Flexible Primitives for Distributed Deep Learning in Ray

Yaroslav Bulatov*
DIUx

Robert Nishihara*
UC Berkeley

Philipp Moritz
UC Berkeley

Melih Elibol
UC Berkeley

Ion Stoica
UC Berkeley

Michael I. Jordan
UC Berkeley

ABSTRACT

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 Ray [12] 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 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 [3, 6]. 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.

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 [8] or parameter server approaches [11]. 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 [9]. In a setting with slow machines, stragglers may become a problem, and techniques like backup workers [14], asynchronous SGD [7] or the stale synchronous parameter server [10] can be used to address this bottleneck.

A more limited set of techniques have been proposed to address the problem of slow parameter servers [4].

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.

2 RAY PRIMITIVES

Ray [12] is a high-performance distributed execution framework targeted at supporting AI applications and machine learning in dynamic environments [13]. 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 [1] 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

*equal contribution

Ray is available at https://github.com/ray-project/ray.

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 [14] or partial pulling [4].

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.

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.

Vanilla Synchronous Parameter Server: In this basic scheme [11], 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 [14] 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 [7] 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 [5, 10], 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.

Partial Pulling: The partial pulling scheme [4] 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.

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

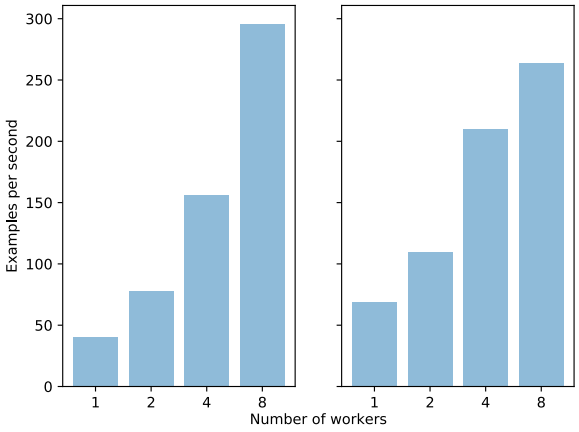

Figure 1: Left: Synchronous parameter server throughput for the pure TensorFlow implementation. Right: throughput of the Ray plus TensorFlow implementation. In both cases, the number of parameter servers is half the number of workers (rounded up).

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 EXPERIMENTS

As a proof of concept, we implemented the five aggregation strategies from Section 3 and integrated them with the TensorFlow CIFAR-10 Resnet implementation provided as part of the official distributed TensorFlow example [2].

In Figure 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 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 Appendix A, 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.

REFERENCES

[1] 2017. Apache Arrow. https://arrow.apache.org/. (2017).

[2] 2017. TensorFlow CIFAR10 Distributed Estimator. https://github.com/tensorflow/models/tree/master/tutorials/image/cifar10_estimator. (2017).

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *OSDI*, Vol. 16. 265–283.

[4] Anonymous. 2018. Faster Distributed Synchronous SGD with Weak Synchronization. *International Conference on Learning Representations* (2018). https://openreview.net/forum?id=H13WofbAb

[5] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. 2012. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment* 5, 8 (2012), 776–787.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.

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

[9] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677* (2017).

[10] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. 2013. More effective distributed ML via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*. 1223–1231.

[11] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server.. In *OSDI*, Vol. 1. 3.

[12] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *arXiv preprint arXiv:1712.05889* (2017).

[13] Robert Nishihara, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith, Richard Liaw, Mehrdad Niknami, Michael I. Jordan, and Ion Stoica. 2017. Real-Time Machine Learning: The Missing Pieces. In *Workshop on Hot Topics in Operating Systems*.

[14] Xinghao Pan, Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2017. Revisiting Distributed Synchronous SGD. *arXiv preprint arXiv:1702.05800* (2017).

A 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 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 [4] 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 2 and Figure 3.

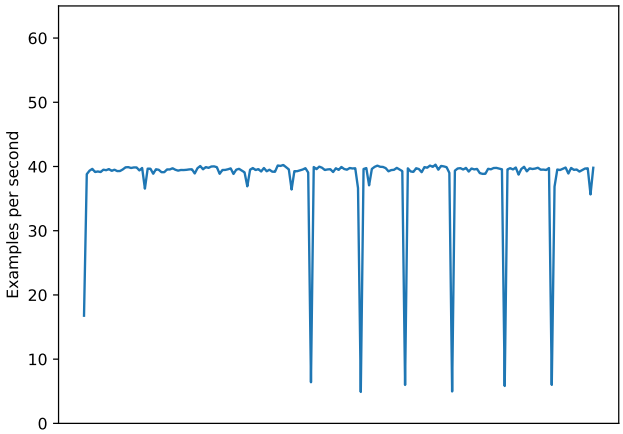

Figure 2: Training throughput of the pure TensorFlow asynchronous parameter server implementation in the presence of periodic slowdowns of a single parameter server shard. Throughput consistently decreases when one of the parameter servers slows down.

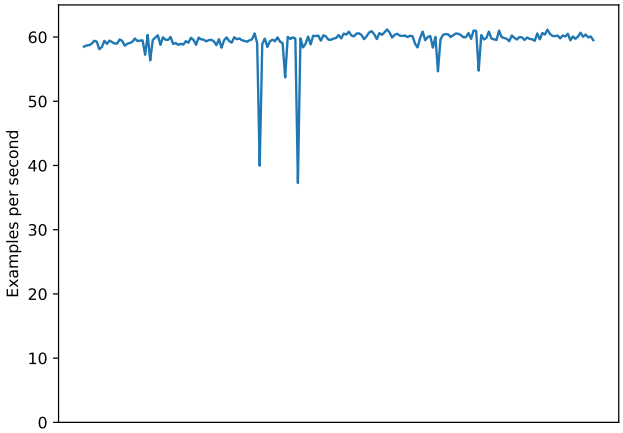

Figure 3: Training throughput of a Ray-based implementation of partial pulling in the presence of periodic slowdowns of a single parameter server shard. This particular aggregation strategy allows workers to avoid waiting for slow parameter servers and hence throughput suffers very little compared with the pure TensorFlow implementation.