Ray使用文档

问题

1. ray的用法做个拆解，搞明白ray的基础原理和用法
2. 着重介绍下利用ray如何部署 vLLM？比如怎么启动ray，怎么启动多个vLLM实例，多个dp rank 怎么调用它

Ray是什么

Ray是一个开源分布式计算框架，为AI等上层应用提供了**并行处理的计算层**，大幅降低分布式工作流程的复杂性。

1. 可以跨多节点和GPU并行和分配ML的工作负载
2. 可以提供计算抽象（统一的API）进行ML系统的扩展集成
3. 自动处理关键流程，如编排，调度，容错，自动缩放等。
4. 大量的库都使用Ray作为并行执行的组件

Ray的基本原理

Ray Core

Task（任务）：

异步执行的Ray函数称作“任务”，也成为Ray远程函数。当一个新的Task被实例化时，会创建一个新的进程(worker)进行对其的调度。Ray使任务能够根据CPU，加速器等资源来指定其资源需求。

使用方式：

```
@ray.remote  # 通过该装饰器将函数改装为Ray远程函数，默认分配一个CPU
def fun(a):
    return a
obj_ref = fun.remote(a)  # 通过使用remote()方法激活函数，首先立即会返回一个结果的future对象，并创建一个异步任务开始执行
value = rey.get(obj_ref)  # 通过ray.get获得返回结果
```

同时，远程对象引用也可以当作参数传递：

```
@ray.remote
def function_with_an_argument(value):
    return value + 1

obj_ref2 = function_with_an_argument.remote(obj_ref)  # 将对象引用作为任务的参数
assert ray.get(obj_ref2) == 2
```

此时，第二个任务取决于第一个任务的输出，因此第二个任务要等待第一个任务执行完毕才开始。若调度在不同的设备上，则结果需要通过网络传输。

Actors（参与者）：

将函数扩展到类，参与者本质上是能存储状态的函数/服务。当一个新的Actor被实例化时，会创建一个新的进程进行对其的调度。同样，参与者支持CPU,加速器和自定义的资源请求。

使用方式

```
@ray.remote
class Counter:
    def __init__(self):
        self.value = 0

    def increment(self):
        time.sleep(1)
        self.value += 1
        print(f'counter: {self.value}')
        return self.value

    def get_counter(self):
        return self.value

c = Counter.remote()  # 实例化，创建一个worker，等待调度

for _ in range(10):
    # 调用类内函数的方式与执行Task一样，采用remote()激活
    # 采用异步执行方式，调度器提交的任务会发送至参与者的等待队列中，参与者依次执行等待队列中的任务
    print(f'scheduler: {_}')
    c.increment.remote()

print(ray.get(c.get_counter.remote()))
```

输出为：

```
scheduler: 0
scheduler: 1
scheduler: 2
scheduler: 3
scheduler: 4
scheduler: 5
scheduler: 6
scheduler: 7
scheduler: 8
scheduler: 9
(Counter pid=32160) counter: 1
(Counter pid=32160) counter: 2
(Counter pid=32160) counter: 3
(Counter pid=32160) counter: 4
(Counter pid=32160) counter: 5
(Counter pid=32160) counter: 6
(Counter pid=32160) counter: 7
(Counter pid=32160) counter: 8
(Counter pid=32160) counter: 9
(Counter pid=32160) counter: 10
10
```

Objects（对象）：

在Ray中，任务和参与者在对象上创建和计算，将这些对象成为远程对象。它们可以存储在Ray集群的任何位置，并通过对象引用（指针）来引用它们。

远程对象可以被缓存在Ray的分布式共享内存中，集群的每个节点都有一块共享内存，同时一个远程对象也可以存放在多个节点中。

创建对象引用的两种方式：

1. remote函数调用获取返回值的对象引用
2. `ray.put()`将某对象放置于Ray的共享内存上并返回在该共享内存上的远程对象引用

Env Dependency（依赖环境）：

每个节点需要相同的依赖环境，Ray提供两种方案：

1. 静态依赖时，在运行前使用Ray Cluster Launcher将依赖打包
2. 动态依赖时，使用Ray的运行时依赖环境，在运行时安装仅对Ray应用可见的包

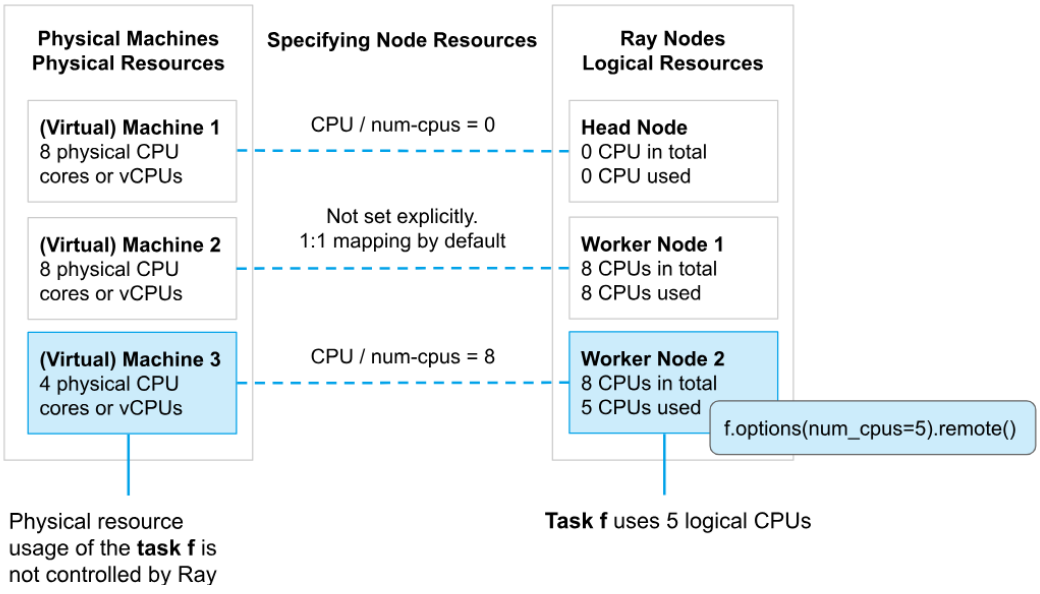
```
runtime_env = {"pip": ["emoji"]}

ray.init(runtime_env=runtime_env)
```

Scheduling（调度）：

对每个任务和参与者都有指定的资源需求。

资源：

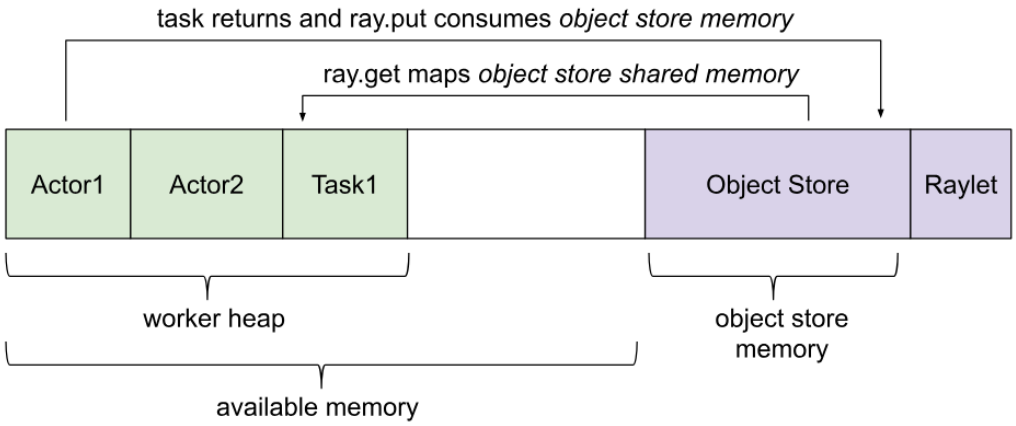
1. 节点的资源：
    1. 资源初始化：每个节点通过`ray.init(num_cpus=?,num_gpus=?,memory=?,custom_resource=?)`进行资源初始化
    2. 资源用键值对表示：键为资源名称，值为浮点数
    3. 使用逻辑资源抽象：不需要物理资源建立一对一映射
    4. 通常CPU的使用不会隔离，而GPU的使用会隔离



2. 节点的状态（针对需要资源的任务或参与者）：
    1. 可行：又分为可用（具有资源且此刻可使用）与不可用（具有资源但此刻被占用）

    2. 不可行：节点无法分配所需的资源
3. 加速器资源：支持GPU,TPU,NPU等等
    1. Ray内部可通过改变环境变量`CUDE_VISIBLE_DEVICE`在任务或参与者进程中只暴露分配给他们的GPU资源，也可人为在外部设置环境变量隐藏某些物理设备。
    2. 碎片化分配：分配加速器的数量可以是小数，使得多个任务可共享同一加速器
    3. 可强制指定加速器类型
4. 内存资源：
    1. Ray系统内存：用于存储每个节点信息，每个节点上的进程信息等内容
    2. 应用程序内存：

> 1. 工作堆
> 2. 对象存储内存：应用程序通过`ray.put`在对象存储创建对象以及从远程函数返回值时使用的内存
> 3. 对象存储共享内存：应用程序通过`ray.get`读取对象时使用的内存，通常节点上若已存储该对象，则无需额外的分配



调度策略：

决定可行节点中的最佳节点

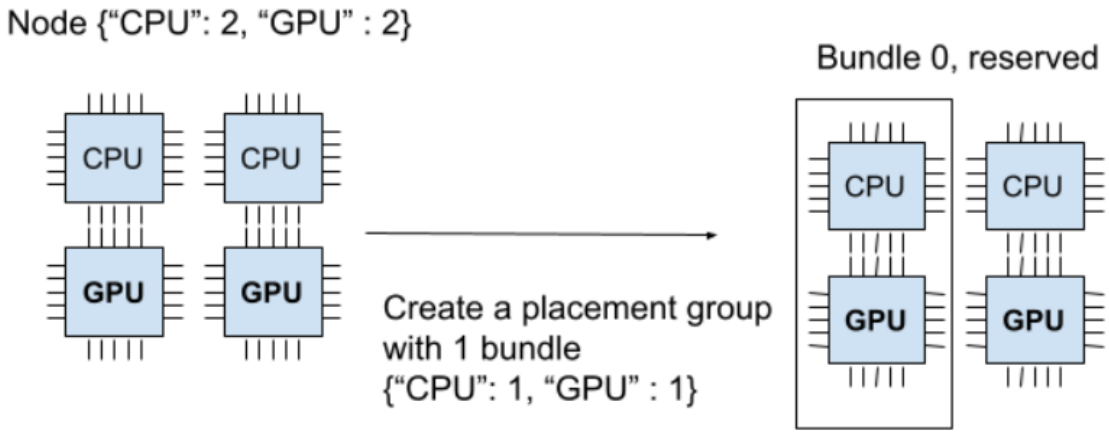
1. 默认：根据利用率由低至高排序取前k个节点，再随机选择。对于不需要任何资源（num_cpus=0）的任务，会随机选择一个节点。
2. Spread：将任务分散到全部节点。

Placement Groups（置放组）：

允许用户跨多个节点进行组调度，可用于安排任务和参与者，使其尽可能靠近本地（PACK）或分散（SPREAD）。

Buddle(捆绑包)：将一系列资源打包成捆绑包，作为预留资源，仅通过专门操作才能调度这些资源。

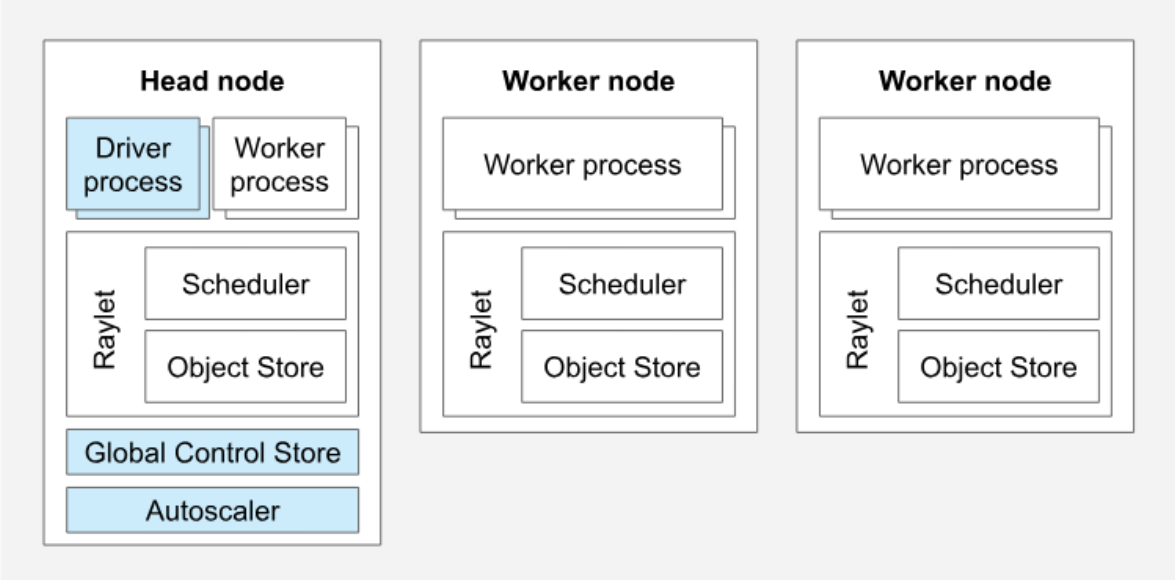
Placement Group(置放组)：一组捆绑包列表，根据集群节点的放置策略放置捆绑包



Ray集群

由单个头节点和任意数量的连接工作节点组成，**Ray中节点的最小单位是服务器**

1. 头节点：运行负责集群管理的单例进程，包括自动缩放器，GCS（全局控制服务），Ray驱动，其余功能与工作节点一致
    1. 自动缩放器（AutoScaling）：根据负载动态增加删除工作节点
    2. Ray驱动：接收提交的Ray作业，调度作业，将其分配至各工作节点执行。
2. 工作节点：运行任务或参与者的用户代码
3. 作业（Ray Jobs）：单个应用程序，源自同一脚本的任务，参与者等集合。各个节点都可以通过Ray Job API或Python脚本来运行作业。



集群建立

1. 使用Ray的配置文件，例如：

```
cluster_name: my_cluster
// 分为头节点和多个工作节点，其他进程或机器通过头节点连接进集群
head_node:
  node_ip_address: 192.168.0.1
  node_name: node1
worker_nodes:
  - node_ip_address: 192.168.0.2
    node_name: node2
  - node_ip_address: 192.168.0.3
    node_name: node3
```

2. 使用CLI： `ray start --head --port={port} --redis-password="{password}"`
3. 使用python脚本： `ray.init()`

集群连接（工作节点/主节点的其他进程 连接进主节点）

`ray.init(address="{ip}:{port}", redis_password={password})`

实战

采用Python API的方式，而非CLI。

启动Ray并部署vLLM

```python
import ray
from vllm import LLM, SamplingParams
import os
os.environ["HF_ENDPOINT"] = "https://hf-mirror.com"
ray.init(num_gpus=8, num_cpus=64)
print(ray.nodes())  # CPU: 96, GPU: 8

# assert 1 == 0

@ray.remote(num_gpus=2)
class vLLMWrapper:
    def __init__(self):
        self.model = LLM(model="facebook/opt-125m")

    def generate(self, prompts, sampling_params):
        outputs = self.model.generate(prompts, sampling_params)

        # Print the outputs.

        re_outputs = []
        for output in outputs:

            # prompt = output.prompt

            generated_text = output.outputs[0].text
            re_outputs.append(generated_text)

            # print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")

        return re_outputs
```

```
prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)

# 创建 vLLM 实例
vllm_instance = vLLMWrapper.remote()

# 生成文本
generate_task = vllm_instance.generate.remote(prompts, sampling_params)
generated_text = ray.get(generate_task)

# 输出生成的文本
print("Input Text:")
print(prompts)
print("Generated Text:")
print(generated_text)
```

部署多个vLLM并做数据并行推理

```
# 创建多个 vLLM 实例，此时分配的资源需要注意
num_instances = 4
use_gpus = [1, 1, 2, 2]  # 每个实例可能所用资源不一样
vllm_instances = [vLLMWrapper.options(num_gpus=num_gpu).remote() for num_gpu in
use_gpus]

# 准备输入数据
input_texts = ["Text 1", "Text 2", "Text 3", "Text 4"]

# 并行生成文本
generate_tasks = [vllm.generate.remote(text) for vllm, text in
zip(vllm_instances, input_texts)]
generated_texts = ray.get(generate_tasks)

# 输出生成的文本
for text in generated_texts:
    print(text)
```

DP（Data Parallel） Rank调用

DP Rank：模型并行将数据集划分为多个子集，每个子集被分配给独立的进程（称为rank），因此DP Rank指这些并行训练的进程。

若推理则与上述大致一致；若为训练，还需要AllReduce同步梯度，通过`ray.get()`与`ray.put()`完成集合通信实现梯度的取入和取出。

参考资料

1. https://github.com/OpenRLHF/OpenRLHF
2. https://docs.ray.io/
3. https://zhuanlan.zhihu.com/p/678828949

4. https://www.usenix.org/system/files/osdi18-moritz.pdf：Ray: A Distributed Framework for Emerging AI Applications
5. https://openmlsys.github.io/
6. https://www.cnblogs.com/jsxyhelu/p/18155194
7. https://docs.vllm.ai/

说明

1. windows不支持vllm，实验室的资源当前被占用，因此代码未完全跑通（但是可以跑通）
2. 周末事情较多，完成该任务时间段为8月4日下午至今，时间有限难免疏漏，还望海涵。