A.1 RL in Spark Streaming

PPO Implementation. In Figure A1, we show the high-level pseudocode of our port of the PPO algorithm to Spark Streaming. Similar to our port of RLlib to RLlib Flow, we only changed the parts of the PPO algorithm in RLlib that affect distributed execution, keeping the core algorithm implementation (e.g., numerical definition of policy loss and neural networks in TensorFlow) as similar as possible for fair comparison. We made a best attempt at working around aforementioned limitations (e.g., using a `binaryRecordsStream` input source to efficiently handle looping, defining efficient serializers for neural network state, and adjusting the microbatching to emulate the RLlib configuration).

```
# RL on Spark Streaming:
# Iterate by saving/detecting states file in a folder:
#   1) Replicate the states to workers
#   2) Sample in parallel (map)
#   3) Collect the samples (reduce)
#   4) Train on sampled batch
#   5) Save the states and trigger next iteration

# Set up the Spark cluster
sc = SparkContext(master_addr)
# Spark detects new states file in path
states = sc.binaryRecordsStream(path)
rep = states.flatMap(replicate_fn)
split = rep.repartion(NUM_WORKERS)
# Restore actor from states and sample
sample = splits.map(actor_sample_fn)
# Collect all samples from actors
reduced = sample.reduce(merge_fn)
# Restore trainer from states and train
new_states = reduced.map(train_fn)
# Save sampling/training states to path
new_states.foreachRDD(save_states_fn)
```

Figure A1: Example of Spark Streaming for Distributed RL.

Experiment Setup. We conduct comparisons between the performance of both implementations. In the experiment, we adopt the PPO algorithm for the CartPole-v0 environment with a fixed sampling batch size B of 100K. Each worker samples (B/# workers) samples each iteration, and for simplicity, the learner updates the model on CPU using a minibatch with 128 samples from the sampled batch. Experiments here are conducted on AWS m4.10xlarge instances.

Data Framework Limitations: Spark Streaming is a data streaming framework designed for general purpose data processing. We note several challenges we encountered attempting to port RL algorithms to Spark Streaming:

1. Support for asynchronous operations. Data processing systems like Spark Streaming do not support asynchronous or non-deterministic operations that are needed for asynchronous RL algorithms.
2. Looping operations are not well supported. While many dataflow models in principle support iterative algorithms, we found it necessary to work around them due to lack of language APIs (i.e., no Python API).
3. Support for non-serializable state. In the dataflow model, there is no way to persist arbitrary state (i.e., environments, neural network models on the GPU). While necessary for fault-tolerance, the requirement for serializability impacts the performance and feasibility of many RL workloads.
4. Lack of control over batching. We found that certain constructs such as the data batch size for on-policy algorithms are difficult to control in traditional streaming frameworks, since they are not part of the relational data processing model.

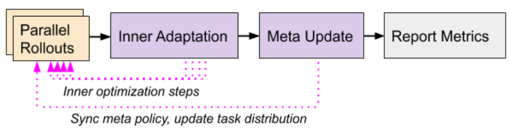
For a single machine (the left three pairs), the breakdown of the running time indicates that the initialization and I/O overheads slow down the training process for Spark comparing to our RLlib Flow. The former overheads come from the nature of Spark that the transformation functions do not persist variables. We have to serialize both the sampling and training states and re-initialize the variables in the next iteration to have a continuous running process. On the other hand, the I/O overheads come from looping back the states back to the input. As an event-time driven streaming system, the stream engine detects changes for the saved states from the source directory and starts new stream processing. The disk I/O leads to high overheads compared to RLlib Flow.

For distributed situation (the right three pairs), the improvement of RLlib Flow becomes more significant against Spark, up to 2.9×. As the number of workers scales up, the sampling time decreases for both the dataflow model. Still, the initialization and I/O overheads stay unchanged, leading to lesser scalability for Spark.

A.2 Implementation Examples

A.2.1 Example: MAML

Figure A2b concisely expresses MAML's dataflow (also shown in Figure A2a) [10]. The MAML dataflow involves nested optimization loops; workers collect pre-adaptation data, perform inner adaptation (i.e., individual optimization calls to an ensemble of models spread across the workers), and collect post-adaptation data. Once inner adaptation is complete, the accumulated data is batched together to compute the meta-update step, which is broadcast to all workers.



(a) MAML dataflow includes a number of nested inner adaptation steps (optimization calls to the source actors) prior to update of the meta-policy. The meta-policy update and inner adaptation steps integrate cleanly into the dataflow, their ordering guaranteed by the synchronous data dependency barrier between the inner adaptation and meta update steps.

```python
# type: List[RolloutActor]
workers = create_rollout_workers()
# type: Iter[List[Rollouts]]
rollouts = ParallelRollouts(workers).gather_sync()
# MAML Algo:
#   Loop until inner adaptation done:
#      1) Aggregate Data from Workers
#      2) Perform Inner Adaption on Data
adapt_op = rollouts.combine(InnerAdapt(workers, steps=5))
# Meta-update Step
train_op = adapt_op.for_each(MetaUpdate(workers))
# type: Iter[Metrics]
return ReportMetrics(train_op, workers)
```

(b) Implementation in RLlib Flow.

Figure A2: Dataflow and implementation of the MAML algorithm.

A.3 Comparison of Implementations in RLlib Flow and RLlib

In this section we report the detailed code comparison of our RLlib Flow and the original RLlib. Listing A1 and Listing A2 are the detailed implementation of A3C in RLlib Flow and RLlib, respectively. Note that the detailed implementation in Listing A1 is exactly the same as we shown before in Figure 9a, but RLlib implementation is much more complicated as the intermixing of the control and data flow. In Listing A3 and Listing A4, we also show the detailed implementation of Ape-X algorithm in our RLlib Flow and RLlib respectively, which also indicates the simplicity, readability and flexibility of our RLlib Flow.

Listing A1: Detailed A3C in RLlib Flow.

```python
# type: List[RolloutActor]
workers = create_rollout_workers()
# type: Iter[Gradients]
grads = ParallelRollouts(workers)
    .par_for_each(ComputeGradients())
    .gather_async()
# type: Iter[TrainStats]
apply_op = grads
    .for_each(ApplyGradients(workers))
# type: Iter[Metrics]
return ReportMetrics(apply_op, workers)
```

Listing A2: Detailed A3C in original RLlib.

```python
# Create timers
apply_timer = TimerStat()
wait_timer = TimerStat()
dispatch_timer = TimerStat()

# Create training information
num_steps_sampled = 0

# type: List[RolloutActor]
workers = create_rollout_workers()

# Get weights from the local rollout actor
local_worker = workers.local_worker()
weights = local_worker.get_weights()

# Put weights in raylet (distributed storage)
weights = ray.put(weights)

# type: Dict[obj_id, RolloutActor]
pending_gradients = dict()

# Get the remote rollout actors
remote_worker = workers.remote_workers()

# Issue gradient computation tasks
for worker in remote_worker:
    # Set weight on remote rollout actor
    worker.set_weights.remote(weights)
    # Collect samples from the remote rollout actor
    samples = worker.sample.remote()

    # Kick off gradient computation
    future = worker.compute_gradients.remote(samples)

    # Map the object id to rollout actor
    pending_gradients[future] = worker

# Start training loop
while pending_gradients:
    # Record the time to wait gradient
    with wait_timer:
        # Get the list of the futures
        futures = list(pending_gradients.keys())

        # Wait for one actor to complete
        wait_results = ray.wait(futures,
                                num_returns=1)

        # Get the ready future
        ready_list = wait_results[0]
        future = ready_list[0]

        # Get and free the gradient and training infos
```

```python
        # from the raylet (maybe on the remote worker)
        gradient, info = ray_get_and_free(future)

        # Pop the used gradient from the map
        worker = pending_gradients.pop(future)

    # Check the validation of the gradient
    if gradient is not None:
        # Record the time for gradient apply
        with apply_timer:
            # Apply the gradient on the local worker
            local_worker = workers.local_worker()
            local_worker.apply_gradients(gradient)

        # Record the metrics from the worker
        num_steps_sampled += info["batch_count"]

    # Record the time to set new weight on the worker
    # and launch gradient computation task
    with dispatch_timer:
        # Get the weight on local rollout actor
        local_worker = workers.local_worker()
        weights = local_worker.get_weights()

        # Set weight on the rollout actor
        worker.set_weights.remote(weights)

        # Sample rollouts on the rollout actor
        samples = worker.sample.remote()
        # Launch gradient computation task on the worker
        future = worker.compute_gradients.remote(samples)

        # Map the new object id to the corresponding worker
        pending_gradients[future] = worker
```

Listing A3: Detailed Ape-X in RLlib Flow.

```python
# type: List[RolloutActor]
workers = create_rollout_workers()

# Create a number of replay buffer actors.
replay_actors = create_colocated(ReplayActor)

# Start the learner thread.
learner_thread = LearnerThread(workers.local_worker())
learner_thread.start()

# We execute the following steps concurrently:
# (1) Generate rollouts and store them in our replay buffer actors. Update
# the weights of the worker that generated the batch.
rollouts = ParallelRollouts(workers, mode="async", num_async=2)
store_op = rollouts \
```

```python
    .for_each(StoreToReplayBuffer(actors=replay_actors))

# Only need to update workers if there are remote workers.
store_op = store_op.zip_with_source_actor() \
    .for_each(UpdateWorkerWeights(workers))

# (2) Read experiences from the replay buffer actors and send to the
# learner thread via its in-queue.
replay_op = Replay(actors=replay_actors, num_async=4) \
    .zip_with_source_actor() \
    .for_each(Enqueue(learner_thread.inqueue))

# (3) Get priorities back from learner thread and apply them to the
# replay buffer actors.
update_op = Dequeue(learner_thread.outqueue) \
    .for_each(UpdateReplayPriorities()) \
    .for_each(TrainOneStep(workers))

# Execute (1), (2), (3) asynchronously as fast as possible. Only output
# items from (3) since metrics aren't available before then.
merged_op = Concurrently(
    [store_op, replay_op, update_op], mode="async", output_indexes=[2])

return ReportMetrics(merged_op, workers)
```

Listing A4: Detailed Ape-X in original RLlib. We leave out some of the configurable argument for simplicity.

```python
# type: List[RolloutActor]
workers = create_rollout_workers()

# Create a learner thread in the main driver to handle
# the asynchronous training
local_worker = workers.local_worker()
learner = LearnerThread(local_worker)

# Start the learner thread and wait for the input
learner.start()

# Create replay actor handling the replay buffer
# create_located: create multiple colocated replay actor
# in the same machine as main driver
replay_actors = create_colocated(ReplayActor)

# Create timers
timers = {
    k: TimerStat()
    for k in [
        "put_weights", "get_samples", "sample_processing",
        "replay_processing", "update_priorities", "train", "sample"
    ]
}
```

```python

# Create training information
num_weight_syncs = 0
num_samples_dropped = 0
learning_started = False

# Number of worker steps since the last weight update
steps_since_update = dict()

# Create manager for replay
replay_tasks = TaskPool()
# Kick off replay tasks for local gradient updates
for actor in replay_actors:
    # Start replay task on remote replay actors
    for _ in range(REPLAY_QUEUE_DEPTH):
        replay_task = actor.replay.remote()
        # add replay task into the manager
        replay_tasks.add(actor, replay_task)

# Create manager for sampling
sample_tasks = TaskPool()

# Get weights of local worker
local_worker = workers.local_worker()
weights = local_worker.get_weights()

# Kick off async background sampling and set the weights
# on remote rollout actors
remote_workers = workers.remote_workers()
for worker in remote_workers:
    # Set weights
    worker.set_weights.remote(weights)
    # Initialize training info for the rollout actor
    steps_since_update[worker] = 0
    for _ in range(SAMPLE_QUEUE_DEPTH):
        # Start sample_with_count task on remote worker
        sample_with_count_task = worker.sample_with_count.remote()
        # Add task in to the sample task manager
        sample_tasks.add(worker, sample_with_count_task)

# Optimize the model for one step
def step(self):
    # Check the availability of the asynchronous learner thread
    # and the remote rollout actors
    assert self.learner.is_alive()
    assert len(self.workers.remote_workers()) > 0

    # Record the start time for training info
    start = time.time()

    # Create variables for training
    sample_timesteps, train_timesteps = 0, 0
    weights = None

```

```
    # Record the sampling and processing step
    with timers["sample_processing"]:
        # Check the completed sampling task in the sampling manager (TaskPool)
        completed = list(sample_tasks.completed())

        # Gather the train info, counts of samples
        counts = ray_get_and_free([c[1][1] for c in completed])

        # Update training information and weights
        for i, (worker, (sample_batch, count)) in enumerate(completed):
            # Update training information
            sample_timesteps += counts[i]

            # Randomly choose one replay actor and send data to it
            random_replay_actor = random.choice(replay_actors)
            random_replay_actor.add_batch.remote(sample_batch)

            # Update train info
            steps_since_update[worker] += counts[i]

            # Update weights on remote rollout worker if needed
            if steps_since_update[worker] >= MAX_WEIGHT_SYNC_DELAY:
                # Note that it's important to pull new weights once
                # updated to avoid excessive correlation between actors
                if weights is None or learner.weights_updated:
                    learner.weights_updated = False

                    # Record time for putting weights
                    with timers["put_weights"]:
                        # Put local weights in raylet
                        local_worker = workers.local_worker()
                        local_weights = local_worker.get_weights()
                        weights = ray.put(local_weights)

                # Set weights on the remote rollout worker
                worker.set_weights.remote(weights)

                # Update train info
                num_weight_syncs += 1
                steps_since_update[worker] = 0

            # Kick off another sample request
            sample_with_count = worker.sample_with_count.remote()
            # Add the task into the sample manager
            sample_tasks.add(worker, sample_with_count)

    # Record the time for replay and processing
    with self.timers["replay_processing"]:
        for actor, replay in replay_tasks.completed():
            # Start another replay task for each completed one
            replay_task = actor.replay.remote()
            replay_tasks.add(actor, replay_task)

```

```python
            # Check the input queue of the learner
            if learner.inqueue.full():
                num_samples_dropped += 1
            else:
                # Record the get sample time
                with self.timers["get_samples"]:
                    samples = ray_get_and_free(replay)

                # Defensive copy against plasma crashes
                learner.inqueue.put((actor, samples.copy()))

    # Record the time for priorities update
    with timers["update_priorities"]:
        # Get output from the leaner to update replay priorities on
        # the remote rollout actors and training info
        while not learner.outqueue.empty():
            # Fetch output from the asynchronous learner
            output = learner.outqueue.get()
            actor, priority_dict, count = output

            # Update the priorities on the remote actors
            actor.update_priorities.remote(priority_dict)
            train_timesteps += count

    # Calculate the time information
    time_delta = time.time() - start

    # Collect metrics for training
    timers["sample"].push(time_delta)
    timers["sample"].push_units_processed(sample_timesteps)
    if train_timesteps > 0:
        learning_started = True
    if learning_started:
        timers["train"].push(time_delta)
        timers["train"].push_units_processed(train_timesteps)

    # Update training info
    num_steps_sampled += sample_timesteps
    num_steps_trained += train_timesteps
```