Kubernetes for Distributed Machine Learning

Yi Wang Apr 22, 2016

Jike and Ian want me describe my experience of using Kubernetes with distributed machine learning. It is easy to explain how and what I do – install Kubernetes and Tensorflow and etc and use them. However, the more interesting part is why we use them, and how our daily work would look like after we use them.

Distributed Machine Learning

Students' Mode

Several months ago I began to learn deep learning. I started with programming Python on a CUDA laptop. This scale of parallel computing is small – a CPU and a GPU. But it seems that many people start with such simple setting. We all know that is not enough for solving real problems.

Research Lab Mode

From there, I started training models using the computing cluster of our team. The cluster was equipped with CUDA GPU towers connected with high speed network. Such settings is common in research labs.

To make full use of GPUs, the team wrote kernel programs specifically for the family of CUDA GPUs we are using. Also, the team implemented an extremely efficient MPI AllReduce operation, which covers details like handling communications between GPUs on the same node and between GPUs on different nodes.

This high performance system enabled fast iteration of research. However, it has some limitations as well.

Fast Algorithm v.s. Scalability and Utilization

We highly optimized the AllReduce operation, because the system runs synchronous SGD algorithm, where the effective mini-batch is the sum of those on each GPU. If we use more than 32 GPUs, the effective mini-batch size would be too big and it would take too long for the job to converge. This is a limitation of parallelism.

Another problem is low utilization. Though the cluster has many nodes, each job cannot use most of them. So we need to run many jobs in parallel to make full use of these nodes. But in practice, we don't always have so many jobs.

Fast I/O Speed v.s. Storage Convenience

To accelerate the training of DNN models, the systems loads data from local disks via high performance adaptors. This implies that we have to duplicate training data over all nodes. For a few times when we got more training data, we ran out of disk space. It is true that we can install more disks for each node, but there is always a limit there.

Specific-purpose Clusters v.s. General-purpose Clusters

Another problem was that when we preprocess new data, we ran out of disk space due to the large amount of intermediate results. We have to manually split the data into small chunks and process them one-by-one.

We did consider building a Hadoop cluster in addition to our GPU cluster, but we cannot build a new cluster every time we face a new kind of job – the maintenance cost would be super high and the overall utilization would be super low.

An even more head-scratching problem is, we are facing a new problem that doesn't fit into any existing framework. MapReduce, Spark, Storm, GraphLab, Petuum – none of them fits all algorithms.

Industrial Challenges

Unfortunately, all aforementioned challenges become real, when a research team grows to do more kinds of research or to serve real business:

- Infinite amount of data are streaming 24x365 days in the form of log messages generated by online Web services, and
- The team has to try many algorithms/solutions to make full use of data and to constantly improve service quality.
- There would be inevitably many kinds of jobs: online serving, online learning, offline data processing, and batch learning.
- To make full utilization of clusters, we need to run all these jobs on the same cluster, high priority ones serving products, low priority ones running experiments.

What is the solution to such a complexity?

Industrial Solutions

Distributed Filesystem

First of all, instead of duplicating data over all nodes, we need a distributed filesystem like Hadoop HDFS and Google GFS reliably limits data duplication.

For example, each chunk of data is duplicated on 3 or 5 nodes. Less duplication means lower data I/O, but it also provides bigger storage. Indeed, for high performance I/O, we can use caching services (e.g., memcached) or NoSQL database (Bigtable and Redis), which preload data into memory.

Cluster Management System

Then we need a common software platform, the distributed operating system or cluster management system, to run the jobs – machine learning and standard ones like memcached and Redis. Possible open source choices include YARN, Mesos and Kubernetes. Their designs are all influenced by Google Borg.

Both Borg and Kubernetes, the open sourced rewrite of Borg in Go, can run arbitrary programs such as nginx and memcached. Users usually write a job description file to pass in information like the number of preferred instances, the resource requirement of each instance, and the maximum trials before restarting failed workers.

Mesos runs only programs specifically written to fit its framework. In order to run Hadoop MapReduce or nginx, we have to write C++ adaptor programs, which, once launched by Mesos, start Hadoop and/or nginx. Similarly, YARN runs only programs following specific patterns.

Distributed Computing Frameworks

To ease the effort of writing distributed programs running on the cluster, it is a good idea to hide the common part that handles concurrency and communication in a library, and let users write only task specific part. These libraries are called distributed computing frameworks.

The machine learning community has long been acquainted with MapReduce, GraphLab and Spark. To show the business value, framework developers tend to show that many algorithms fit in their framework. But the reality is that there is a balance between generally applicability and runtime performance.

Google showed us a good example – each framework for a certain type of jobs. MapReduce for batch index building and other batch data processing, Pregel for efficient PageRank computing and graph computing, GBR for maximum entropy model training and other AllReduce-operation based algorithms, SETI specifically for click-through rate prediction and feature learning.

Borg and Kubernetes were designed to provide some neat APIs that helps engineers and researchers write their own frameworks easily and specifically tailored for their new algorithms, without requiring users knowing details shielded by the cluster management system.

Deployment, Versioning, Rollout/Rollback

A straight-forward and commonly-used way to deployment and run jobs is as follows:

1. build the program into a binary file,
2. make a tarball of the binary file and configuration file and other dependents like shared libraries,
3. scp the tarball to all nodes, and untar.
4. use MPI to start the program on some nodes.
5. remove the untar-ed files after the job finishes.

However, this procedure is usually painful.

It is inefficient to deploy and remove a program every time we run a job, because we might run a program multiple times. However, it is a hard manual labor to maintain a cache of deployed programs – consider that once a new version of the program is released, we need to make sure that newly started jobs run the new version, without killing existing jobs that run the old version. This is called *rollout*. A similar problem is *rollback* – if we find that the newly released version has a bug, we need to go back with the old version.

Dynamic and Elastic Scheduling

Consider the case that an experiment model training job is using all the 100 GPUs in the cluster. A production job gets started and asks for 50 GPUs. If we use MPI, we'd have to kill the experiment job so to release enough resource to run the production job. This tends to make the owner of the experiment job get the impression that he is doing a "second-class" job.

Kubernetes is smarter than MPI as it can kill only 50 workers of the experiment job, so to allow both jobs run at the same time. But this also requires that we change our program – we have to abandon the highly optimized AllReduce operation, because it would block if some workers are killed and cannot join this collaborative operation.

Remember the reason that we use AllReduce is because we use synchronous SGD. But in order to write a distributed DNN training program, we might have to use asynchronous SGD, even if asynchronous SGD is arguably slower than its synchronous counterpart.

With asynchronous SGD, workers don't talk with each other; instead, each worker occasionally talk with the parameter server. If some workers get preempted by higher priority jobs, other workers can go on with the training job. Once the preempted worker gets restarted, it can ask the parameter server for the current model and re-join the job. If the parameter server itself gets preempted, all workers can go on update their local model for a while without talking to the parameter server. Once the parameter server gets restarted, workers can resume the occasional update with the parameter server.

Auto Fault Recovery and Scalability

Above example introduces the concept of auto fault recovery, which is important if we want scalability.

In a general purpose cluster, there is always a chance that some workers get preempted by some higher priority jobs. The more workers in a job, and the longer the time needed by the job, the higher probability that some workers get preempted during the run. It is true that we can restart a job, but if the system is not auto recoverable, the restarted job has to start from the very beginning As a result, the jobs might never complete, because every time it's restarted, it is likely being preempted.

The key to auto fault recovery is to cut off the data dependencies between workers. With MPI, every worker can talk to any other worker at any time. Suppose that we want to restart a worker A and bring it back to the status right before it was preempted, we need to re-send to A all those messages it accepted during its previous life. That means we need to restart all those workers who talked to A. Such dependencies propagate and it is often that all workers need to be restarted, and the job restarts from the very beginning.

The MapReduce framework is well known for its capability of auto fault recovery. A MapReduce job contains three phases: map, shuffle, and reduce. Map and reduce workers are not allowed to communicate with each other, thus there is no data dependencies in map and reduce phases, once a worker gets restarted, it can continue the unfinished task or the next task. A limited form of data exchange happens in the shuffle stage, but is hidden from the user and taken care of by the MapReduce framework. MapReduce is efficient at fault recovery and is very scalable, but it is not flexible and might not fit algorithms that do need inter-worker communication.

Pregel, another Google framework, is in between MapReduce, an extreme for scalability, and MPI, the other extreme of flexibility. In Pregel, the computation is defined as a sequence of *super-steps*. Within each super-step, workers can communicate with each other. At the end of each super-step, the framework checkpoints all messages sent between workers during the super-step, so that if some workers get preempted, the job restarts and resume from the most recent checkpoint.

Isolation and Quota

Cutting off data dependencies is not easy. In above examples, we talked about eliminating network communications between workers. But there are other forms of data dependency. For example, via the filesystem:

Consider that a job A writes intermediate results to the `/tmp` directory, and it removes these intermediate results periodically by calling `rm /tmp/*`. Another job B writes checkpoints to `/tmp`. A might unintentionally remove checkpoints of

B. And it is very hard to detect such kind of bugs, since the error rarely appears again.

With Kubernetes, people have to build their programs into Docker *images* that run as Docker *containers*. Each container has its own filesystem and network port space. When A runs as a container, it removes only files in its own `/tmp` directory. This is to some extent like that we define C++ classes in namespaces, which helps us removing class name conflicts.

Put Together

The Cluster

If we use Kubernetes, the general picture of computing is illustrated as follows:

- The cluster has a distributed filesystem like HDFS, which logically organizes all disk space into a single big storage.
- We can run in-memory storage systems to preload the data from HDFS if we want high performance I/O.
- Kubernetes manages hardware resources, and monitors workers/jobs. A job is described by the number of workers and the amount of resource needed for each worker. Kubernetes starts workers on nodes that have enough idle resource.
- Kubernetes can run arbitrary programs, including those programs depending on frameworks. Kubernetes APIs make it easy for researchers to write their own framework if they want to.
- The design of algorithms and its parallel implementation is highly correlated. Recall the sync and asynchronous SGD example. So it is likely that a valuable algorithm needs specially tailored parallel computing framework.
- It is often **the case** that we need to change the algorithm so to control the data dependencies between workers. This would make it easier for us to design a fault recoverable and scalable framework for this algorithm.

An Example

A typical Kubernetes cluster runs an automatic speech recognition business might be running the following jobs:

1. The speech service, with as many instances so to serve many simultaneous user requests.
2. The Kafka system, whose each channel collects a certain log stream of the speech service.

3. Kafka channels are followed by Storm jobs for online data processing. For example, a Storm job joins the utterance log stream and transcription stream.
4. The joined result, namely session log stream, is fed to an ASR model trainer that updates the model.
5. This trainer notifies ASR server when it writes updated models into Ceph.
6. Researchers might change the training algorithm, and run some experiment training jobs, which serve testing ASR service jobs.

Daily Work

Kubernetes also changes our daily work. Suppose that we put our code on Github, every time we push a feature branch, Github should call CI system to test it, and build Docker images from the feature branch. The images are transferred to our Docker Hub server and is are runnable by Kubernetes for experiments. If the change in the feature branch is proven a good one, developers do code review and merge it into the master branch.