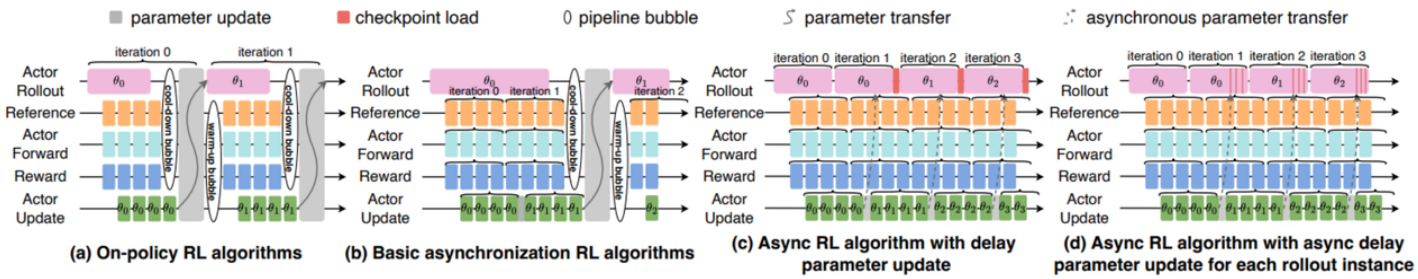
One step async rl

问题背景

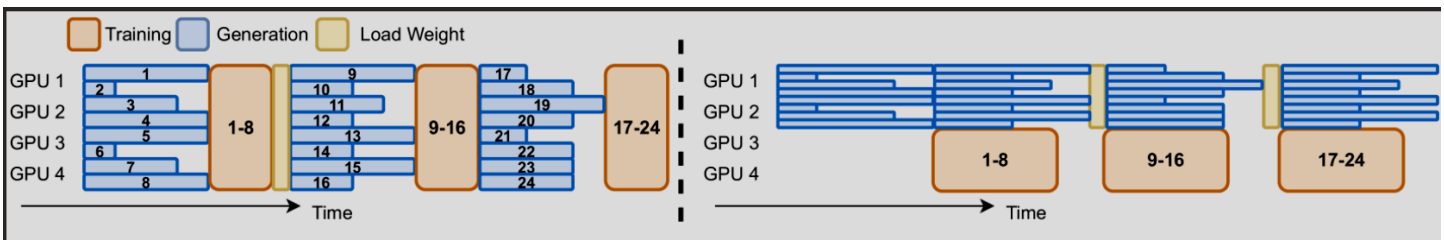
当前 verl 默认的 RL 流程是全共卡同步的。在每个步骤中，由最新模型生成训练样本，并在训练完成后更新模型。这种全同步方案有明显的效率问题，因为 policy model 的 update 需要依赖生成阶段最长输出完成。在生成长尾样本时，GPU 处于空闲状态，导致资源利用率不高，且样本生成中的长尾问题约严重，端到端训练吞吐就越低。例如，在 DAPO Qwen2.5-32B 模型训练中，Rollout 占了总时间的 70%，增加资源并不能减少 Rollout 的持续时间。

方案

由于 verl 当前按照 batch (`train_batch_size`)的维度进行数据流转的，要实现类似 AsyncFlow 那样的以 `micro_batch_size` 粒度的异步 RL 系统工作量较大



先参考 AReaL 提出的 one-step-overlap RL 方案缓解上述问题：这种方法将 Rollout 和训练阶段并行起来，利用前一步生成的样本进行当前的训练，并为 Rollout 分配专用资源，剩余资源分配给训练。整个过程，生成和训练参数保持 one-step off policy。



AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning

实现

One step aysnc pipeline

One step async pipeline 和核心机制是通过 `async_gen_next_batch` 方法进行异步 rollout，按照如下伪代码编排 `ray_trainer.py` 的控制流程：

代码块

```
def fit(self):
    ...

    # iterator generator, simplify one-step integration of the training process
    def create_continuous_iterator(self):
        for epoch in range(self.config.trainer.total_epochs):
            iterator = iter(self.train_dataloader)
            for batch_dict in iterator:
                yield epoch, batch_dict

    # read next batch samples, parameters sync and launch asyn gen_seq

    def async_gen_next_batch(self, continuous_iterator):
        # 1. read next batch samples
        try:
            epoch, batch_dict = next(continuous_iterator)
        except StopIteration:
            return Nonebatch = DataProto.from_single_dict(batch_dict)
        gen_batch = batch_pocess(batch)

        # 2. parameters sync(actor -> rollout)
        self.sync_rollout_weights()

        # 3. async generation
        gen_batch_output = self.rollout_wg.async_generate_sequences(gen_batch)

        # 4. return future handler
        return GenerationBatchFuture(epoch, batch, gen_batch_output)

    continuous_iterator = self.create_continuous_iterator()
    # run rollout first to achieve one-step-off
    batch_data_future = self.async_gen_next_batch(continuous_iterator)

    while batch_data_future is not None:
        # 1. wait for the gen_seq result from the previous step
        batch = batch_data_future.get()

        # 2. launch the next async call to generate sequences
        batch_data_future = self.async_gen_next_batch(continuous_iterator)

        # 3. compute advantages
        batch = critic.compute_values(batch)
        batch = reference.compute_log_prob(batch)
        batch = reward.compute_reward(batch)
        batch = compute_advantages(batch)
```

```
46
47          # 4. model update
48          critic_metrics = critic.update_critic(batch)
49          actor_metrics = actor.update_actor(batch)
```

Parameter synchronization

参数同步走 hccl 后端，需要在 Rollout workers 和 Actor workers之间建单独的集合通信组，由于 PyTorch 的约束仅允许建一个全局进程组，参考 OpenRLHF 实现重写 `init_process_group` 以支持组不同的全局通信组

https://github.com/OpenRLHF/OpenRLHF/blob/728d35d65beb7a678c5069b2b635ece10cd3fea6/openrlhf/utils/distributed_util.py#L29

代码块

```
1   # actor&rollout worker
2   class ActorRolloutRefWorker(ARRWorker):
3       # actor acquires the meta-info of model parameters for parameter sync
4       @register(dispatch_mode=Dispatch.ONE_TO_ALL)
5       def get_actor_weights_info(self):
6           params = self._get_actor_params()
7           ret = []
8           for key, tensor in params.items():
9               ret.append((key, tensor.size(), tensor.dtype))
10          self._weights_info = ret
11          return ret
12
13      # rollout sets the meta-info of model parameters for parameter sync
14      @register(dispatch_mode=Dispatch.ONE_TO_ALL)
15      def set_actor_weights_info(self, weights_info):
16          assert self._is_rollout
17          self._weights_info = weights_info
18
19      # create parameter sync group between actor & rollout worker
20      @register(dispatch_mode=Dispatch.ONE_TO_ALL, blocking=False)
21      def create_weight_sync_group(
22          self, master_address, master_port, world_size, rank_offset,
    backend="hccl", group_name="actor_rollout"
23      ):
24          rank = self.rank + rank_offset
25          self._weight_sync_group = init_process_group( # using a custom
    init_process_group function
26              backend=backend,
27              init_method=f"tcp://{master_address}:{master_port}",
28              world_size=world_size,
29              rank=rank,
```

```
            group_name=group_name
        )


# --------------------
class AsyncRayPPOTrainer(RayPPOTrainer):
    def init_workers(self):
        ...
        weights_info = self.actor_wg.get_actor_weights_info()[0]
        self.rollout_wg.set_actor_weights_info(weights_info)
        self.create_weight_sync_group()

    def create_weight_sync_group():
        if not self.hybrid_engine:
            master_address =
ray.get(self.actor_wg.workers[0]._get_node_ip.remote())
            master_port =
ray.get(self.actor_wg.workers[0]._get_free_port.remote())
            world_size = len(self.actor_wg.workers + self.rollout_wg.workers)
            self.actor_wg.create_weight_sync_group(
                master_address,
                master_port,
                world_size,
                rank_offset=0,
                backend="hccl",
                group_name="actor_rollout",
            )
            ray.get(self.rollout_wg.create_weight_sync_group(
                master_address,
                master_port,
                world_size,
                rank_offset=len(self.actor_wg.workers),
                backend="hccl",
                group_name="actor_rollout",
            ))
```

代码块

```
class AsyncRayPPOTrainer(RayPPOTrainer):
    # drive process call the actor and rollout respectively to sync parameters
by hccl
    def sync_rollout_weights(self):
        if not self.hybrid_engine:
            self.actor_wg.sync_rollout_weights()
            ray.get(self.rollout_wg.sync_rollout_weights())

```

```
# --------------------------
# Megatron Model
@register(dispatch_mode=Dispatch.ONE_TO_ALL, blocking=False)
def sync_rollout_weights(self):
    assert (self._is_actor or self._is_rollout) and not
self.config.hybrid_engine
    assert hasattr(self, "_weights_info") and self._weights_info is not None

    params = self._get_actor_params() if self._is_actor else None
    if self._is_rollout:
        inference_model = (

self.rollout.inference_engine.llm_engine.model_executor.driver_worker.worker.mo
del_runner.model
        )
        patch_vllm_moe_model_weight_loader(inference_model)
        params_generator = iter(params)
    for key, shape, dtype in self._weights_info:
        if self._is_actor:
            weight_key, weight = next(params_generator)
            assert key == weight_key
            assert shape == weight.size()
            assert dtype == weight.dtype

        tensor = torch.empty(shape, dtype=dtype,
device=get_torch_device().current_device())
        if self._is_actor and torch.distributed.get_rank() == 0:
            tensor.copy_(weight)

        torch.distributed.broadcast(tensor, 0, group=self._weight_sync_group)
        if self._is_rollout:
            inference_model.load_weights([(key, tensor)])
```