*. 2014, pp. 19–27.

[93] 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.

[94] Eric Liang et al. "RLlib: Abstractions for Distributed Reinforcement Learning". In: *International Conference on Machine Learning 35*. 2018.

[95] Richard Liaw et al. "Tune: A research platform for distributed model selection and training". In: *arXiv preprint arXiv:1807.05118* (2018).

[96] H. Lin and J. Bilmes. "Optimal selection of limited vocabulary speech corpora". In: *Proc. Interspeech*. 2011.

[97] P.-L. Lions and B. Mercier. "Splitting algorithms for the sum of two nonlinear operators". In: *SIAM Journal on Numerical Analysis* 16.6 (1979), pp. 964–979.

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

[99] 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.

[100] 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.

[101] S.T. McCormick. "Handbook on Discrete Optimization". In: Elsevier, 2006. Chap. Submodular Function Minimization, pp. 321–391.

[102] O. Meshi and A. Globerson. "An alternating direction method for dual MAP LP relaxation". In: *Machine Learning and Knowledge Discovery in Databases*. Springer, 2011, pp. 470–483.

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

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

[105] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018. URL: `https://www.usenix.org/conference/osdi18/presentation/nishihara`.

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

[107] 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`.

[108] 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`.

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

[110] Mukund Narasimhan and Jeff Bilmes. "Local Search for Balanced Submodular Clusterings." In: *IJCAI*. 2007, pp. 981–986.

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

[112] Robert Nishihara, Stefanie Jegelka, and Michael I. Jordan. "On the Convergence Rate of Decomposable Submodular Function Minimization". In: *Advances in Neural Information Processing Systems 27*. 2014, pp. 640–648.

[113] Robert Nishihara et al. "A General Analysis of the Convergence of ADMM". In: *International Conference on Machine Learning 32*. 2015, pp. 343–352.

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

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

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

[117] James Orlin. "A faster strongly polynomial time algorithm for submodular function minimization". In: *Math. Programming* 118 (2009), pp. 237–251.

[118] 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.

[119] Xinghao Pan et al. "Revisiting distributed synchronous SGD". In: *arXiv preprint arXiv:1604.00981* (Feb. 2017). URL: http://arxiv.org/abs/1604.00981.

[120] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[121] P. Patrinos, L. Stella, and A. Bemporad. "Douglas–Rachford splitting: complexity estimates and accelerated variants". In: *IEEE Conference on Decision and Control*. 2014, pp. 4234–4239.

[122] P. Patrinos, L. Stella, and A. Bemporad. "Forward-backward truncated Newton methods for large-scale convex composite optimization". In: *arXiv preprint arXiv:1402.6655* (2014).

[123] Devin Petersohn and Anthony D Joseph. *Scaling Interactive Data Science Transparently with Modin*. 2018. URL: `https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-191.pdf`.

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

[125] 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.

[126] Matthew Rocklin. *Dask Benchmarks*. `http://matthewrocklin.com/blog/work/2017/07/03/scaling`. 2017.

[127] 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.

[128] B. Romera-Paredes and M. Pontil. "A New Convex Relaxation for Tensor Completion". In: *Advances in Neural Information Processing Systems*. 2013, pp. 2967–2975.

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

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

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

[132] 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.

[133] H. Sedghi, A. Anandkumar, and E. Jonckheere. "Multi-Step Stochastic ADMM in High Dimensions: Applications to Sparse Optimization and Matrix Decomposition". In: *Advances in Neural Information Processing Systems*. 2014, pp. 2771–2779.

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

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

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

[137] P. Stobbe and A. Krause. "Efficient Minimization of Decomposable Submodular Functions". In: *Advances in Neural Information Processing Systems*. 2010.

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

[139] *TensorFlow CIFAR10 Distributed Estimator*. `https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10_estimator`. 2017.

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

[141] 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.

[142] 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).

[143] 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.

[144] Paul Tseng. "Alternating Projection-Proximal methods for convex programming and variational inequalities". In: *SIAM Journal on Optimization* 7.4 (1997), pp. 951–965.

[145] 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.

[146] 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.

[147] Sara Vicente, Vladimir Kolmogorov, and Carsten Rother. "Joint optimization of segmentation and appearance models". In: *Int. Conference on Computer Vision (ICCV)*. 2009.

[148] John Von Neumann. *Functional Operators: The Geometry of Orthogonal Spaces*. Princeton University Press, 1950.

[149] B. Wahlberg et al. "An ADMM algorithm for a class of total variation regularized estimation problems". In: *IFAC Symposium on System Identification*. 2012, pp. 83–88.

[150] H. Wang and A. Banerjee. "Online Alternating Direction Method". In: *International Conference on Machine Learning*. 2012, pp. 1119–1126.

[151] H. Wang et al. "Large Scale Distributed Sparse Precision Estimation". In: *Advances in Neural Information Processing Systems*. 2013, pp. 584–592.

[152] 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.

[153] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012. ISBN: 1449311520, 9781449311520.

[154] Cong Xie, Oluwasanmi O. Koyejo, and Indranil Gupta. *Faster Distributed Synchronous SGD with Weak Synchronization*. 2018. URL: https://openreview.net/forum?id=H13WofbAb.

[155] 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.

[156] 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.

[157] C. Zhang, H. Lee, and K. G. Shin. "Efficient distributed linear classification algorithms via the alternating direction method of multipliers". In: *International Conference on Artificial Intelligence and Statistics*. 2012, pp. 1398–1406.

[158] R. Zhang and J. Kwok. "Asynchronous distributed ADMM for consensus optimization". In: *International Conference on Machine Learning*. 2014, pp. 1701–1709.