分布式机器学习笔记

Ruichen Wang

March 15, 2019

Abstract

—

Contents

1 基本知识

Communication Backends 主要有四种:

- TCP
  - 适用范围广，大部分系统和机器都提供支持。支持cpu上的p2p和collective functions。但是不支持GPU，优化程度也不高。
- MPI
  - 全称The Message Passing Interface，是高性能计算的一个标准工具。高度的并行化，在大规模集群上优化程度较高。pytorch使用MPI需要自己重新手动编译
- Gloo

  - 优化collective function, 同时支持gpu 与cpu .使用GPUDirect技术，在gpu间通信不需要通过cpu.

- NCCL
  - 全称The NVIDIA Collective Communications Library，主要针对Nvidia多机多卡做了对应优化.

Backend	gloo		mpi		nccl	
Device	**CPU**	**GPU**	**CPU**	**GPU**	**CPU**	**GPU**
send	✓	✘	✓	?	✘	✘
recv	✓	✘	✓	?	✘	✘
broadcast	✓	✓	✓	?	✘	✓
all_reduce	✓	✓	✓	?	✘	✓
reduce	✓	✘	✓	?	✘	✓
all_gather	✓	✘	✓	?	✘	✓
gather	✓	✘	✓	?	✘	✘
scatter	✓	✘	✓	?	✘	✘
barrier	✓	✘	✓	?	✘	✓

Figure 1: backends

参考：`https://pytorch.org/tutorials/intermediate/dist_tuto.html#our-own-ring-allreduce`

1.1 ps vs ringAllReduce ?

主流分布式机器学习采用的是ps架构。如tensorflow。ps全称Parameter Server Architecture 也就是参数服务器。
在Parameter server架构（PS架构）中，集群中的节点被分为两类：parameter server和worker。其中parameter server存放模型的参数，而worker负责计算参数的梯度。在每个迭代过程，worker从parameter sever中获得参数，然后将计算的梯度返回给parameter server，parameter server聚合从worker传回的梯度，然后更新参数，并将新的参数广播给worker。

pytorch 采用的是Uber Horovod的形式，也是baidu开源的ringAllReduce算法。采用PS计算模型的分布式，通常会遇到网络的问题，随着worker数量的增加，其加速比会迅速的恶化，例如resnet50这样的模型，目前的TF在10几台机器的时候，加速比已经开始恶化的不可接受了。因此，经常要上RDMA、InfiniBand等技术，并且还带来了一波网卡的升级，有些大厂直接上了100GBs的网卡，有

钱任性。而Uber的Horovod，采用的RingAllReduce的计算方案，其特点是网络通信量不随着worker（GPU）的增加而增加，是一个恒定值。简单看下图理解下，GPU 集群被组织成一个逻辑环，每个GPU有一个左邻居、一个右邻居，每个GPU只从左邻居接受数据、并发送数据给右邻居。即每次梯度每个gpu只获得部分梯度更新，等一个完整的Ring完成，每个GPU都获得了完整的参数。

2 pytorch distirbuted

pytorch 提供distribute包。

```python
def init_processes(rank, size, fn, backend='nccl'):
    """ Initialize the distributed environment. """
    os.environ['MASTER_ADDR'] = '127.0.0.1'
    os.environ['MASTER_PORT'] = '29500'
    dist.init_process_group(backend, rank=rank, world_size=size)
    fn(rank, size)


if __name__ == "__main__":
    size = 4
    processes = []
    for rank in range(size):
        p = Process(target=init_processes, args=(rank, size, run))
        p.start()
        processes.append(p)
```

Figure 2: backends

3 tensorflow distributed

tensorflow 提供tf.train.ClusterSpec来创建相关cluter集群。

```
cluster = tf.train.ClusterSpec({'ps': ps_spec, 'worker': worker_spec})
server = tf.train.Server(cluster, job_name=FLAGS.job_name, task_index=FLAGS.task_index)
if FLAGS.job_name == 'ps':
    server.join()

is_chief = (FLAGS.task_index == 0)
with tf.device(tf.train.replica_device_setter(
        cluster=cluster
)):
    global_step = tf.Variable(0, name='global_step', trainable=False)

    hid_w = tf.Variable(tf.truncated_normal([IMAGE_PIXELS * IMAGE_PIXELS, FLAGS.hidden_units],
                                            stddev=1.0 / IMAGE_PIXELS), name='hid_w')
    hid_b = tf.Variable(tf.zeros([FLAGS.hidden_units]), name='hid_b')

    sm_w = tf.Variable(tf.truncated_normal([FLAGS.hidden_units, 10],
                                           stddev=1.0 / math.sqrt(FLAGS.hidden_units)), name='sm_w')
    sm_b = tf.Variable(tf.zeros([10]), name='sm_b')

    x = tf.placeholder(tf.float32, [None, IMAGE_PIXELS * IMAGE_PIXELS])
    y_ = tf.placeholder(tf.float32, [None, 10])

    hid_lin = tf.nn.xw_plus_b(x, hid_w, hid_b)
    hid = tf.nn.relu(hid_lin)

    y = tf.nn.softmax(tf.nn.xw_plus_b(hid, sm_w, sm_b))
    cross_entropy = -tf.reduce_sum(y_ * tf.log(tf.clip_by_value(y, 1e-10, 1.0)))

    opt = tf.train.AdamOptimizer(FLAGS.learning_rate)
```

Figure 3: backends

分别在对应的ps,worker机器上执行相关脚本,等待所有节点加入后，训练开始

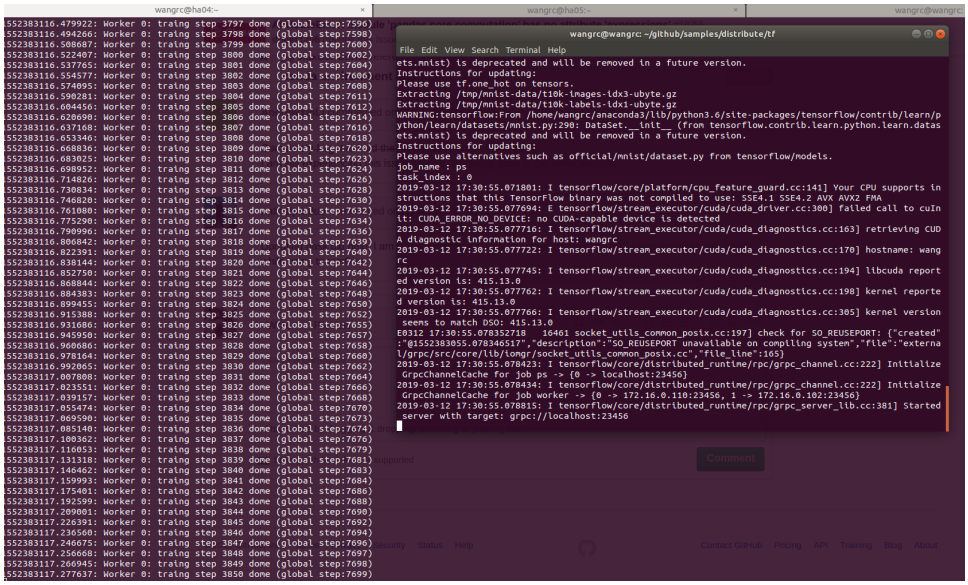

Figure 4: distribute training using tensorflow gpu

4 docker + tensorflow gpu/cpu

4.1 docker 是什么?

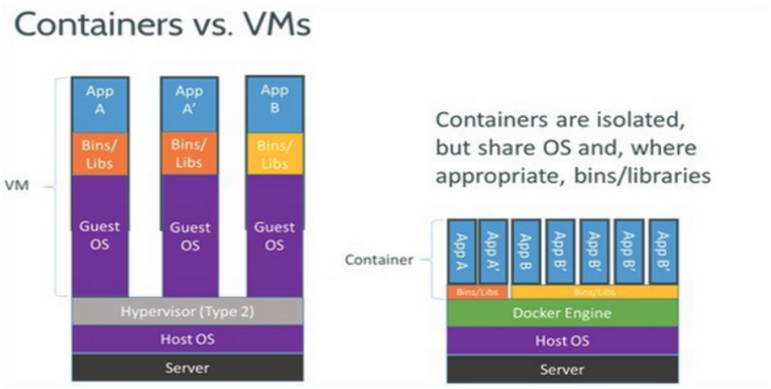

Figure 5: docker

4.2 安装docker

安装教程: `https://docs.docker.com/install/linux/docker-ce/ubuntu/`

- 查看镜像
  docker images
- 修改docker镜像源
  {
  "registry-mirrors": ["https://docker.mirrors.ustc.edu.cn"]
  }
- 安装tensorflow docker cpu, 可以使用-tag选择想要的tf版本和python版本
  docker pull tensorflow/tensorflow:1.12.0-py3
- 安装tensorflow docker gpu
  1. 首先安装nvidia-docker
  (https://github.com/NVIDIA/nvidia-docker)
  2. 确认nvidia docker 安装完成
  docker run –runtime=nvidia –rm nvidia/cuda nvidia-smi

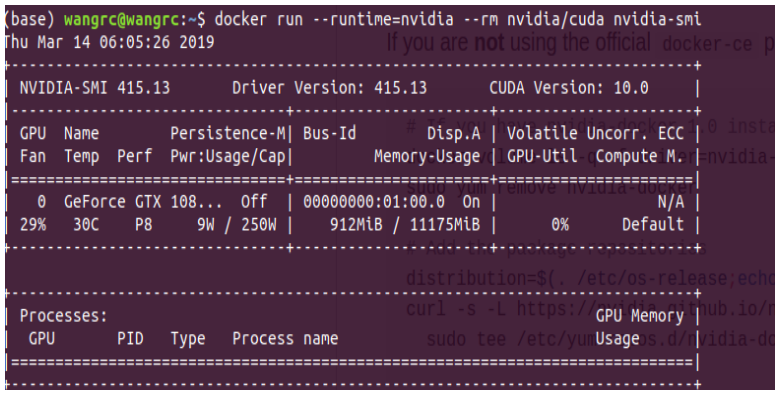

Figure 6: distribute training using tensorflow gpu

3. 安装tensorflow gpu
docker pull tensorflow/tensorflow:1.12.0-gpu-py3
4. 启动支持nvidia的docker
docker run –runtime=nvidia -it –rm tensorflow/tensorflow:1.12.0-gpu-py3 bash

Figure 7: training using docker tensorflow gpu

5 k8s

5.1 安装k8s

https://kubernetes.io/docs/setup/

5.2 简介

随着docker、容器的日渐成熟，容器编排的问题就凸显出来，大量的容器怎么去管理，怎么调度，怎么启停都成了迫切需要解决的问题。单纯地使

用docker没有办法限制cpu or gpu资源. k8s 支持为每个请求分配cpu与gpu资源. k8s优点这里就不详细列举了。

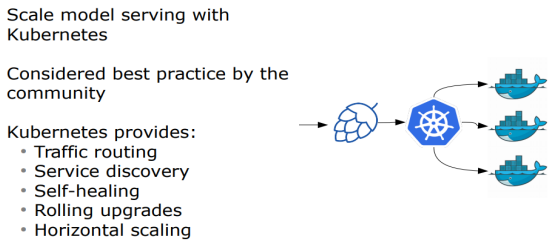

Figure 8: k8s and docker

	物理机	k8s
更改系统配置	需要每台节点修改	更改dockerfile，全部生效
增加节点	需要安装相同环境	修改pod数量即可
节点宕机	此节点上业务停止	自动在迁移至其他节点

Figure 9: k8s 2

典型的Kubernetes 集群包含一个master 和很多node。Master 是控制集群的中心，node 是提供CPU、内存和存储资源的节点。Master 上运行着多个进程，包括面向用户的API 服务、负责维护集群状态的Controller Manager、负责调度任务的Scheduler 等。每个node 上运行着维护node 状态并和master 通信的kubelet，以及实现集群网络服务的kube-proxy。

Kubernetes 中部署的最小单位是pod，而不是Docker 容器。实时上Kubernetes 是不依赖于Docker 的，完全可以使用其他的容器引擎在Kubernetes 管理的集群中替代Docker。在与Docker 结合使用时，一个pod 中可以包含一个或多个Docker 容器。但除了有紧密耦合的情况下，通常一个pod 中只有一个容器，这样方便不同的服务各自独立地扩展。

k8s 为pod分配gpu资源时可以指定gpu个数和类型:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: cuda-vector-add
spec:
  restartPolicy: OnFailure
  containers:
    - name: cuda-vector-add
      # https://github.com/kubernetes/kubernetes/blob/v1.7.11/test/images/nvidia-cuda/Dockerfile
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1
  nodeSelector:
    accelerator: nvidia-tesla-p100 # or nvidia-tesla-k80 etc.
```

Figure 10: k8s gpu schedule

整体架构:

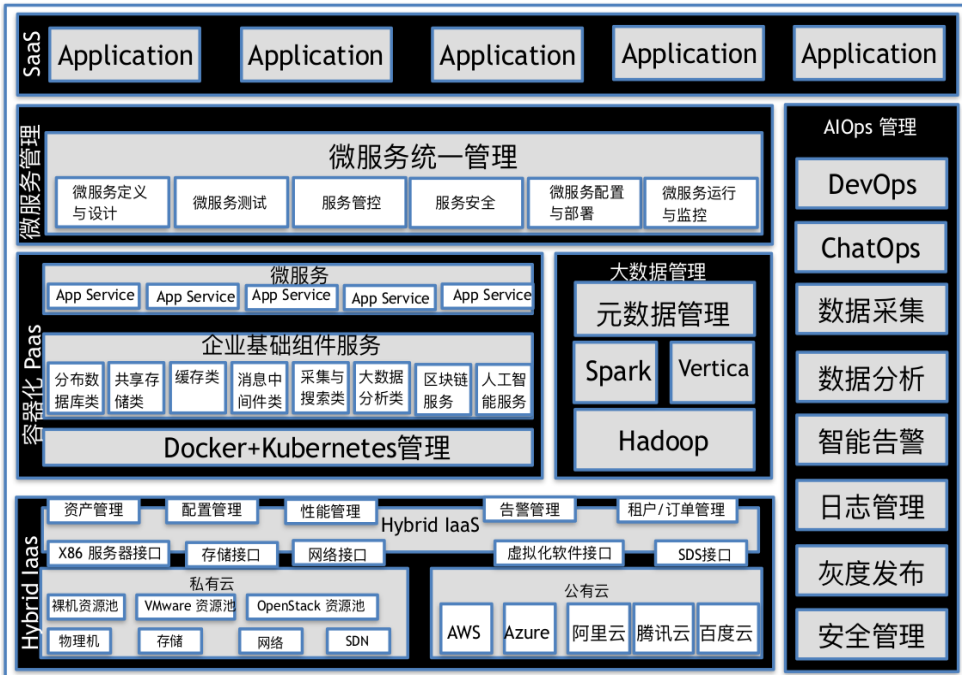

Figure 11: k8s+docker 整体架构

技术构架:

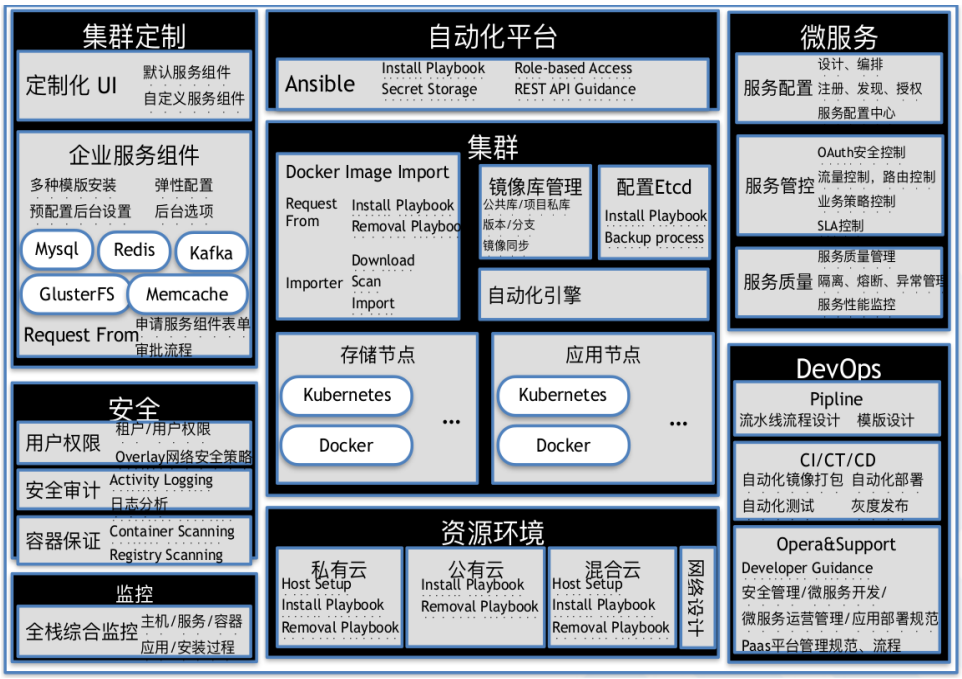

Figure 12: k8s+docker 技术架构