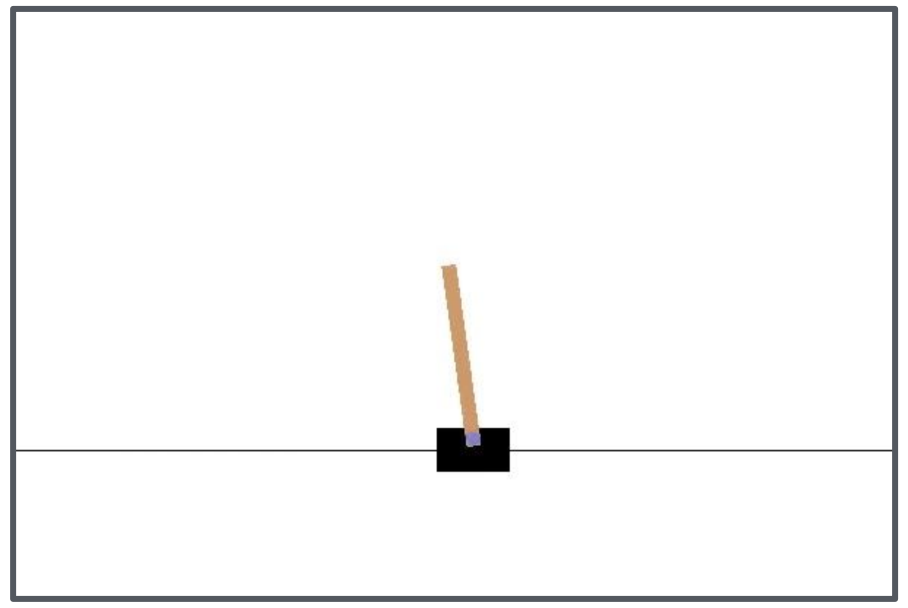
RLlib Flow

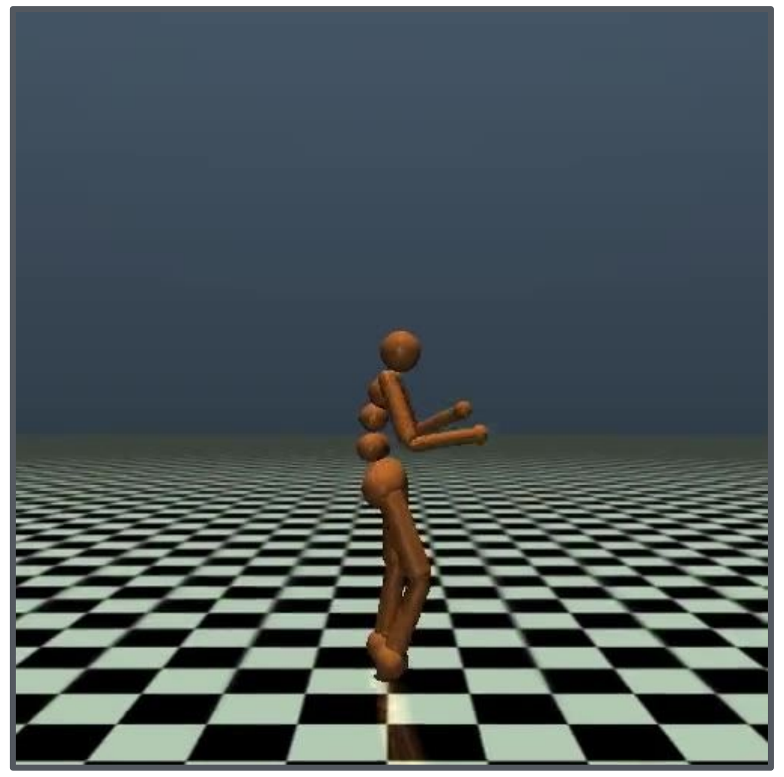
Distributed Reinforcement Learning is a Dataflow Problem

Eric Liang*, **Zhanghao Wu***, Michael Luo, Sven Mika, Joseph E. Gonzalez, Ion Stoica

UC Berkeley, Anyscale

Deep Reinforcement Learning





- Reinforcement learning can be defined in **high-level update equations**.
- The implementation have **remained quite low-level**, i.e. at the level of message passing.

Figure credit to OpenAi Gym

Needs of RL Researchers

Library	Distribution Scheme	Generality	Programmability	#Algo
RLGraph	Pluggable	General Purpose	Low-level / Pluggable	10+
Deepmind Acme	Actors + Reverb	Async Actor-Learner	Limited	10+
Intel Coach	Actor + NFS	Async Actor-Learner	Limited	30+
RLlib	Ray Actors	General Purpose	Flexible, but Low-level	20+
RLlib Flow	Actor / Dataflow	General Purpose	**Flexible and High-level**	20+

- RL practitioners are typically **not system engineers**
- RL algorithms should be **customizable** in various ways

RL Implementation Remains Low Level

```python
# launch gradients computation tasks
pending_gradients = dict()
for worker in remote_workers:
    worker.set_weights.remote(weights)
    future = worker.compute_gradients
        .remote(worker.sample.remote())
    pending_gradients[future] = worker
# asynchronously gather gradients and apply
while pending_gradients:
    wait_results = ray.wait(
        pending_gradients.keys(),
        num_returns=1)
    ready_list = wait_results[0]
    future = ready_list[0]

    gradient, info = ray.get(future)
    worker = pending_gradients.pop(future)
    # apply gradients
    local_worker.apply_gradients(gradient)
    weights = local_worker.get_weights()
    worker.set_weights.remote(weights)
    # launch gradient computation again
    future = worker.compute_gradients
        .remote(worker.sample.remote())
    pending_gradients[future] = worker
```

A3C Implementation in RLlib

Data Flow

Worker Management

Execution Logic

RL Implementation Remains Low Level

```
# launch gradients computation tasks
pending_gradients = dict()
for worker in remote_workers:
    worker.set_weights.remote(weights)
    future = worker.compute_gradients
        .remote(worker.sample.remote())
    pending_gradients[future] = worker
# asynchronously gather gradients and apply
while pending_gradients:
    wait_results = ray.wait(
        pending_gradients.keys(),
        num_returns=1)
    ready_list = wait_results[0]
    future = ready_list[0]

    gradient, info = ray.get(future)
    worker = pending_gradients.pop(future)
    # apply gradients
    local_worker.apply_gradients(gradient)
    weights = local_worker.get_weights()
    worker.set_weights.remote(weights)
    # launch gradient computation again
    future = worker.compute_gradients
        .remote(worker.sample.remote())
    pending_gradients[future] = worker
```

A3C Implementation in RLlib

Data Flow

Worker Management

Execution Logic

RL Implementation Remains Low Level

```
# launch gradients computation tasks
pending_gradients = dict()
for worker in remote_workers:
    worker.set_weights.remote(weights)
    future = worker.compute_gradients
        .remote(worker.sample.remote())
    pending_gradients[future] = worker
# asynchronously gather gradients and apply
while pending_gradients:
    wait_results = ray.wait(
        pending_gradients.keys(),
        num_returns=1)
    ready_list = wait_results[0]
    future = ready_list[0]

    gradient, info = ray.get(future)
    worker = pending_gradients.pop(future)
    # apply gradients
    local_worker.apply_gradients(gradient)
    weights = local_worker.get_weights()
    worker.set_weights.remote(weights)
    # launch gradient computation again
    future = worker.compute_gradients
        .remote(worker.sample.remote())
    pending_gradients[future] = worker
```

A3C Implementation in RLlib

Data Flow

Worker Management

Execution Logic

RL Implementation Remains Low Level

```python
# launch gradients computation tasks
pending_gradients = dict()
for worker in remote_workers:
    worker.set_weights.remote(weights)
    future = worker.compute_gradients
        .remote(worker.sample.remote())
    pending_gradients[future] = worker
# asynchronously gather gradients and apply
while pending_gradients:
    wait_results = ray.wait(
        pending_gradients.keys(),
        num_returns=1)
    ready_list = wait_results[0]
    future = ready_list[0]

    gradient, info = ray.get(future)
    worker = pending_gradients.pop(future)
    # apply gradients
    local_worker.apply_gradients(gradient)
    weights = local_worker.get_weights()
    worker.set_weights.remote(weights)
    # launch gradient computation again
    future = worker.compute_gradients
        .remote(worker.sample.remote())
    pending_gradients[future] = worker
```

A3C Implementation in RLlib

Data Flow

Worker Management

Execution Logic

Hard to read, customize and optimize

Complex Algorithms for RL

- Complex algorithms possible but require low-level code
  - Ape-X: 250 lines of Python
  - IMPALA: 694 lines of Python

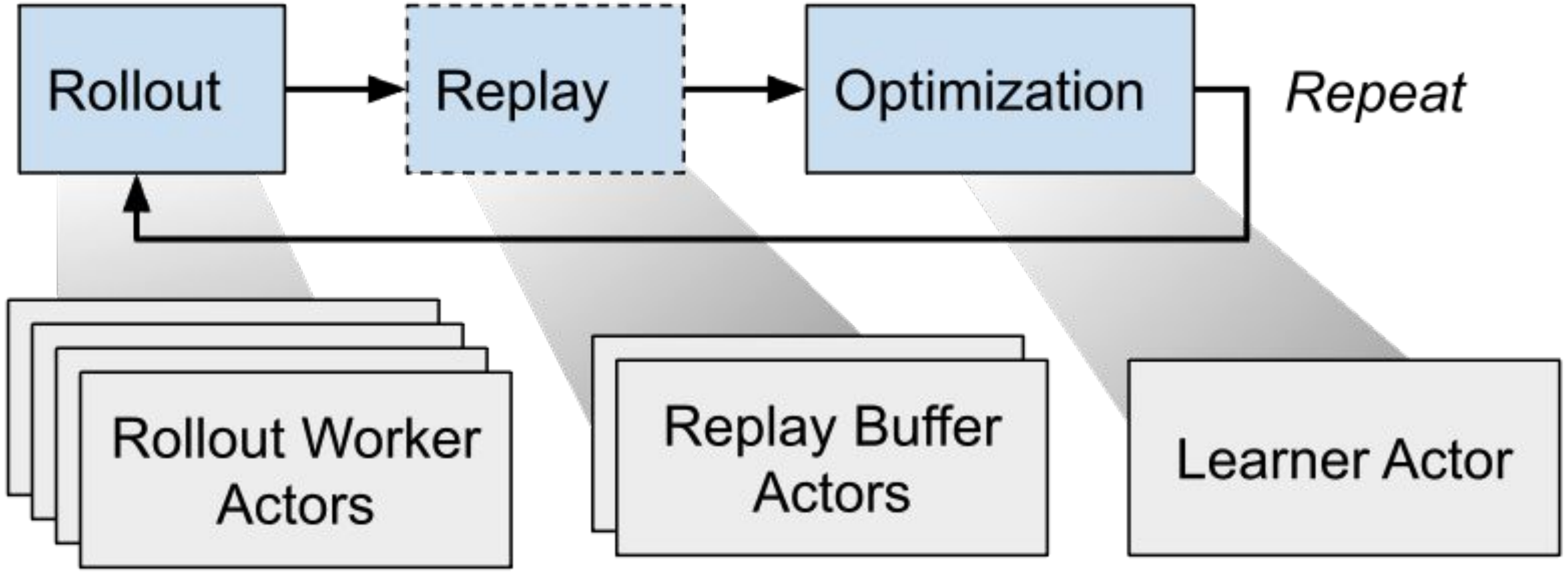
How can we **reduce the lines of code** required to define a new distributed algorithm?

Multi-Agent Use Cases

- From the systems perspective, multi-agent training often does not impact distributed execution
- Exceptions:
  - Training agents different optimization frequencies
  - Training agents with different distributed algorithms

How can we support **composing existing RL algorithms** without requiring a rewrite?

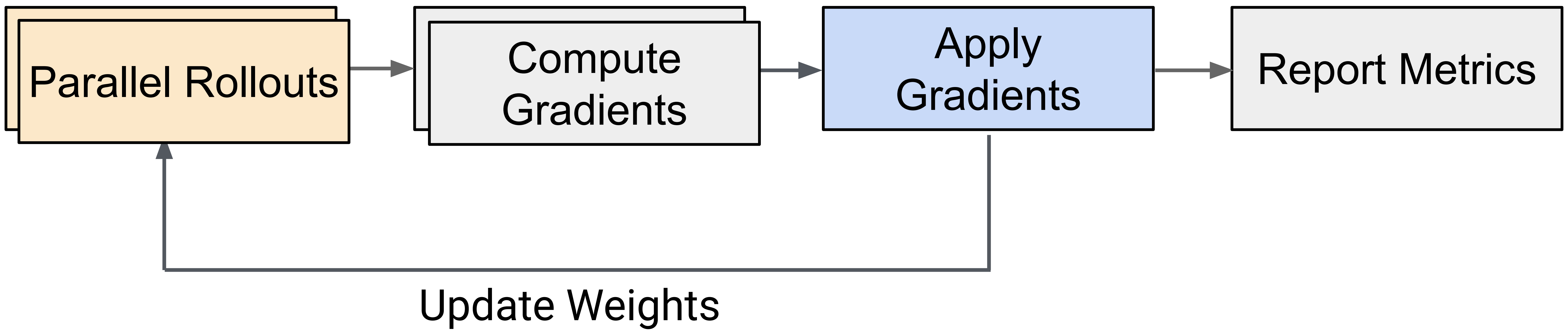
Reinforcement Learning Basics



- RL is more like **data analytics** than supervised learning.
- We can view RL training as **dataflow**

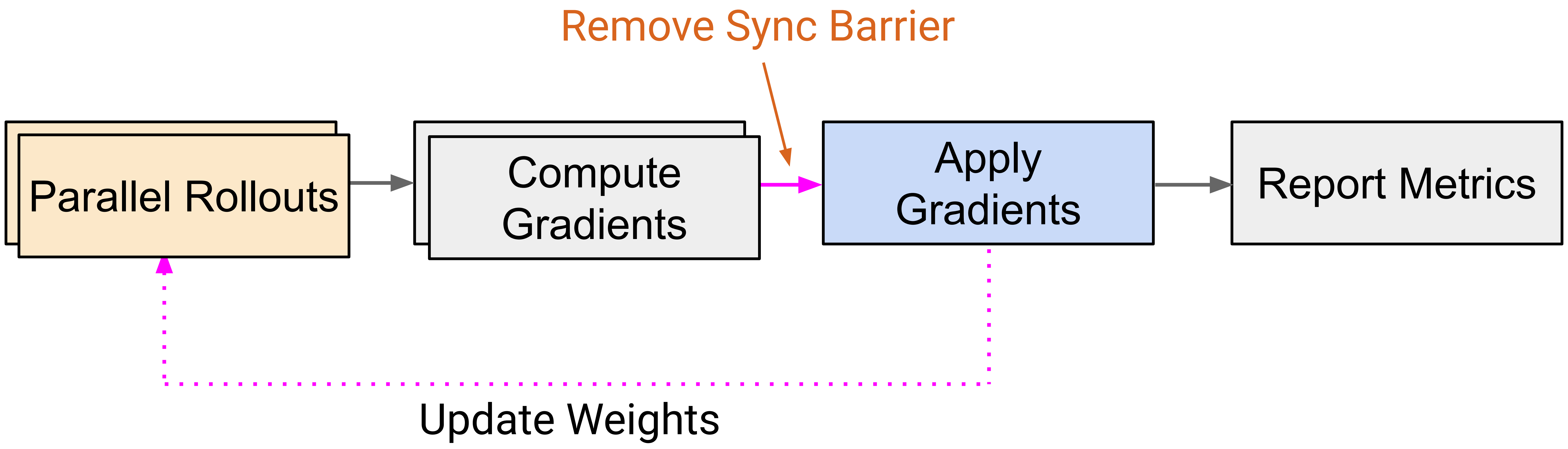
Dataflow of Synchronous Training Loop

- Bulk synchronous algorithms like A2C, PPO.



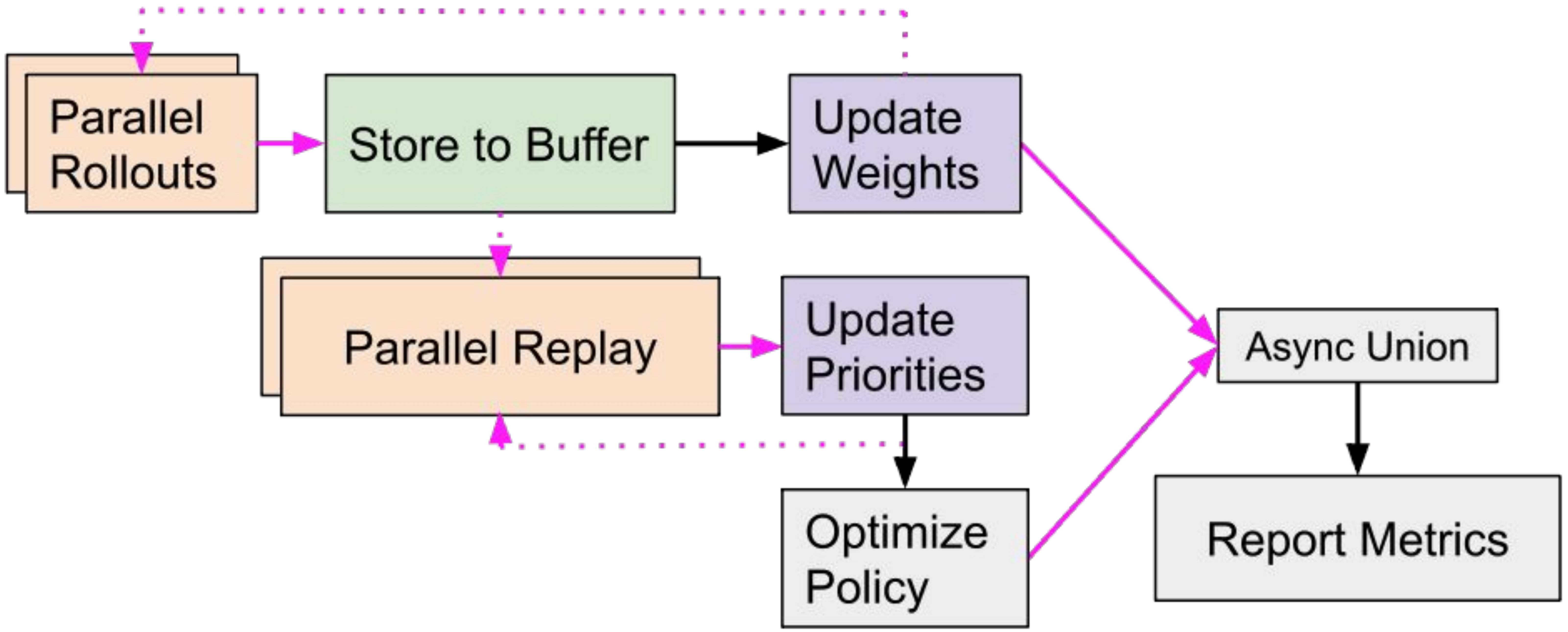
Dataflow of Asynchronous Training

- Small change for async optimization (A3C)

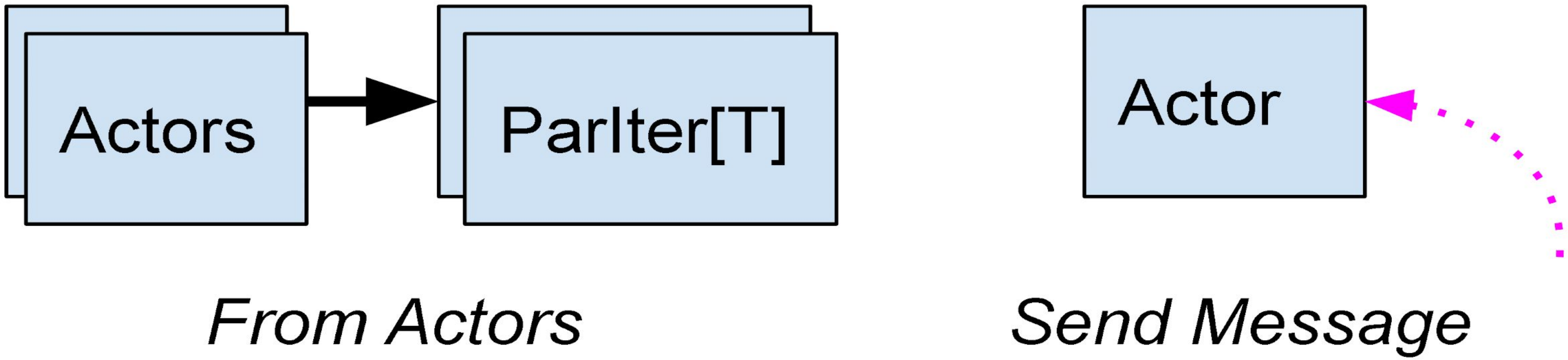


Dataflow of Distributed Prioritized DQN

- Mixed async dataflow (Ape-X), with fine-grained updates

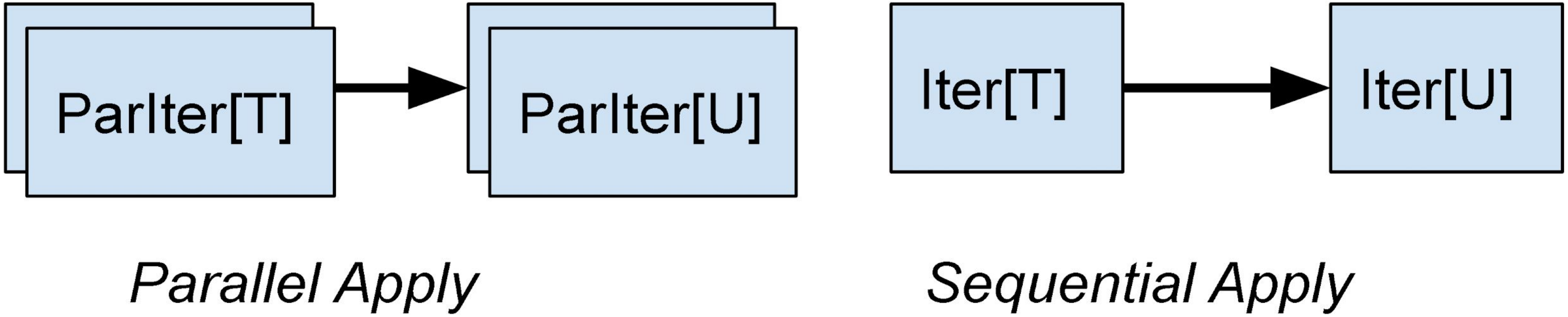


Dataflow Operators for RL



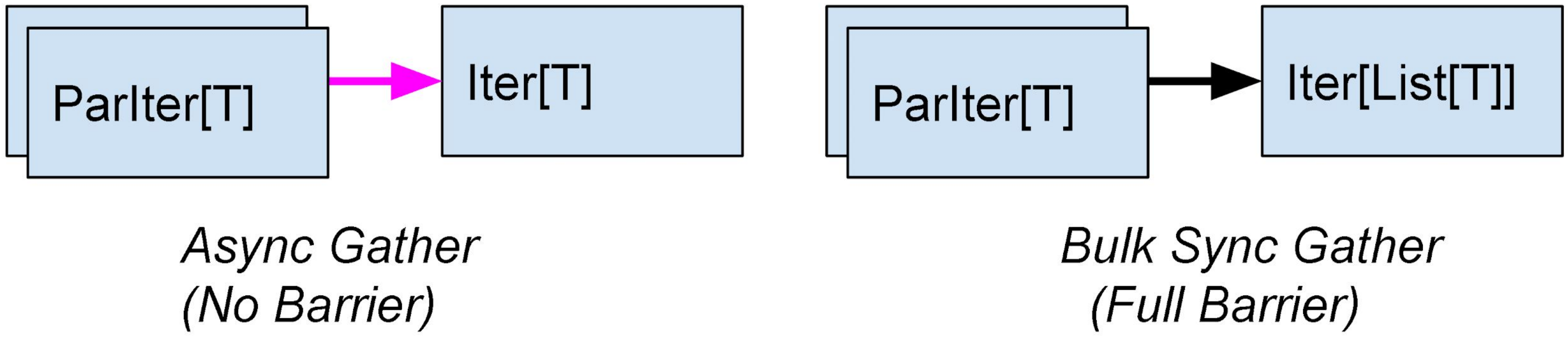
(a) Creation & Message Passing

Dataflow Operators for RL



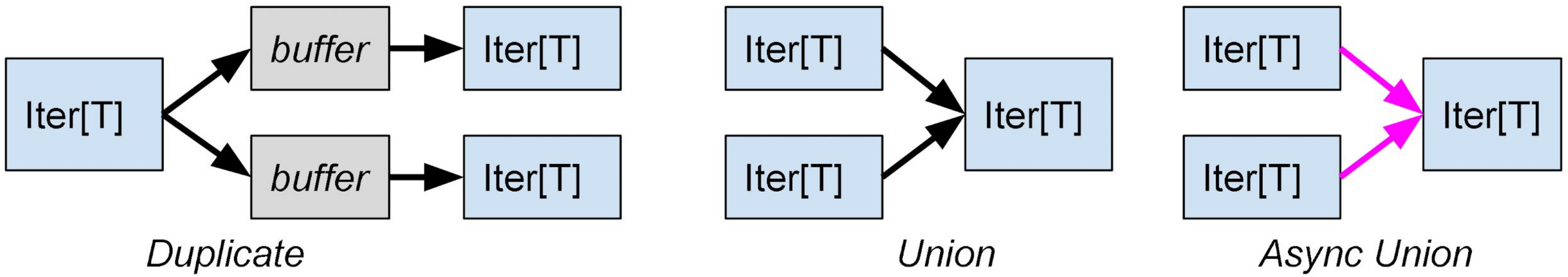
(b) Transformation

Dataflow Operators for RL



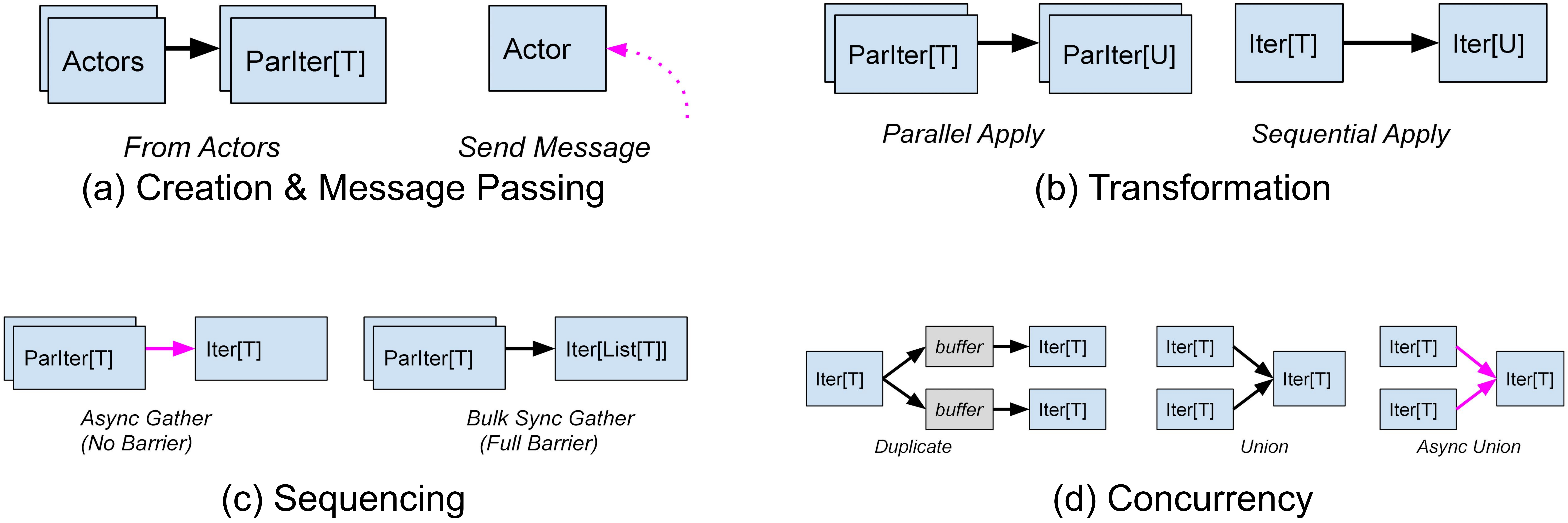
(c) Sequencing

A Dataflow Programming Model for Distributed RL

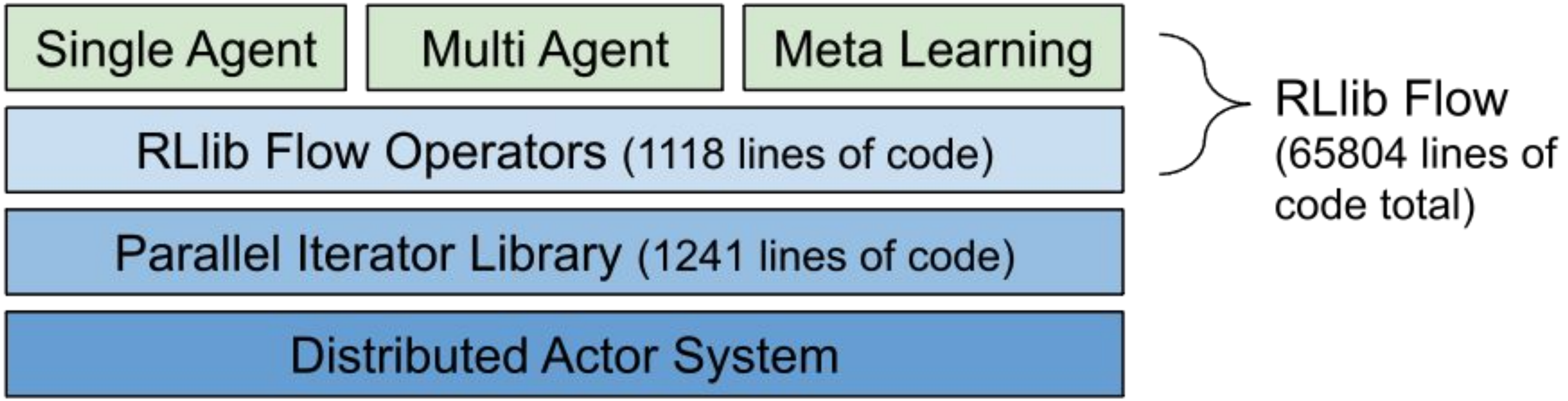


(d) Concurrency

A Dataflow Programming Model for Distributed RL

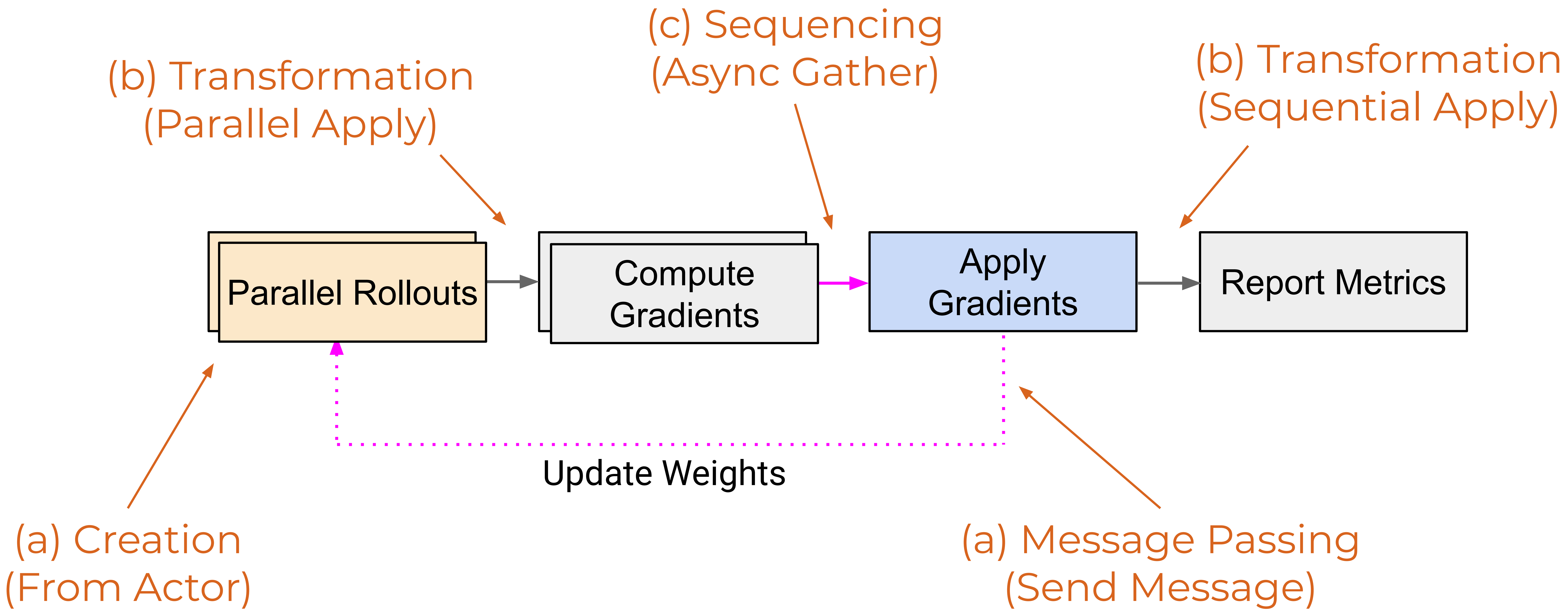


Implementation over Distributed Actor Framework

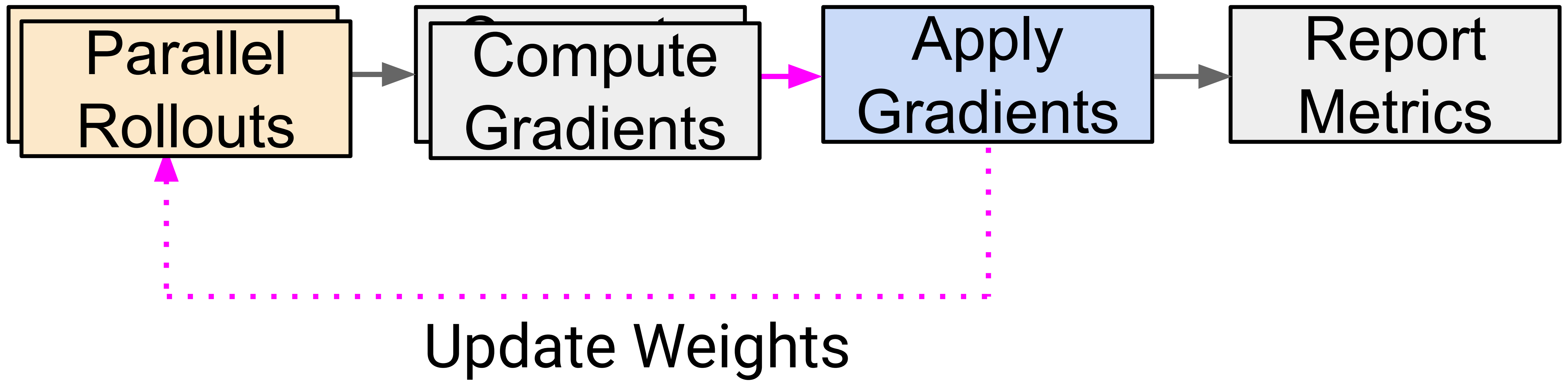


- Two separate modules: A general purpose **parallel iterator library**; a collection of RL specific **dataflow operators**

Evaluation: Revisiting A3C



Evaluation: A3C Comparison



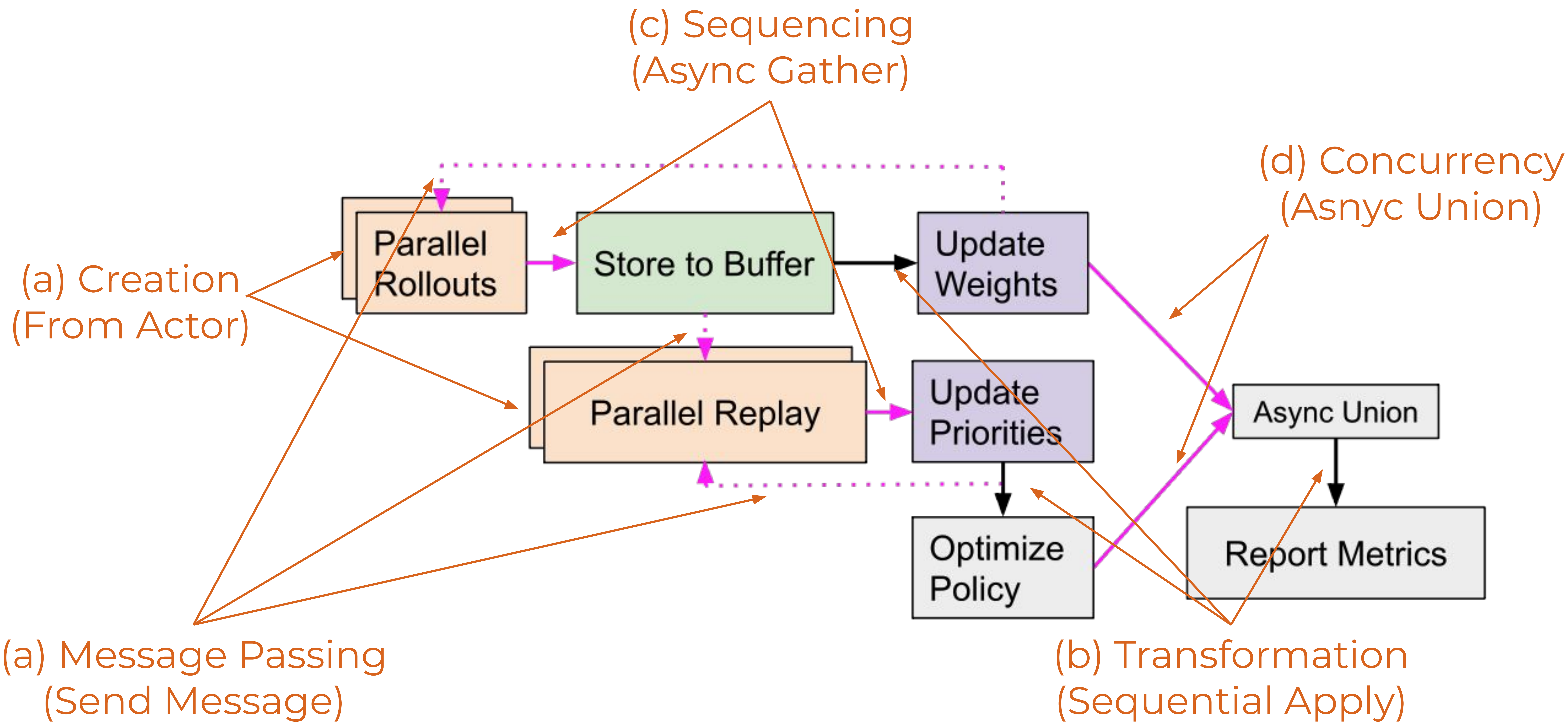

```python
# type: List[RolloutActor]
workers = create_rollout_workers()
# type: Iter[Gradients]
grads = ParallelRollouts(workers)
    .par_for_each(ComputeGradients())
    .gather_async()
# type: Iter[TrainStats]
apply_op = grads
    .for_each(ApplyGradients(workers))
# type: Iter[Metrics]
return ReportMetrics(apply_op,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

A3C Implementation in RLlib Flow

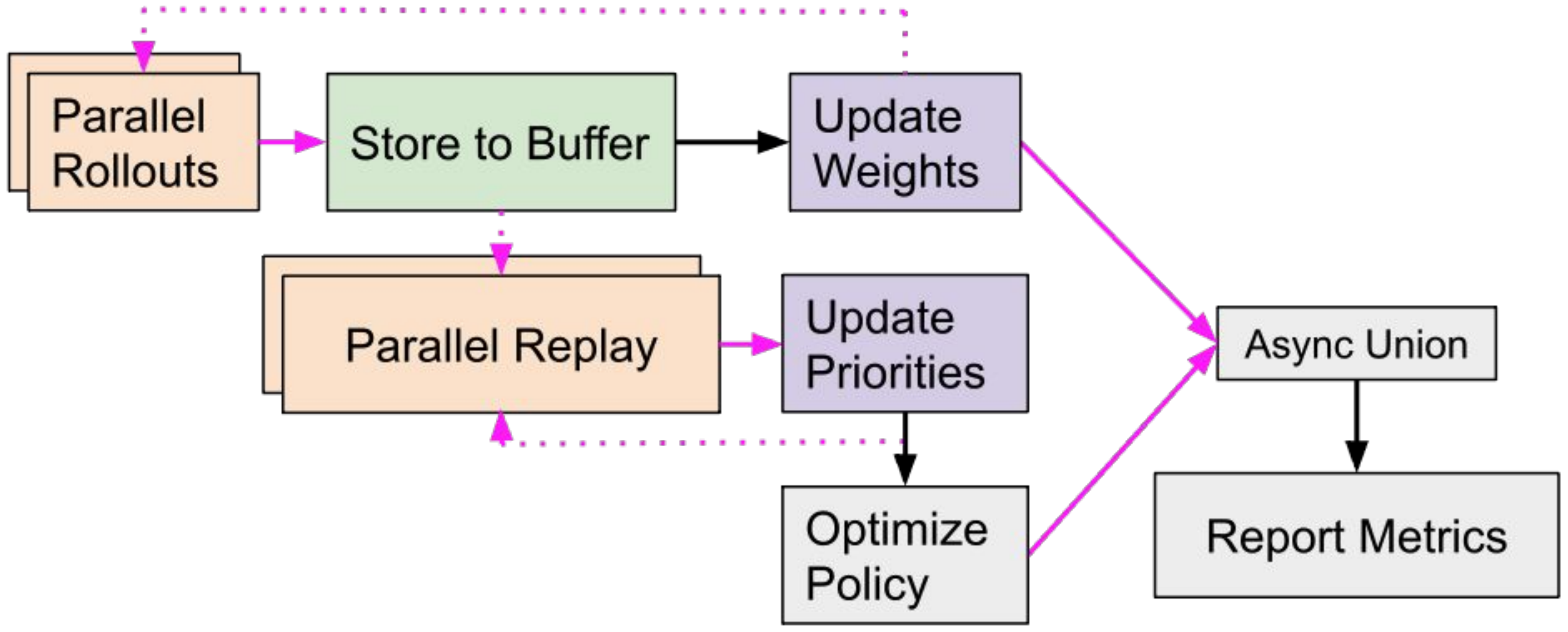
```python
# launch gradients computation tasks
pending_gradients = dict()
for worker in remote_workers:
    worker.set_weights.remote(weights)
    future = worker.compute_gradients
        .remote(worker.sample.remote())
    pending_gradients[future] = worker
# asynchronously gather gradients and apply
while pending_gradients:
    wait_results = ray.wait(
        pending_gradients.keys(),
        num_returns=1)
    ready_list = wait_results[0]
    future = ready_list[0]

    gradient, info = ray.get(future)
    worker = pending_gradients.pop(future)
    # apply gradients
    local_worker.apply_gradients(gradient)
    weights = local_worker.get_weights()
    worker.set_weights.remote(weights)
    # launch gradient computation again
    future = worker.compute_gradients
        .remote(worker.sample.remote())
    pending_gradients[future] = worker
```

A3C Implementation in Previous RLlib

Evaluation: Revisiting Ape-X



Evaluation: Readability (Ape-X)




```
workers = create_rollout_workers()
replay_buffer = create_replay_actors()
rollouts = ParallelRollouts(workers).gather_async()

store_op = rollouts
    .for_each(StoreToBuffer(replay_buffer))
    .for_each(UpdateWeights(workers))

replay_op = ParallelReplay(replay_buffer)
    .gather_async()
    .for_each(UpdatePriorities(workers))
    .for_each(TrainOneStep(workers))

return ReportMetrics(
    Union(store_op, replay_op), workers)
```

Evaluation: Readability (Ape-X)

- Previous implementation:

Source code for ray.rllib.optimizers.async_replay_optimizer

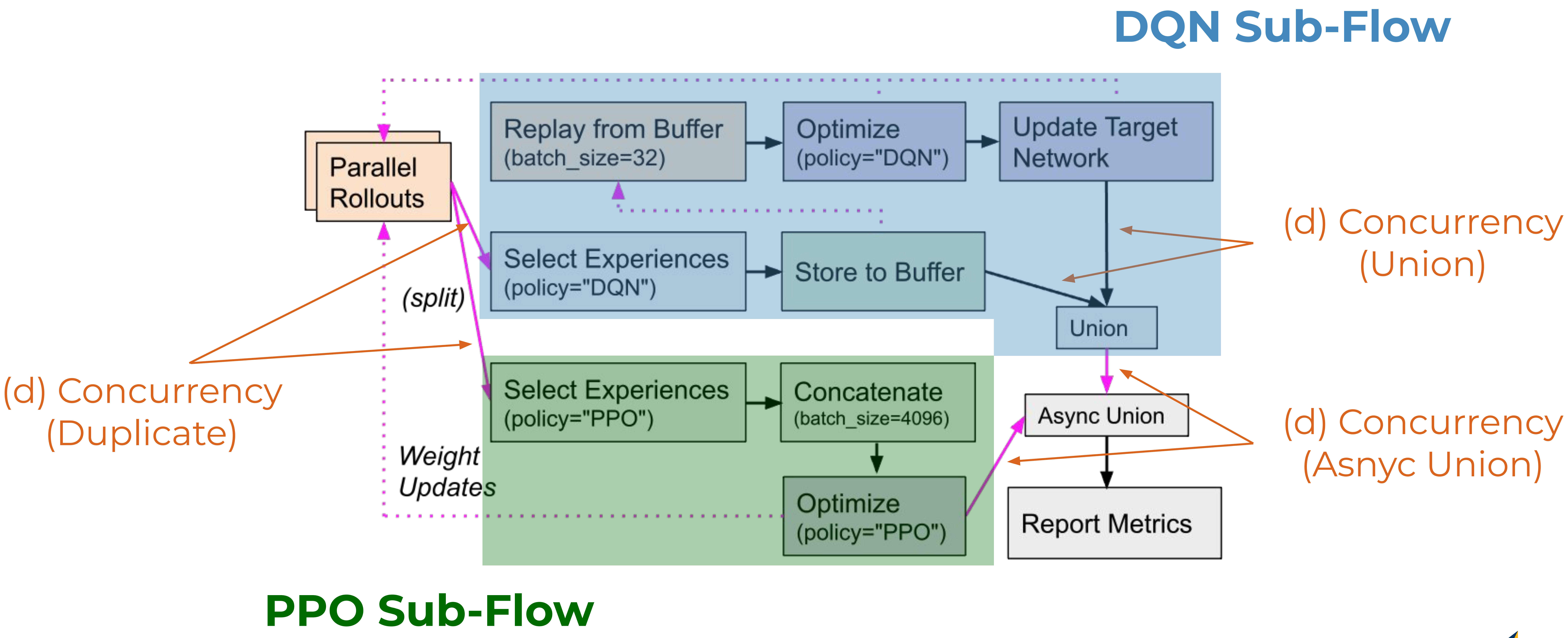
```python
"""Implements Distributed Prioritized Experience Replay.

https://arxiv.org/abs/1803.00933"""

import collections
import logging
import numpy as np
import os
import random
from six.moves import queue
import threading
import time

import ray
from ray.exceptions import RayError
from ray.util.iter import ParallelIteratorWorker
from ray.rllib.evaluation.metrics import get_learner_stats
from ray.rllib.policy.policy import LEARNER_STATS_KEY
from ray.rllib.policy.sample_batch import SampleBatch, DEFAULT_POLICY_ID, \
    MultiAgentBatch
from ray.rllib.optimizers.policy_optimizer import PolicyOptimizer
from ray.rllib.optimizers.replay_buffer import PrioritizedReplayBuffer
from ray.rllib.utils.annotations import override
from ray.rllib.utils.actors import TaskPool, create_colocated
from ray.rllib.utils.memory import ray_get_and_free
from ray.rllib.utils.timer import TimerStat
from ray.rllib.utils.window_stat import WindowStat

SAMPLE_QUEUE_DEPTH = 2
REPLAY_QUEUE_DEPTH = 4
LEARNER_QUEUE_MAX_SIZE = 16

logger = logging.getLogger(__name__)


class AsyncReplayOptimizer(PolicyOptimizer):                            [docs]
    """Main event loop of the Ape-X optimizer (async sampling with replay).

    This class coordinates the data transfers between the learner thread,
    remote workers (Ape-X actors), and replay buffer actors.

    This has two modes of operation:
        - normal replay: replays independent samples.
        - batch replay: simplified mode where entire sample batches are
            replayed. This supports RNNs, but not prioritization.

    This optimizer requires that rollout workers return an additional
    "td_error" array in the info return of compute_gradients(). This error
    term will be used for sample prioritization."""

    def __init__(self,
                 workers,
                 learning_starts=1000,
                 buffer_size=10000,
                 prioritized_replay=True,
                 prioritized_replay_alpha=0.6,
                 prioritized_replay_beta=0.4,
                 prioritized_replay_eps=1e-6,
                 train_batch_size=512,
                 rollout_fragment_length=50,
                 num_replay_buffer_shards=1,
                 max_weight_sync_delay=400,
                 debug=False,
                 batch_replay=False):
        """Initialize an async replay optimizer.
```

```python
        Arguments:
            workers (WorkerSet): all workers
            learning_starts (int): wait until this many steps have been sampled
                before starting optimization.
            buffer_size (int): max size of the replay buffer
            prioritized_replay (bool): whether to enable prioritized replay
            prioritized_replay_alpha (float): replay alpha hyperparameter
            prioritized_replay_beta (float): replay beta hyperparameter
            prioritized_replay_eps (float): replay eps hyperparameter
            train_batch_size (int): size of batches to learn on
            rollout_fragment_length (int): size of batches to sample from
                workers.
            num_replay_buffer_shards (int): number of actors to use to store
                replay samples
            max_weight_sync_delay (int): update the weights of a rollout worker
                after collecting this number of timesteps from it
            debug (bool): return extra debug stats
            batch_replay (bool): replay entire sequential batches of
                experiences instead of sampling steps individually
        """
        PolicyOptimizer.__init__(self, workers)

        self.debug = debug
        self.batch_replay = batch_replay
        self.replay_starts = learning_starts
        self.prioritized_replay_beta = prioritized_replay_beta
        self.prioritized_replay_eps = prioritized_replay_eps
        self.max_weight_sync_delay = max_weight_sync_delay

        self.learner = LearnerThread(self.workers.local_worker())
        self.learner.start()

        if self.batch_replay:
            replay_cls = BatchReplayActor
        else:
            replay_cls = ReplayActor
        self.replay_actors = create_colocated(replay_cls, [
            num_replay_buffer_shards,
            learning_starts,
            buffer_size,
            train_batch_size,
            prioritized_replay_alpha,
            prioritized_replay_beta,
            prioritized_replay_eps,
        ], num_replay_buffer_shards)

        # Stats
        self.timers = {
            k: TimerStat()
            for k in [
                "put_weights", "get_samples", "sample_processing",
                "replay_processing", "update_priorities", "train", "sample"
            ]
        }
        self.num_weight_syncs = 0
        self.num_samples_dropped = 0
        self.learning_started = False

        # Number of worker steps since the last weight update
        self.steps_since_update = {}

        # Otherwise kick of replay tasks for local gradient updates
        self.replay_tasks = TaskPool()
        for ra in self.replay_actors:
            for _ in range(REPLAY_QUEUE_DEPTH):
                self.replay_tasks.add(ra, ra.replay.remote())

        # Kick off async background sampling
        self.sample_tasks = TaskPool()
        if self.workers.remote_workers():
            self._set_workers(self.workers.remote_workers())

    @override(PolicyOptimizer)                                          [docs]
    def step(self):
        assert self.learner.is_alive()
        assert len(self.workers.remote_workers()) > 0
        start = time.time()
        sample_timesteps, train_timesteps = self._step()
        time_delta = time.time() - start
        self.timers["sample"].push(time_delta)
        self.timers["sample"].push_units_processed(sample_timesteps)
```

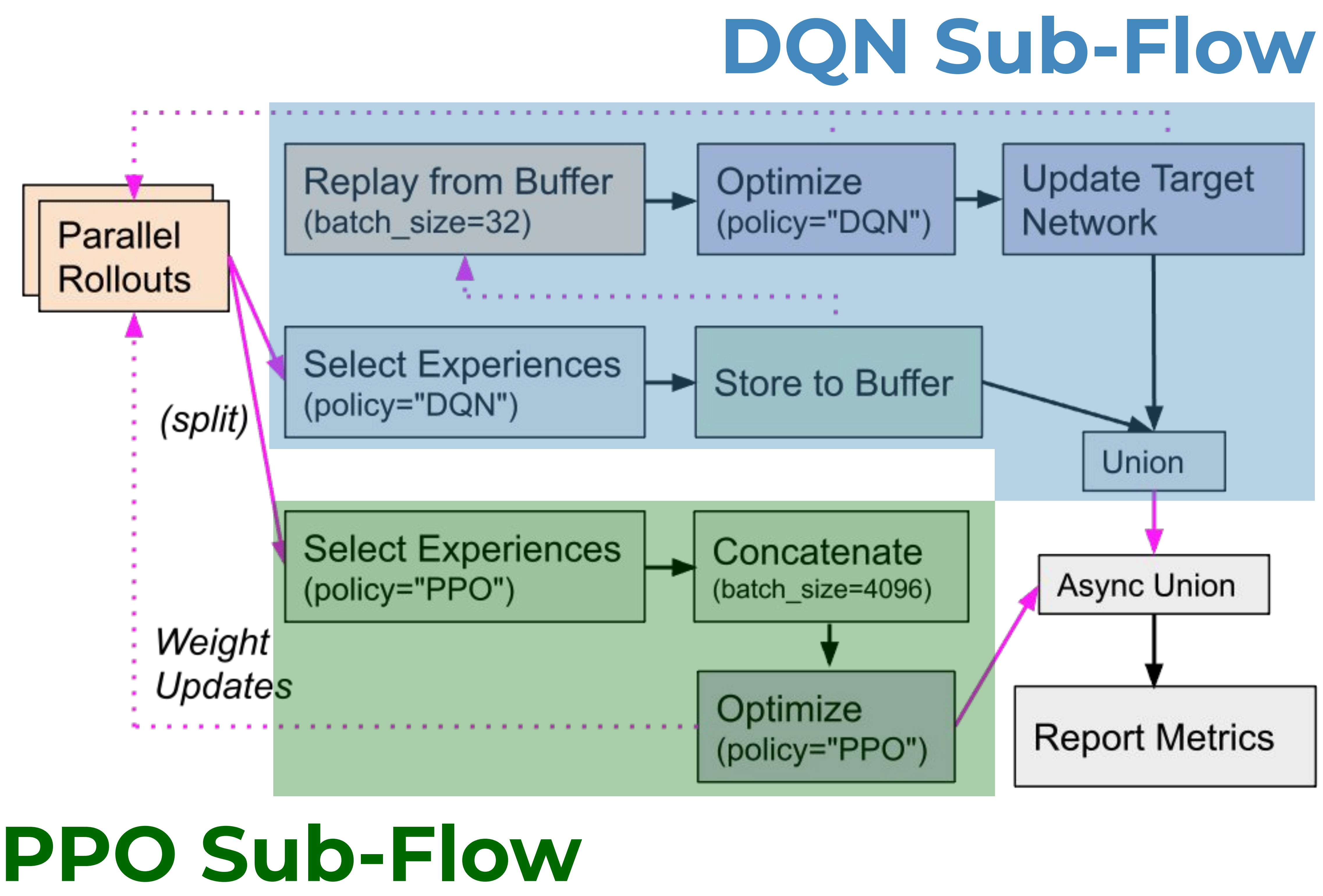
```python
        assert len(self.workers.remote_workers()) > 0
        start = time.time()
        sample_timesteps, train_timesteps = self._step()
        time_delta = time.time() - start
        self.timers["sample"].push(time_delta)
        self.timers["sample"].push_units_processed(sample_timesteps)
        if train_timesteps > 0:
            self.learning_started = True
        if self.learning_started:
            self.timers["train"].push(time_delta)
            self.timers["train"].push_units_processed(train_timesteps)
        self.num_steps_sampled += sample_timesteps
        self.num_steps_trained += train_timesteps

    @override(PolicyOptimizer)                                          [docs]
    def stop(self):
        for r in self.replay_actors:
            r.__ray_terminate__.remote()
        self.learner.stopped = True

    @override(PolicyOptimizer)                                          [docs]
    def reset(self, remote_workers):
        self.workers.reset(remote_workers)
        self.sample_tasks.reset_workers(remote_workers)

    @override(PolicyOptimizer)                                          [docs]
    def stats(self):
        replay_stats = ray_get_and_free(self.replay_actors[0].stats.remote(
            self.debug))
        timing = {
            "{}_time_ms".format(k): round(1000 * self.timers[k].mean, 3)
            for k in self.timers
        }
        timing["learner_grad_time_ms"] = round(
            1000 * self.learner.grad_timer.mean, 3)
        timing["learner_dequeue_time_ms"] = round(
            1000 * self.learner.queue_timer.mean, 3)
        stats = {
            "sample_throughput": round(self.timers["sample"].mean_throughput,
                                       3),
            "train_throughput": round(self.timers["train"].mean_throughput, 3),
            "num_weight_syncs": self.num_weight_syncs,
            "num_samples_dropped": self.num_samples_dropped,
            "learner_queue": self.learner.learner_queue_size.stats(),
            "replay_shard_0": replay_stats,
        }
        debug_stats = {
            "timing_breakdown": timing,
            "pending_sample_tasks": self.sample_tasks.count,
            "pending_replay_tasks": self.replay_tasks.count,
        }
        if self.debug:
            stats.update(debug_stats)
        if self.learner.stats:
            stats["learner"] = self.learner.stats
        return dict(PolicyOptimizer.stats(self), **stats)

    # For https://github.com/ray-project/ray/issues/2541 only
    def _set_workers(self, remote_workers):
        self.workers.reset(remote_workers)
        weights = self.workers.local_worker().get_weights()
        for ev in self.workers.remote_workers():
            ev.set_weights.remote(weights)
            self.steps_since_update[ev] = 0
            for _ in range(SAMPLE_QUEUE_DEPTH):
                self.sample_tasks.add(ev, ev.sample_with_count.remote())

    def _step(self):
        sample_timesteps, train_timesteps = 0, 0
        weights = None

        with self.timers["sample_processing"]:
            completed = list(self.sample_tasks.completed())
            # First try a batched ray.get().
            ray_error = None
            try:
```

```python
        with self.timers["sample_processing"]:
            completed = list(self.sample_tasks.completed())
            # First try a batched ray.get().
            ray_error = None
            try:
                counts = {
                    i: v
                    for i, v in enumerate(
                        ray_get_and_free([c[1][1] for c in completed]))
                }
            # If there are failed workers, try to recover the still good ones
            # (via non-batched ray.get()) and store the first error (to raise
            # later).
            except RayError:
                counts = {}
                for i, c in enumerate(completed):
                    try:
                        counts[i] = ray_get_and_free(c[1][1])
                    except RayError as e:
                        logger.exception(
                            "Error in completed task: {}".format(e))
                        ray_error = ray_error if ray_error is not None else e

            for i, (ev, (sample_batch, count)) in enumerate(completed):
                # Skip failed tasks.
                if i not in counts:
                    continue

                sample_timesteps += counts[i]
                # Send the data to the replay buffer
                random.choice(
                    self.replay_actors).add_batch.remote(sample_batch)

                # Update weights if needed.
                self.steps_since_update[ev] += counts[i]
                if self.steps_since_update[ev] >= self.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 self.learner.weights_updated:
                        self.learner.weights_updated = False
                        with self.timers["put_weights"]:
                            weights = ray.put(
                                self.workers.local_worker().get_weights())
                    ev.set_weights.remote(weights)
                    self.num_weight_syncs += 1
                    self.steps_since_update[ev] = 0

                # Kick off another sample request.
                self.sample_tasks.add(ev, ev.sample_with_count.remote())

            # Now that all still good tasks have been kicked off again,
            # we can throw the error.
            if ray_error:
                raise ray_error

        with self.timers["replay_processing"]:
            for ra, replay in self.replay_tasks.completed():
                self.replay_tasks.add(ra, ra.replay.remote())
                if self.learner.inqueue.full():
                    self.num_samples_dropped += 1
                else:
                    with self.timers["get_samples"]:
                        samples = ray_get_and_free(replay)
                    # Defensive copy against plasma crashes, see #2610 #3452
                    self.learner.inqueue.put((ra, samples and samples.copy()))

        with self.timers["update_priorities"]:
            while not self.learner.outqueue.empty():
                ra, prio_dict, count = self.learner.outqueue.get()
                ra.update_priorities.remote(prio_dict)
                train_timesteps += count

        return sample_timesteps, train_timesteps
```

Evaluation: Composing Multiple Workflows



Evaluation: Multi-Agent Training



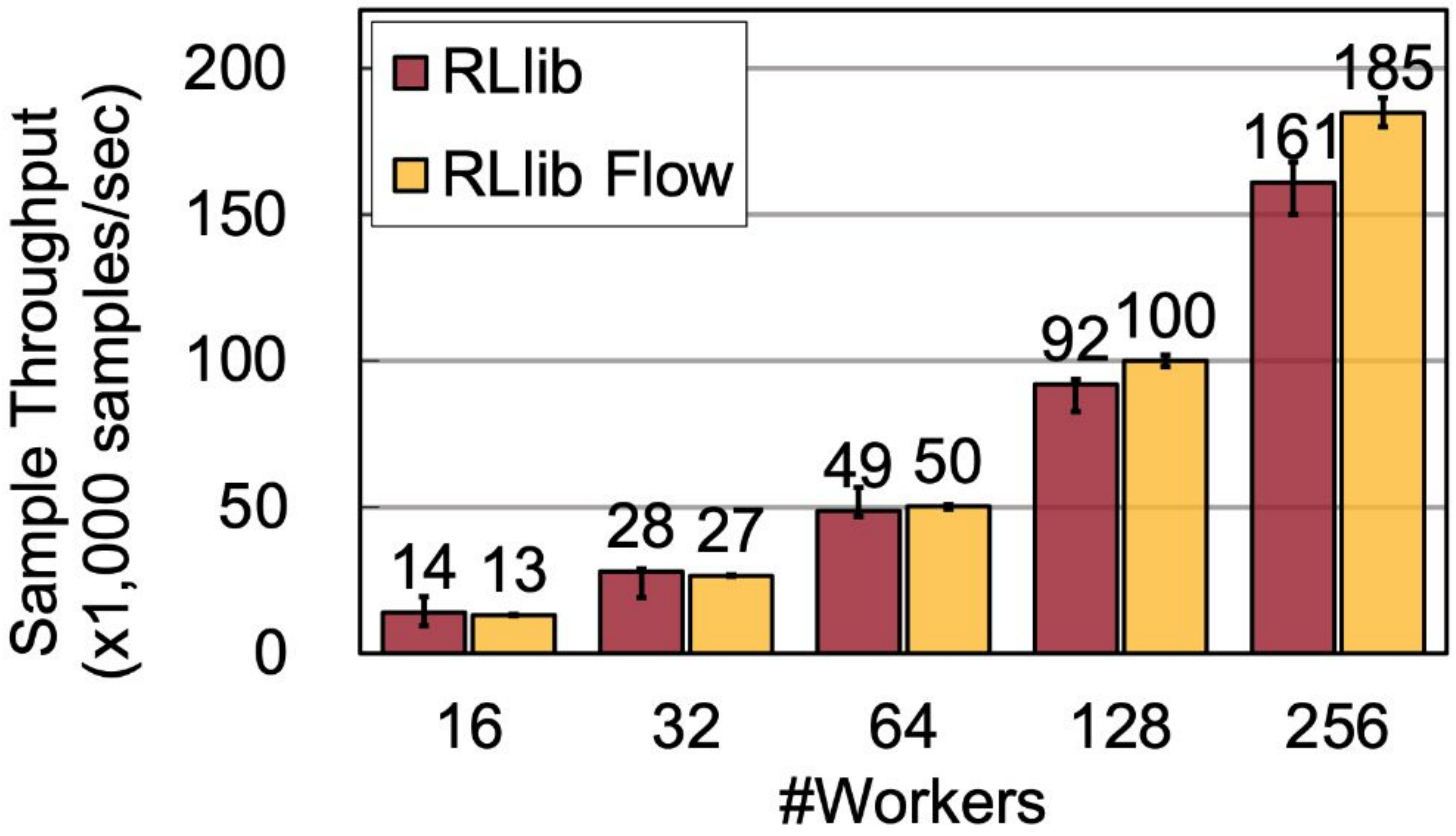

```python
# type: List[RolloutActor]
workers = create_rollout_workers()
# type: Iter[Rollout], Iter[Rollout]
r1, r2 = ParallelRollouts(workers).split()
# type: Iter[TrainStats], Iter[TrainStats]
ppo_op = ppo_plan(
    Select(r1, policy="PPO"), workers)
dqn_op = dqn_plan(
    Select(r2, policy="DQN"), workers)
# type: Iter[Metrics]
return ReportMetrics(
    Union(ppo_op, dqn_op), workers)
```

Evaluation: Readability

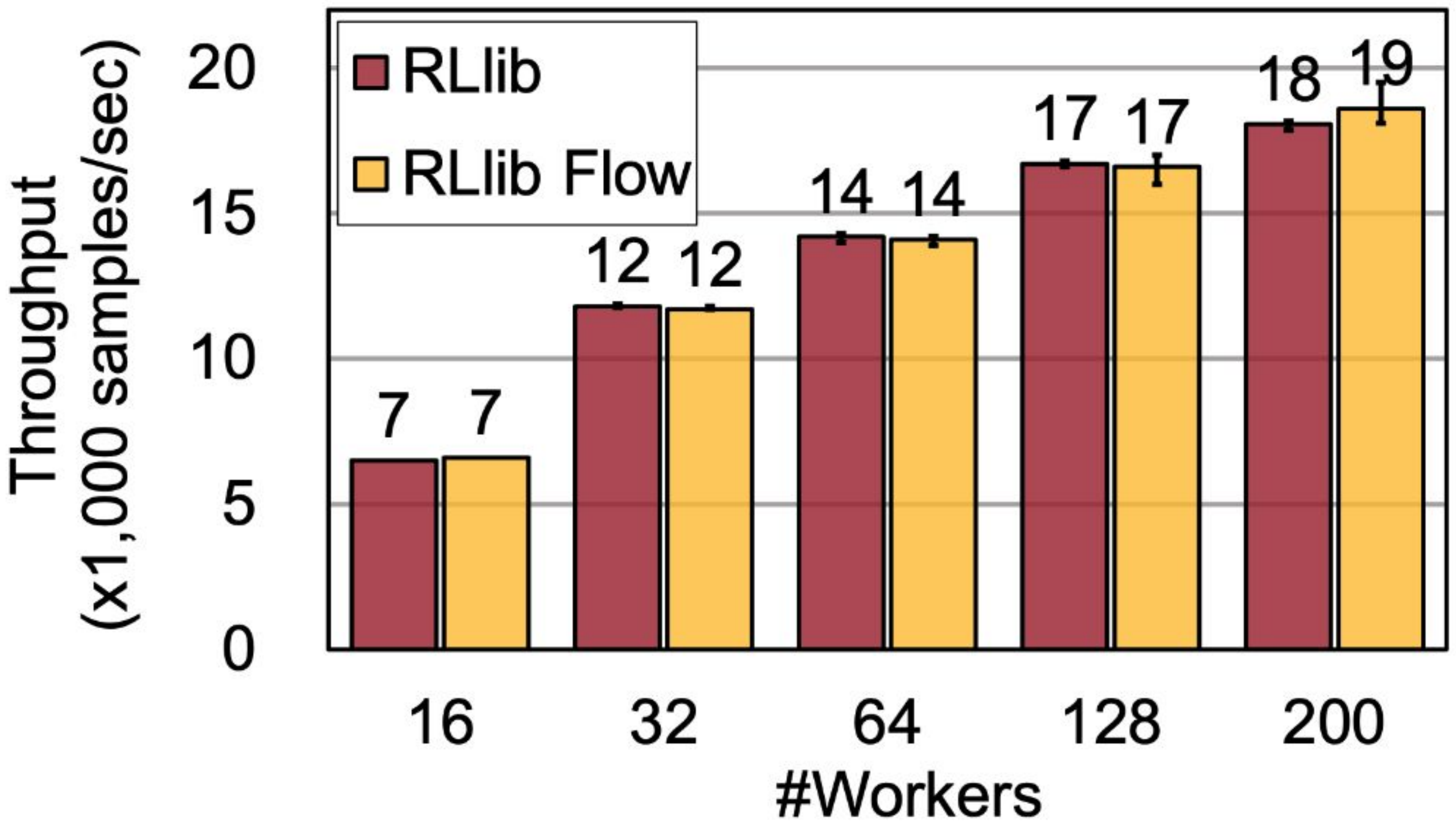
- Lines of code saved for RLlib algorithms

	RLlib	RLlib Flow	+shared	Ratio
A3C	87	11	52	1.6-9.6×
A2C	154	25	50	3.1-6.1×
DQN	239	87	139	1.7-2.7×
PPO	386	79	225	1.7-4.8×
Ape-X	250	126	216	1.1-1.9×
IMPALA	694	89	362	1.9-7.8×
MAML	370*	136	136	2.7×

Performance against RLlib



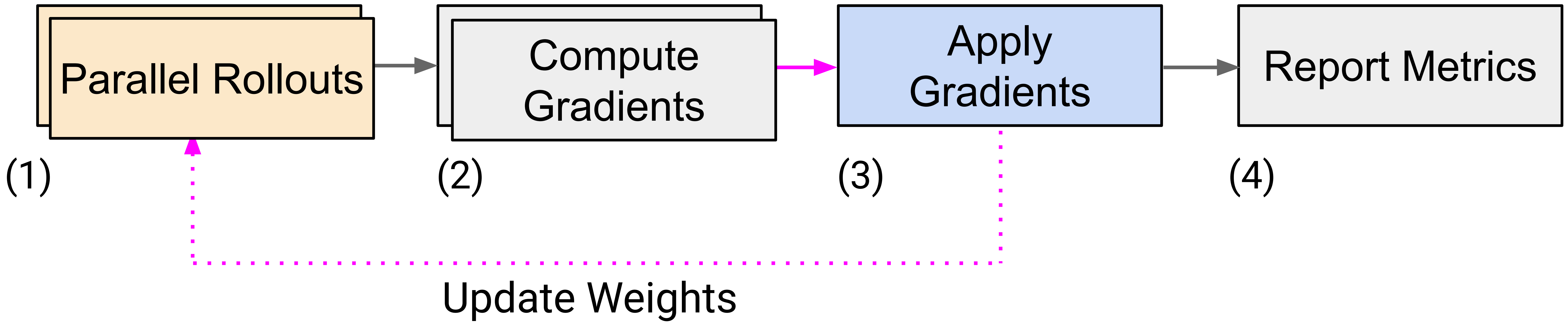
(a) Sample efficiency on CartPole (Dummy)



(b) Training throughput on Atari (IMPALA)

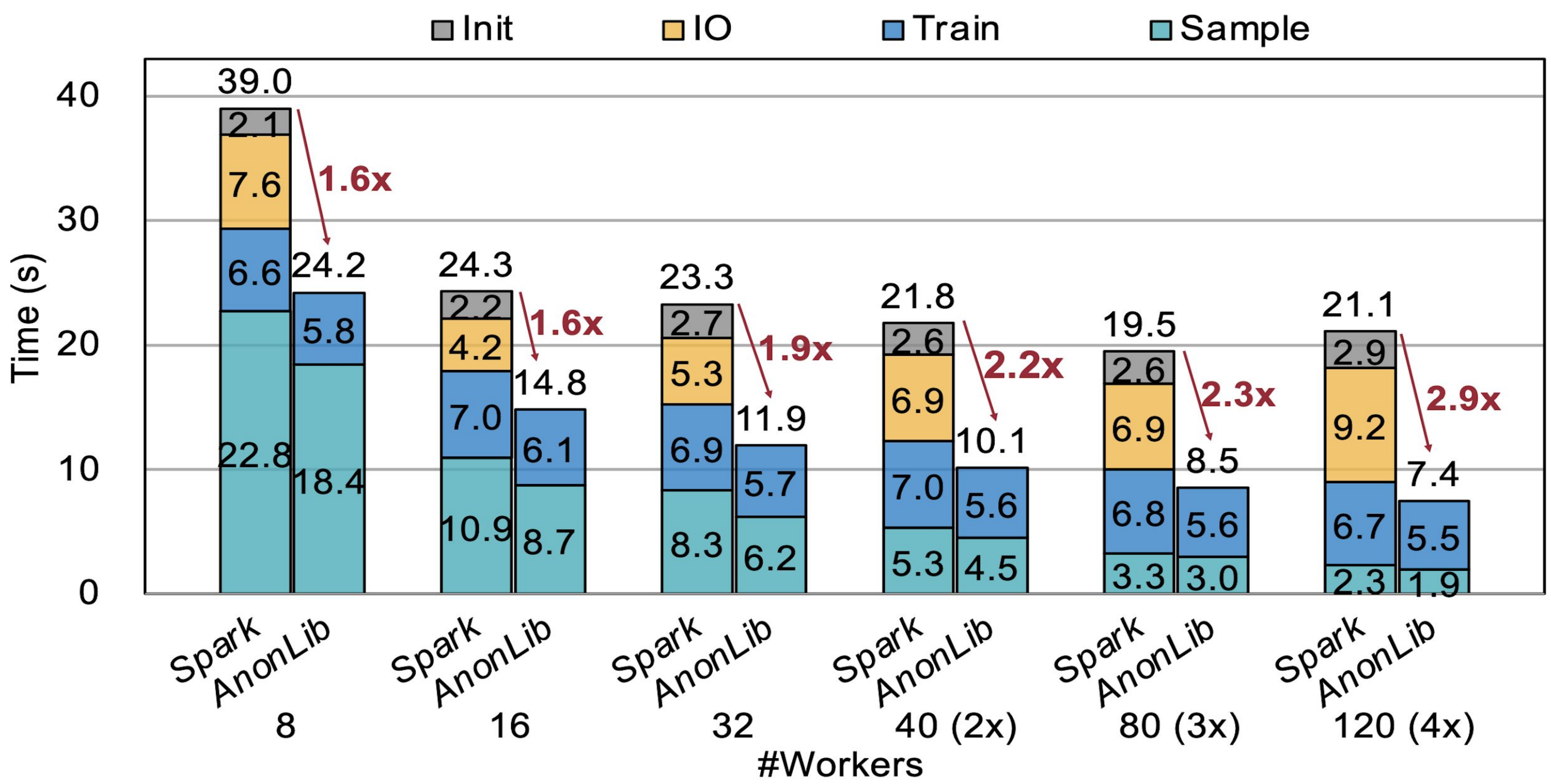
- The abstraction of RLlib Flow **does not introduce overhead**

Reinforcement Learning vs Data Streaming



- Asynchronous Dependencies (pink): no deterministic ordering
- Message Passing (pink dotted): update upstream operator state
- Consistency and Durability: less strict requirements

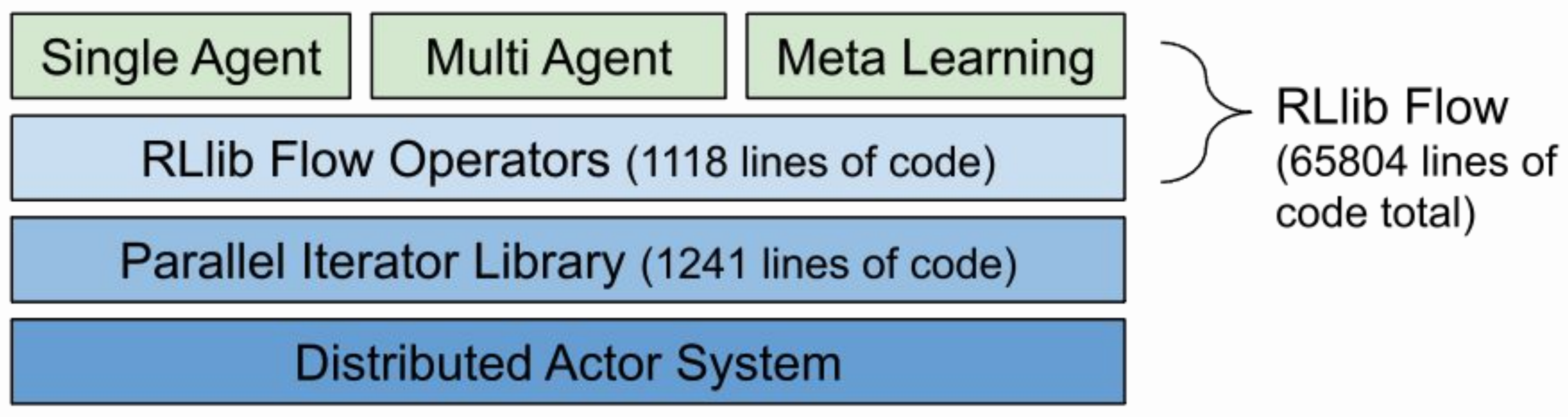
Performance against Spark Streaming



- **Lower-overhead** than streaming frameworks -- take advantage of RL requirements vs. data processing

RLlib Flow

Distributed Reinforcement Learning is a Dataflow Problem



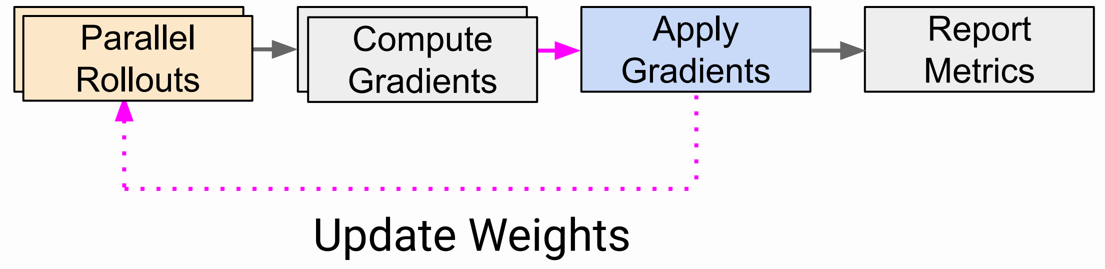
Architecture of RLlib Flow

RLlib Flow Operators:

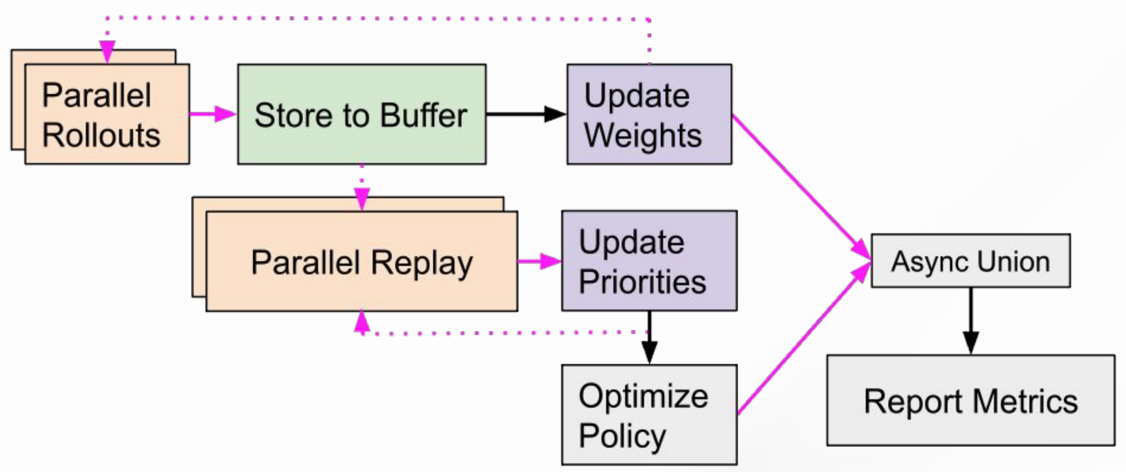
1. Creation & Message Passing
2. Transformation
3. Sequencing
4. Concurrency

	RLlib	RLlib Flow	+shared	Ratio
A3C	87	11	52	1.6-9.6×
A2C	154	25	50	3.1-6.1×
DQN	239	87	139	1.7-2.7×
PPO	386	79	225	1.7-4.8×
Ape-X	250	126	216	1.1-1.9×
IMPALA	694	89	362	1.9-7.8×
MAML	370*	136	136	2.7×

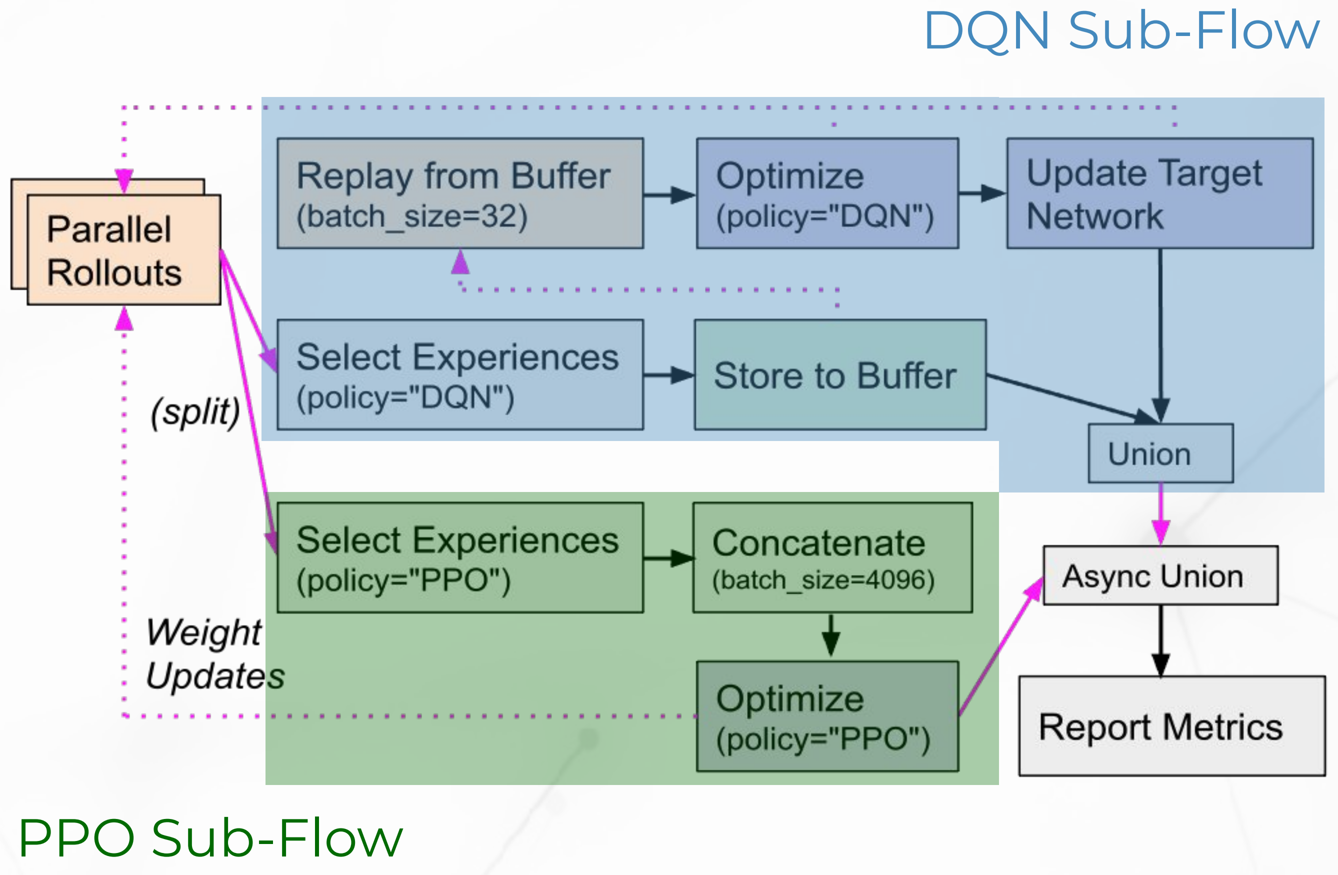
Lines of Code



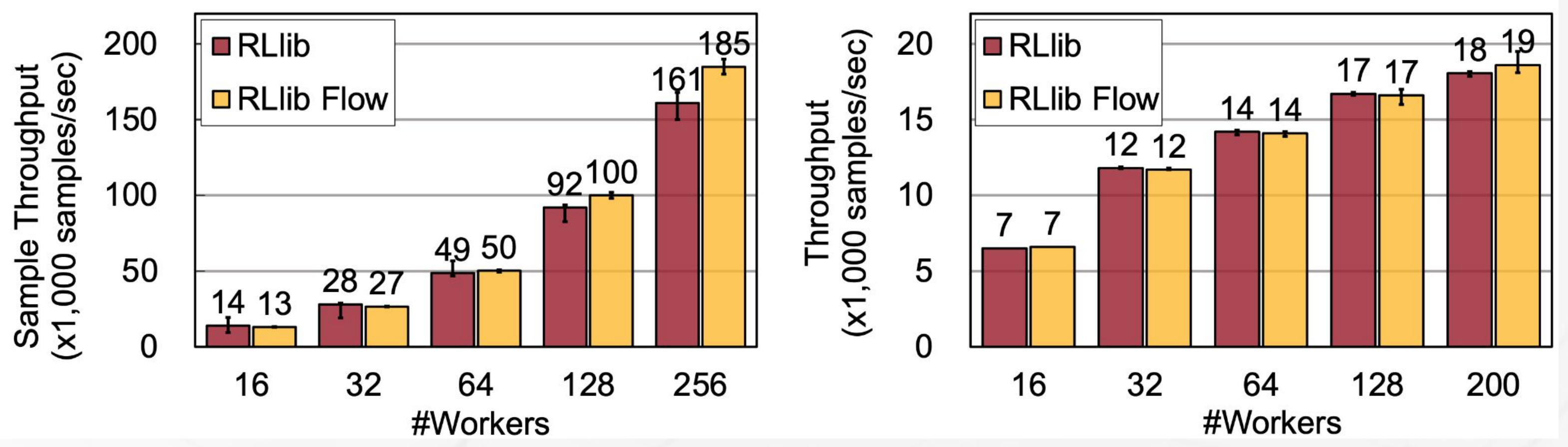
Dataflow of A3C



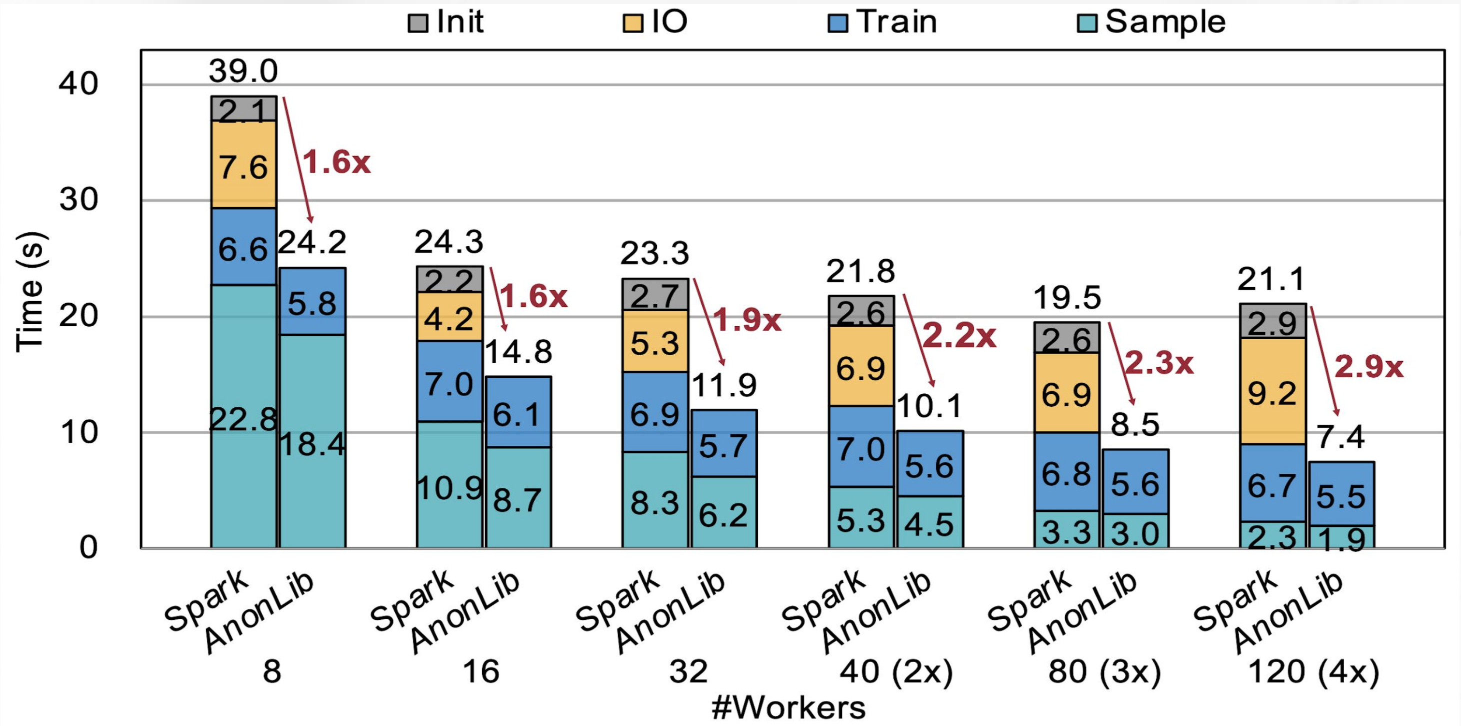
Dataflow of Ape-X



Dataflow of Multi-Agent



Benchmark



Comparison to Spark

Correspondence to:
Eric Liang<ericliang@berkeley.edu>,
Ion Stoica<istoica@berkeley.edu>