MSRL: Distributed Reinforcement Learning with Dataflow Fragments

Huanzhou Zhu, *Imperial College London;* Bo Zhao, *Imperial College London and Aalto University;* Gang Chen, Weifeng Chen, Yijie Chen, and Liang Shi, *Huawei Technologies Co., Ltd.;* Yaodong Yang, *Peking University;* Peter Pietzuch, *Imperial College London;* Lei Chen, *Hong Kong University of Science and Technology*

https://www.usenix.org/conference/atc23/presentation/zhu-huanzhou

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Open access to the Proceedings of the 2023 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

MSRL: Distributed Reinforcement Learning with Dataflow Fragments

Huanzhou Zhu*
Imperial College London

Bo Zhao*
Imperial College London and Aalto University

Gang Chen
Huawei Technologies Co., Ltd.

Weifeng Chen
Huawei Technologies Co., Ltd.

Yijie Chen
Huawei Technologies Co., Ltd.

Liang Shi
Huawei Technologies Co., Ltd.

Yaodong Yang
Peking University

Peter Pietzuch
Imperial College London

Lei Chen
Hong Kong University of Science and Technology

Abstract

A wide range of *reinforcement learning* (RL) algorithms have been proposed, in which agents learn from interactions with a simulated environment. Executing such RL training loops is computationally expensive, but current RL systems fail to support the training loops of different RL algorithms efficiently on GPU clusters: they either hard-code algorithm-specific strategies for parallelization and distribution; or they accelerate only parts of the computation on GPUs (e.g., DNN policy updates). We observe that current systems lack an abstraction that decouples the *definition* of an RL algorithm from its *strategy* for distributed execution.

We describe MSRL, a distributed RL training system that uses the new abstraction of a *fragmented dataflow graph* (FDG) to execute RL algorithms in a flexible way. An FDG is a heterogeneous dataflow representation of an RL algorithm, which maps functions from the RL training loop to independent parallel dataflow *fragments*. Fragments account for the diverse nature of RL algorithms: each fragment can execute on a different device using its own low-level dataflow implementation, e.g., an operator graph of a DNN engine, a CUDA GPU kernel, or a multi-threaded CPU process. At deployment time, a *distribution policy* governs how fragments are mapped to devices, without changes to the algorithm implementation. Our experiments show that MSRL exposes trade-offs between different execution strategies, while surpassing the performance of existing RL systems.

1 Introduction

Reinforcement learning (RL) solves decision-making problems by having agents learn policies – typically represented as deep neural networks (DNNs) – on how to act in an environment [51]. RL has achieved remarkable outcomes: in game play, AlphaGo [49] defeated a world champion in the Go board game; in biology, AlphaFold [21] predicts three-dimensional structures for protein folding; in robotics, RL allows robots to perform dexterous manipulation without human intervention [15]; and the ChatGPT chatbot [41] uses a reinforcement step with PPO [47] to fine-tune its model.

Such advances in RL, however, come with high computational demands: AlphaStar trained 12 agents on 384 TPUs and 1,800 CPUs for 44 days to achieve grandmaster level in StarCraft II game play [54]; OpenAI Five trained to play Dota 2 games for 10 months with 1,536 GPUs and 172,800 CPUs [3].

Existing RL systems (e.g., SEED RL [8], Acme [18], Ray [34], RLlib [25], Podracer [16]) are therefore optimized for specific types of RL algorithms and the structure of their RL training loops. In particular, systems hardcode a strategy for parallelizing and distributing the RL computation:

Parallelization. Most RL systems only accelerate the DNN computation on GPUs or TPUs [8, 18, 25] using current DNN engines (e.g., PyTorch [42], TensorFlow [14], and MindSpore [19]). Other parts of RL algorithms (e.g., action generation, environment execution, and trajectory sampling) are executed as sequential Python functions on worker nodes, potentially becoming performance bottlenecks.

Some systems try to accelerate more parts of RL training loops: Podracer [16] uses the JAX [12] compilation framework to vectorize Python implementations of RL algorithms; WarpDrive [23] implement the entire RL training loop using CUDA on a GPU; and RLlib Flow [26] uses a set of parallel dataflow operators [58] to express an RL training loop. All of these approaches, however, require users to rewrite the *complete* RL algorithm (e.g., agents, learners, and environments) using a single API with a fixed set of dataflow operators.

Distribution. When distributing computation, current RL systems allocate algorithmic components (e.g., actors and learners) to workers in a fixed way: SEED RL [8] assumes that learners perform policy inference and training on TPUs, and actors execute on CPUs; Acme [18] only distributes actors and maintains a single learner; and TLeague [50] distributes learners but co-locates environments with actors on CPU workers. As we shown in §6, such decisions are algorithm-specific: since different algorithms deployed on a given set of resources exhibit diverse bottlenecks, a single distribution strategy cannot exhibit the best performance in all cases.

*Equal contribution.

We observe that the above challenges come from a lack of separation between the *definition* of an RL algorithm and how it is *executed* by the system. For example, many RL systems allow users to define RL algorithms as a set of Python functions for agents, learners, and environments. The system then directly invokes the implementation of e.g., an agent's `act()` function to produce new actions for the environment. While this simplifies system implementation, it removes control from the system regarding how algorithmic components are parallelized or distributed at deployment time.

DNN training systems use intermediate representations (IRs), which are compiled to target devices for execution, to decouple DNN definition from execution [1,6,56]. Such approaches, however, assume a homogeneous training computation (forward/backpropagation over differentiable DNNs [2]), which can be expressed by a fixed set of computational operators over tensor types. In contrast, the space of RL algorithms exhibits more heterogeneity in terms of the computation performed by algorithmic components (agents, actors, learners, polices, environments, leaderboards), their exchanged data (observations, actions, policy updates) and communication patterns (one-to-one, one-to-many, all-reduce).

Our goal is to explore a new design for an RL training system that requires users to define an RL algorithm only once. At deployment, the system then supports (i) the execution of arbitrary parts of the RL computation on parallel devices (GPUs and CPUs); and (ii) the deployment of parts of the computation on distributed workers.

We describe **MSRL**, a distributed RL system that achieves this by decoupling the specification of a RL algorithm from its execution through the abstraction of a *fragmented dataflow graph* (FDG). Unlike dataflow approaches of DNN and data analytics systems, an FDG does not enforce a single uniform dataflow representation, which is challenging for diverse RL algorithms. Instead, it allows different components of an RL algorithm to have bespoke GPU or CPU implementations, chosen by the user at deployment time.

In summary, MSRL's design makes three contributions:

(1) Fragmented dataflow graphs (§3). From the RL algorithm implementation, MSRL constructs an FDG, which consists of independent *fragments*. Each fragment can have its own dataflow representation (e.g., DNN operators, CUDA, or Python) targeting GPUs or CPUs. MSRL then maps instances of fragments to devices at deployment time.

To obtain fragments, MSRL statically analyzes the RL algorithm implementation to group functions into fragments. By default, the boundaries between fragments are chosen based on the algorithmic components of the RL algorithm. Since fragments are deployed on different devices, MSRL synthesizes appropriate communication operators that allows fragments to exchange data.

(2) API with distribution policies (§4). Users specify an RL algorithm by implementing its algorithmic components as Python functions in a traditional way. The implementation makes no assumptions about how the algorithm will be executed: all runtime interactions between components are managed by calls to MSRL APIs. A separate *deployment configuration* defines the devices available for execution.

Since FDGs separate algorithm implementations from execution, MSRL can apply different *distribution policies* to govern how fragments are mapped to devices. MSRL supports distribution policies, which subsume the hard-coded distribution strategies of current RL systems: e.g., a policy can distribute multiple actors to scale environment interaction (like Acme [18]); distribute actors and move policy inference to learners (like SEED RL [8]); distribute both actors and learners (like Sebulba [16]); or represent the full RL training loop on a GPU (like WarpDrive [23] and Anakin [16]).

When a user changes the algorithm configuration, its hyper-parameters or deployment resources, they can also switch between distribution policy to maintain high training efficiency without having to change the RL algorithm implementation.

(3) Heterogeneous fragment execution (§5). For execution, MSRL deploys hardware-specific implementations of fragments on CPUs and GPUs. MSRL supports different fragment implementations: CPU implementations use regular (multi-process) Python code, and GPU implementations are generated as compiled computational graphs for DNN engines (e.g., MindSpore or TensorFlow) if a fragment is implemented using operators, or are implemented directly in CUDA.

MSRL optimizes co-located fragments on the same worker: it fuses data-parallel fragments for more efficient execution by batching data items (e.g., tensors) and using single-instruction-multiple-data (SIMD) execution.

We evaluate MSRL experimentally and show that MSRL's abstraction supports flexible training across different RL algorithm without compromising training performance compared to current hardcoded RL training systems: MSRL scales to 64 GPUs and outperforms the Ray distributed RL system [34] by up to 3×. By switching between distribution policies, MSRL can improve the training time of the PPO RL algorithm by up to 2.4× as hyper-parameters, network properties or hardware resources change.

2 Distributed Reinforcement Learning

Next we give background on RL algorithms (§2.1), discuss the requirements for RL training (§2.2), and survey the design space of existing RL systems (§2.3).

2.1 Reinforcement learning

Reinforcement learning (RL) solves a sequential decision-making problem in which an *agent* operates in an *environment*. The agent's goal is to learn a *policy* that maximizes the cumulative *reward* based on the feedback from the environment (see Fig. 1). RL training performs three steps: ❶ *policy inference*: an agent obtains an action by performing inference on a policy; ❷ *environment execution*: the environment executes the action, generating *trajectories* of $\langle state, reward \rangle$

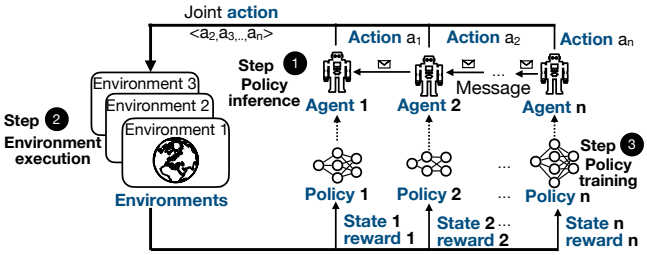


Fig. 1: RL training loop with multiple agents

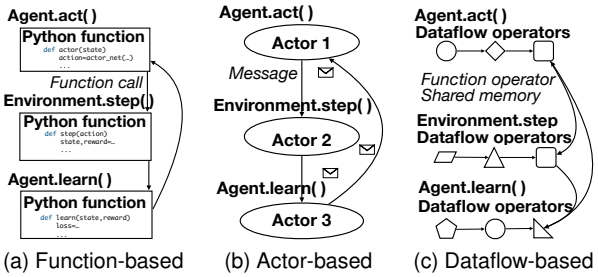


Fig. 2: Types of RL system designs

pairs; and ❸ *policy training*: the agent improves the policy by adapting it based on the reward.

RL algorithms are diverse in nature, falling into three categories: (1) *value-based* algorithms (e.g., DQN [33]) use a DNN to approximate a value function that predicts the expected return of actions. Agents then select actions based on these estimated values; (2) *policy-based* algorithms (e.g., Reinforce [55]) directly learn a parameterized policy – approximated by a DNN – for selecting actions without a value function. Agents use batched trajectories to train the policy by updating its parameters to maximize the reward; and (3) *actor–critic* algorithms (e.g., PPO [47], DDPG [27], A2C [32]) combine the two by learning a policy that selects actions and a value function that evaluates them.

Multi-agent reinforcement learning (MARL) employs multiple agents, each optimizing its own cumulative reward when interacting with the environment or other agents (see Fig. 1). A3C [32] executes agents asynchronously on separate environment instances; MAPPO [57] extends PPO to a multi-agent setting in which agents share a global parameterized policy.

2.2 Requirements for distributed RL systems

RL algorithms explore large spaces of actions, states and DNN parameters, which grow exponentially with the number of agents [37]. RL systems must thus exploit the parallelism of GPUs and scale computation to many worker nodes.

Due to the diverse computational patterns exhibited by different RL algorithms, there is no single strategy for parallelization and distribution that is optimal for all RL algorithms, e.g., in terms of both achieving the lowest iteration time and scaling to the most workers. Bottlenecks during training depend on the specific algorithm, the training workloads and the employed hardware resources: e.g., our experiments show that, for PPO [47], environment execution (❷) takes up to 98% of execution time; for MuZero [46], a large MARL algorithm with many agents, instead 97% of time is spent on policy inference and training (❶+❸).

Therefore, there are many proposals how to parallelize and distribute RL computation: e.g., in single-agent RL, environment execution (❷ in Fig. 1), policy inference and training (❶+❸) can be distributed across workers [8, 16, 16, 18, 34]; in MARL, agents can be distributed [25, 26, 34, 45] and exchange training state [29, 39]. Environment instances can execute in parallel [16, 32] or be distributed [7].

Instead of committing to one approach for parallelizing and distributing the RL computation, an RL system should provide the flexibility to change its execution approach based on the workload. This leads us to the following requirements:

(1) Execution abstraction. The system should have a flexible execution abstraction for parallelizing and distributing computation, unencumbered by how the algorithm is defined.

(2) Distribution strategies. The system should support multiple strategies for distributing RL computation. Users should be permitted to switch between strategies based on the training workload, without changes to the algorithm.

(3) Acceleration support. The system should exploit the parallelism of GPUs and CPUs, accelerating not just policy training and inference (❶+❸) but the full RL training loop, including environment execution (❷) [23].

(4) Algorithm abstraction. The system should expose familiar APIs to users for defining RL algorithms and their training loops, structured around algorithmic components [9, 13, 22], such as agents, actors, learners, policies, environments, etc.

2.3 Design space of existing RL systems

We analyze the design space of RL systems. Existing designs fit into three types (see Fig. 2):

(a) Function-based RL systems are the most common type. They express RL algorithms typically as Python functions, executed directly by workers (see Fig. 2a). The RL training loop is implemented through direct function calls. For example, Acme [18] and SEED RL [8] organize algorithms as actor/learner functions; RLGraph [45] uses a component abstraction, and users register Python callbacks to define functionality. Distributed execution is delegated to backend engines, e.g., TensorFlow [14], PyTorch [42], Ray [34].

(b) Actor-based RL systems execute algorithms as a set of (programming language) actors deployed on worker nodes (see Fig. 2b). For example, Ray [34] uses an actor model to define tasks, which are distributed among nodes using remote calls. Defining control flow in an actor model, however, is burdensome. To overcome this issue, RLlib [25] adds logically centralized control on top of Ray. Similarly, MALib [61] offers a higher level abstractions for population-based MARL algorithms (e.g., PSRO [35]) on Ray.

(c) Dataflow-based RL systems define algorithms through a

Type	System	(1) Execution	(2) Distribution	(3) Acceleration	(4) Algorithm
Function-based	SEED RL [8]	Python functions	environment only	DNNs only	actor/learner/env
	Acme [18]	Python classes			
	RLGraph [45]		delegated to backend		agent
Actor-based	Ray [34]	task (stateless) actor (stateful)	scheduler/RPC	DNNs only	Python functions with Ray API [34]
	RLlib [25]				
	MALib [61]				agent/actor/learner/env
Dataflow-based	Podracer [16]	JIT-compiled by JAX [12]	hardcoded	funcs/DNNs/envs	JAX [12] API
	RLlib Flow [26]	predefined dataflow operators	dataflow operators/ Ray tasks [34]	DNNs only	operator API
	WarpDrive [23]	GPU thread blocks	—	CUDA kernels	CUDA API
Fragmented dataflow	**MSRL**	heterogeneous fragments	any fragment	funcs/operators/ DNNs/envs	agent/actor/learner/env

Tab. 1: Design space of distributed RL systems

set of data-parallel operators, implemented by GPU kernels or distributed tasks (see Fig. 2c). Users must express the complete RL training loop using operators APIs. For example, Podracer [16] uses JAX [12] to compile vectorized Python to TPU kernels. RLlib Flow [26] provides Spark-like dataflow operators on top of Ray. WarpDrive [23] executes RL training loops implemented in CUDA using GPU thread blocks.

Tab. 1 considers how well these approaches satisfy the four requirements from §2.2:

(1) Execution abstraction. Function- and actor-based systems execute RL algorithms directly through implemented (Python) functions and user-defined language actors, respectively. This prevents systems from applying optimizations how RL algorithms are parallelized or distributed. In contrast, dataflow-based systems execute computation using operators [16, 26] or CUDA kernels [23]. This allows for execution optimizations, but algorithm implementations are restricted by the supported set of operators.

(2) Distribution strategies. Most function-based systems only support a hardcoded strategy, e.g., one that distributes actors to parallelize the environment execution (❶+❷ in Fig. 1) with a single learner. In actor-based systems, a scheduler assigns stateful actors and stateless tasks to workers, and users have no control over the distribution approach.

Similarly, existing dataflow-based systems only support fixed policies how dataflow operators are assigned to workers: *Anakin* [16] co-hosts an environment and an agent on each TPU core; *Sebulba* distributes the environment, learners and actors on different TPUs; and RLlib Flow [26] shards dataflow operators across distributed Ray actors.

(3) Acceleration support. Most RL systems only accelerate DNN policy inference and training (❶+❸). Some dataflow-based systems (e.g., Podracer [16] and WarpDrive [23]) also accelerate other parts of training, requiring bespoke dataflow implementations: e.g., Podracer accelerates environment execution (❷) on TPU cores; WarpDrive executes the entire RL training loop (❶–❸) on a single GPU using CUDA.

(4) Algorithm abstraction. Function-based RL systems provide intuitive actor/learner/env APIs. Actor-based RL systems exhibit harder-to-use low-level APIs for distributed components (e.g., Ray's `get`/`wait`/`remote` [34]) and must rely on high-level libraries (e.g., RLlib's `PolicyOptimizer` API [25]) to bridge the gap. Dataflow-based systems come with their own dataflow operators, requiring the rewriting of a complete RL training loop. For example, JAX [12] require users to express RL algorithms in terms of the `vmap` and `pmpa` operators for vectorization and single-program multiple-data (SPMD) parallelism, respectively.

We note that there is an opportunity to combine the usability of a function-based algorithmic abstraction, which allow users to express RL algorithms naturally using algorithmic components, with the acceleration potential of dataflow-based approaches. Such a design, however, requires a new execution abstraction, which also retains the flexibility of supporting different distribution strategies.

3 Fragmented Dataflow Graphs

We now describe the dataflow abstraction that we use to represent the heterogenous computation of RL algorithms and to map it to various devices for execution.

3.1 Overview

Our aim is to take an arbitrary RL training loop of a single- or multi-agent RL algorithm (Fig. 1) and translate it to a dataflow representation. The RL system can then use the dataflow representation to parallelize and distribute the computation across heterogeneous devices. We observe that RL training loops combine different types of computations: e.g., actors decide on an action to carry out based on inference results from the DNN policy, obtaining trajectories first; learners update the DNN policy using a DNN training algorithm; and environments execute steps in e.g., a physics simulator, returning trajectories based on the current simulation state.

Unlike existing dataflow models for DNN computation [1, 6, 19], this heterogeneity of computation makes it challenging to impose a single uniform dataflow model that prescribes a set of computational operators and a single data representation (e.g., tensors) between them. Instead, we adopt a *heteroge-*

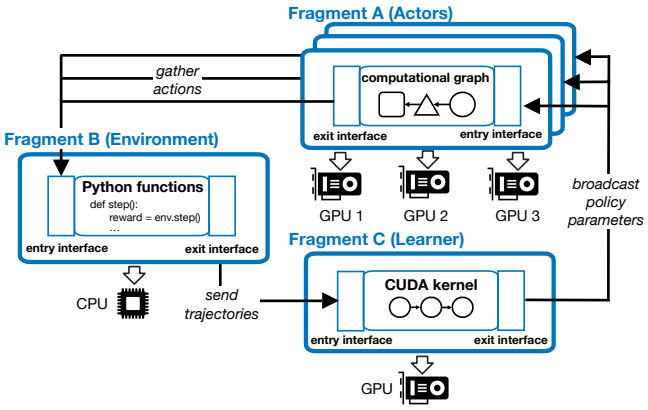


Fig. 3: Fragmented dataflow graph

neous dataflow model, in which independent dataflow representations for different algorithmic components of the RL training loop can be "stitched together" through well-defined interfaces. We refer to this dataflow model as a *fragmented dataflow graph* (FDG), shown in Fig. 3.

Fragments. Each node in an FDG is a potentially data-parallel *fragment*, which is implemented using a bespoke dataflow representation. For example, fragment A in Fig. 3 represents the action computation of an *actor* using the data-parallel operators of a DNN engine [1, 6, 19], performing model inference to decide on an action; fragment B implements the environment simulation directly as parallel Python code; and fragment C conducts the model training, which is implemented as CUDA kernels [11, 38].

Based on the fragment allocation, FDGs support the execution of RL computation on different devices. Each fragment is assigned to one or more devices: the DNN operator representation of fragment A allows it to be deployed on GPUs or CPUs; fragment B requires a Python interpreter with the `multiprocessing` library [10] on CPU cores; and instances of fragment C must run as CUDA kernels on GPUs.

In addition, it is possible to parallelize fragment execution by having multiple instances of a fragment and assigning each instance to a separate device. In Fig. 3, fragment A is replicated 3 times and executed by 3 GPU devices in parallel.

Communication. To form a connected FDG of the complete RL training loop, each fragment must support *entry* and *exit* interfaces, allowing them to exchange data: the entry interface receives data as a byte buffer, which is transformed into a fragment-specific representation (e.g., a tensor); and the exit interface requires a fragment to provide output, which is serialized for consumption by the next fragment.

The implementation of these interfaces depends on how the fragments are deployed: if two fragments are placed on different workers, the interface must use network communication to exchange data e.g., using an RPC protocol over Infiniband [48]; if two fragments are co-located on devices on the same worker, they can share data structures e.g., using inter-GPU communication links such as NVLink [40].

According to the communication method and distribution policy (§4.2), fragment interfaces may be *blocking*, which means that they only execute after all data has arrived, e.g., after a collective communication `AllReduce` round when aggregating DNN gradients. Alternatively, they can be *non-blocking*, which means that they execute continuously, e.g., allowing actors to interact with environments asynchronously.

3.2 Trade-offs with fragmented dataflow graphs

FDGs subsume execution strategies of existing RL systems. For example, an FDG may represent an actor and its environment as a single CPU-based fragment, and a learner as a GPU-based fragment, as proposed by Acme [18]. Alternatively, it may create a larger GPU fragment by moving the DNN policy to the learner, accelerating policy inference, as proposed by SEED RL [8]. An even larger fragment may contain the actor, learner, policy, and environment, executing the whole training loop on a single GPU, as proposed by WarpDrive [23] and Anakin [16].

More generally, FDGs expose two dimensions that impact execution performance:

Fragment granularity refers to the code size, which affects device utilization: a small fragment may underutilize a GPU, and a large one may exhaust GPU memory.

Fragment granularity also determines the ratio between computation and communication. The frequency and amount of data synchronization between fragments often limit scalability: coarser fragments require less synchronization with other fragments, which reduces communication overhead, but they remove opportunities for parallelism. For example, multiple fragments may exchange trajectories frequently at each step; alternatively, they may batch data from multiple steps and communicate only once in each episode.

Fragment co-location is the assignment of fragments to devices on the same worker. Co-locating fragments avoids network communication (e.g., Ethernet or InfiniBand) and instead uses more efficient intra-node communication (e.g., NVLink or PCIe). Whether two fragments can be co-located depends on the available resources on the worker, such as the number of available GPUs.

Choosing the right trade-off between fragment granularity and co-location is key to achieving good performance. In the next section, we describe how FDGs allow users to define an RL algorithm and select between different distribution policies, which expose these trade-offs.

4 Using MSRL

MSRL is our system that implements FDGs for parallel and distributed execution of RL algorithms based on distribution policies. We describe the APIs supported by MSRL for users to define RL algorithms (§4.1) and the distribution policies supported by MSRL to deploy FDGs (§4.2).

Type	API	Description
Component	`Agent, Actor, Learner, Trainer`	Abstract classes for components
	`Actor.act(...)`	Trajectory collection
	`Learner.learn(...)`	DNN policy training
	`Trainer.train(...)`	RL training loop
	`MSRL.agent_act(...)`	Invoke actor
	`MSRL.agent_learn(...)`	Invoke learner
	`MSRL.env_step(...)`	Execute environment
	`MSRL.env_reset()`	Reset environment
Interaction	`MSRL.replay_buffer_insert(...)`	Store trajectories in buffer
	`MSRL.replay_buffer_sample()`	Sample trajectories from buffer

Tab. 2: MSRL APIs

4.1 MSRL APIs

MSRL's APIs are designed to decouple the algorithm logic from its deployment, while supporting familiar algorithmic concepts (i.e., agents, actors, learners, trainers, and environments). As listed in Tab. 2, MSRL supports *component* and *interaction* APIs:

Component APIs specify an RL algorithm by defining algorithmic components derived from abstract classes. An `Agent` consists of actors and learners: actors collect trajectories in `Actor.act()` by invoking `MSRL.env_step()`; and learners implement the DNN update logic in `Learner.learn()`. A *trainer* constructs the RL training loop in `Trainer.train()`. It can use `MSRL.env_step()` to invoke the environment implementation and `MSRL.env_reset()` to reset the training episode.

Interaction APIs offer RL-specific functionality to algorithmic components. For example, an *actor* can store collected trajectories in a replay buffer using `MSRL.replay_buffer_insert()`, and a *learner* can sample from that replay buffer with `MSRL.replay_buffer_sample()`. This avoids direct invocations between components, which allows MSRL to distributed fragments transparently.

Alg. 1 shows a sample implementation of the multi-agent PPO (MAPPO) algorithm [57]. (For brevity, it omits the DNN policy definition.) The `MAPPOAgent` (line 1) defines the agent behavior: it interacts with the environment through `MAPPOActor` (line 6), and performs the policy training with `MAPPOLearner` (line 12). The agent collects trajectories (lines 8–9), and updates its DNN policy (lines 15–21).

`MAPPOTrainer` defines the RL training loop (line 23). At the start of each episode, it resets the environment (`MSRL.env_reset()`) and calls `MSRL.agent_act()` to place trajectories (line 28) in a replay buffer (line 10). The trainer invokes the learner through `MSRL.agent_learn()` (line 29).

To separate the algorithm's logic from its deployment, MSRL uses configurations, specified as Python dictionaries: an *algorithm configuration* instantiates the algorithmic components and their hyper-parameters (e.g., the number of agents and learning rates). In the MAPPO example (lines 30–38), the configuration requests 4 agents, each with 3 actor and 1 learner. Each actor interacts with 32 environments; and a *deployment configuration* defines (i) the resources (e.g., GPUs, CPUs, and worker nodes) and (ii) a *distribution policy*. In the example (lines 39–42), it deploys workers with 4 GPUs each, using the SingleLearnerCoarse distribution policy.

Algorithm 1: MAPPO algorithm in MSRL

```
1  class MAPPOAgent(Agent):
2    def act(self,state):
3      return self.actors.act(state)
4    def learn(self,sample):
5      return self.learner.learn(sample)
6  class MAPPOActor(Actor)
7    def act(state):
8      action = self.actor_net(state)
9      reward,new_state = MSRL.env_step(action)
10     MSRL.replay_buffer_insert(reward,new_state)
11     return reward,new_state
12 class MAPPOLearner(Learner):
13   def learn():
14     sample = MSRL.replay_buffer_sample()
15     action,reward,state,next_state = sample
16     last_pred = self.critic_net(next_state)
17     pred = self.critic_net(state)
18     r = discounted_reward(reward,last_pred,self.gamma)
19     adv = gae(reward,next_state,pred,last_pred,self.
           gamma)
20     for i in range(self.iter):
21       loss += self.mappo_net_train(action,state,adv,r)
22     return loss / self.iter
23 class MAPPOTrainer(Trainer):
24   def train(self,episode):
25     for i in range(episode):
26       state = MSRL.env_reset()
27       for j in range(self.duration):
28         reward,new_state = MSRL.agent_act(state)
29       loss = MSRL.agent_learn()
30 mappo_algorithm_config = {
31   'agent':{'num':4,'name':MAPPOAgent,
32            'actor':MAPPOActor,'learner':MAPPOLearner},
33   'actor':{'num':3,'name':MAPPOActor,
34            'policy':MAPPOActorNet,'env':True},
35   'learner':{'num':1,'name':MAPPOLearner,
36              'policy':[MAPPOCriticNet,MAPPONetTrain],
37              'params':{'gamma':0.9}},
38   'env':{'name':MPE,'num':32,'params':{'name':'MPE'}}}
39 mappo_deployment_config = {
40   'workers':[198.168.152.19, 198.168.152.20, [...],
41   'GPUs_per_worker':4},
42   'distribution_policy':'SingleLearnerCoarse'}
```

4.2 Distribution policies

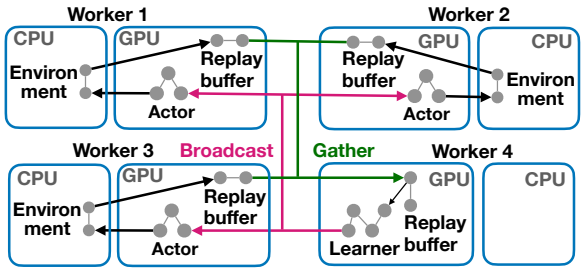
A *distribution policy* (DP) governs how MSRL distributes and parallelizes an RL algorithm by allocating, replicating and collocating fragments from the FDG to workers and devices.

In general, there exists no single DP that is optimal in all cases: the performance and applicability of a DP depends on the type of RL algorithm, the size and complexity of the DNN model, its hyper-parameters, the available cluster compute resources (i.e., CPUs and GPUs), and the network bandwidth. MSRL allows users to easily switch between DPs, either for the same RL algorithm or when using different algorithms. MSRL provides six DPs, which follow widely used hard-coded distribution strategies of existing RL systems.

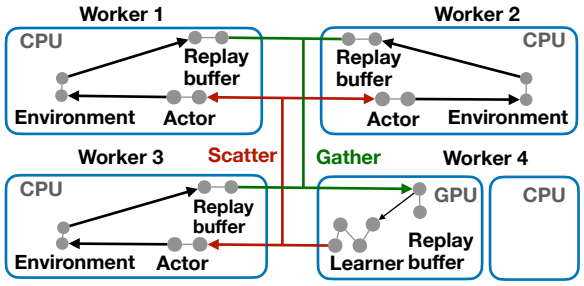
Next we give an overview of the support DPs and their trade-offs. Tab. 3 shows how three of the DPs deploy the fragments of an RL algorithm. (We list all DPs currently implemented by MSRL in Appendix A).

DP-SingleLearnerCoarse replicates the actor and environment fragments but uses a single learner. The policy DNN is replicated across the actors and learner, which only requires coarse synchronization. This policy is therefore most suitable with

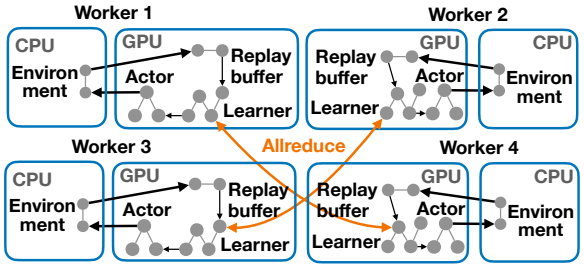
DP-SingleLearnerCoarse
replicate: *actor*,*env* split: *learner* e.g., Acme [18], Sebulba [16]



DP-SingleLearnerFine
replicate: fused *actor/env* split: *learner* e.g., SEED RL [8]



DP-MultiLearner
replicate: fused *actor/learner*, *env*



Tab. 3: Sample distribution policies with deployments

computationally-expensive environments that need scaling out, but small DNN models that can be synchronized in a batched fashion, e.g., Acme [18], Sebulba [16].

The MAPPO deployment in Alg. 1 uses DP-SingleLearnerCoarse: each agent is partitioned into 4 GPU fragments, i.e., 3 actors and 1 learner, and 3 CPU fragments for environments. Actor and environment fragments are collocated. This setting is replicated for each of the 4 MAPPO agents, as specified in the algorithm configuration.

In contrast, DP-SingleLearnerFine fuses the actor and environment into a single CPU fragment, and only deploys the learner on a GPU. Therefore it does not communicate policy parameters between workers, which is preferable for large DNN models with many parameters. Compared to the DP-SingleLearnerCoarse, it relies on fine-grained synchronization: training data is exchanged at each step, instead of being batched per episode. For good performance, DP-SingeLearnerFine therefore requires high bandwidth connectivity between workers, e.g., SEED RL [8].

DP-MultiLearner performs data-parallel training with multiple learners. This policy is necessary when the data generated from actors becomes too large for a single GPU, and e.g., DP-SingleLearnerCoarse cannot be used. However, it requires the tuning of hyper-parameters (e.g., the learning rate) to scale due to its reliance on data parallelism. Since workers only exchange information about the trained policy (e.g., aggregated DNN gradients), DP-MultiLearner is communication efficient, supporting fully decentralized MARL training [5, 43, 59, 62].

MSRL supports further policies: DP-GPUOnly fuses the RL training loop into a single GPU fragment and distributes it to multiple GPU devices. DP-Environments dedicates one or more workers for the execution of complex or compute-intensive environments (e.g., physics simulations). Finally, DP-Central introduces a separate fragment for a centralized component (e.g., policy pool [61] or parameter server [24]).

The choice of the best distribution policy depends on the algorithm's characteristics and available hardware resources: single-agent RL algorithms, such as PPO/A3C, exhibit the best performance under the DP-SingleLearnerCoarse policy, which distributes actors to speed up trajectory collection through data parallelism; multi-agent algorithms, such as MAPPO/MADDPG, require a DP-MultiLearner policy that distributes actors and learners separately from agents, thus parallelizing both trajectory collection and model training; a DP-GPUOnly policy can be used in a GPU environment to fuse the training loop and execute it entirely on GPUs, which offers the best performance.

Based on the hardware resources, bottlenecks shift between DPs: the DP-SingleLearnerFine policy exchanges data between actors/environments at a fine granularity by distributing inference/training to one GPU worker and environments across CPU workers. Despite the need for frequent communication, this policy is suitable in situations where GPUs are scarce; in contrast, the DP-SingleLearnerCoarse policy co-locates the GPU DNN inference with the environments, enabling the learner to gather batched training data. With enough GPUs, this policy accelerates trajectory collection.

5 MSRL Architecture

We describe MSRL's architecture, explaining how FDGs are generated (§5.1) and executed (§5.2).

MSRL follows a coordinator/worker design (see Fig. 4): a user submits the RL algorithm implementation to the *coordinator* ❶. The coordinator generates the fragments that constitute the FDG and dispatches them to the workers ❷. Each worker maintains one or more *execution backends* (e.g., a DNN engine, a CUDA job scheduler, a Python interpreter), each managing devices (e.g., GPUs or CPU cores). After receiving fragments, the worker optimizes them ❸ and submits them to a backend for execution ❹.

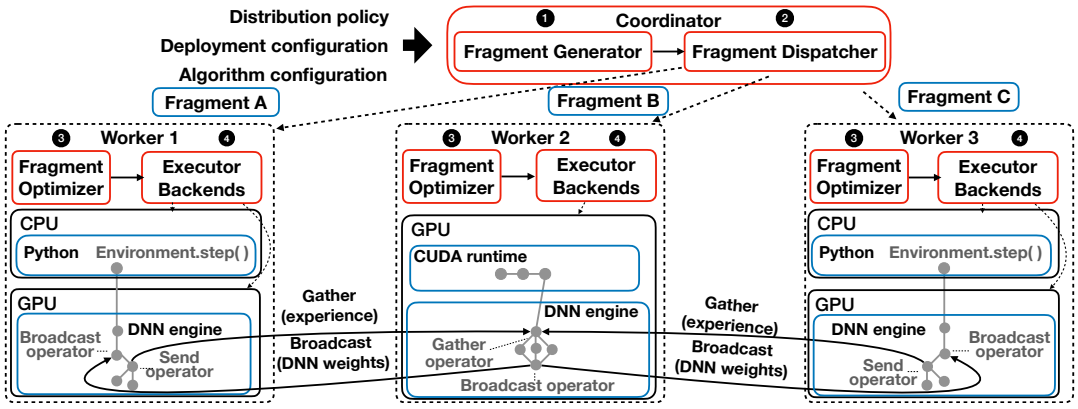


Fig. 4: Overview of the MSRL architecture

5.1 Generating FDGs

The coordinator has two components:

The **FDG Generator** partitions the RL algorithm according to the DP specified in the deployment configuration (§4.1). It splits the implementation at fragment boundaries and injects code for the interface implementations between fragments. The fragment logic is then emitted as part of a `run()` method in a generated `Fragment` class.

The partitioning of the RL algorithm into fragments uses the information associated with a DP. Each DP provides a set of rules about (1) how fragments are generated and (2) how they are distributed. The DP contains a *fragment template*, which associates each fragment with a Python class that has placeholders for class names, member functions, and other relevant elements. These placeholders instruct the Generator where to insert specific algorithm logic, such as actor computation, into the generated fragments. The DP also defines the communication operations required by the interfaces. To choose appropriate implementations, the DP refers to communication operations supported by backends (e.g., `comms.AllGather` [20] in a DNN engine). The DP also specifies which fragments are replicated into multiple instances for parallel execution, or co-located on the same worker.

When partitioning the RL training loop, the boundaries between fragments follow the algorithmic components (actors, learners, environments). The data to be transferred between fragments is defined in terms of the function signatures of the components. The partitioning is done on a dataflow representation of the RL algorithm: nodes in the dataflow graph are Python statements; edges represent the dataflow through variables. Therefore, edges at the boundary of algorithmic components describe fragment interfaces, and we refer to them as *boundary edges*. MSRL creates fragments by partitioning the dataflow graph at these boundary edges.

As an example, consider partitioning the MAPPO algorithm (Alg. 1) into actor and learner fragments, with the boundary between lines 28 and 29. Fig. 5a shows the simplified dataflow graph obtained after static analysis, with the input/output data of the components shown in red. Splitting the graph at these boundaries, partitions it into two fragments (see Fig. 5b), which communicate through the new interface obtained from the boundary edges (shown in red).

Algorithm 2 Generation of fragmented dataflow graphs

function generate_FDG (*alg*, *DP*):
0: *FDG* ← {}, *DFG* ← *generate_DFG(alg)*
1: *boundary_edges* ← *obtain_boundary_edges(DFG)*
2: *interfaces* ← *generate_interfaces(boundary_edges, DP)*
3: **for** *boundary* in *boundary_edges* **do**
4: *fragment_code* ← *build_fragment(alg, boundary)*
5: *fragment* ← *build_fragment(fragment_code, interfaces, DP)*
6: *FDG* ← *FDG* ∪ *fragment*
7: **return** FDG

Alg. 2 summarizes the FDG generation. The Generator takes the RL algorithm's abstract syntax tree (*alg*) and distribution policy *DP* as input (line 0) and constructs its dataflow graph (*DFG*) (line 1). Next, it locates the algorithmic components and determines the boundary edges from the DFG (line 2). Based on the information from the DP, it constructs the communication interfaces (line 3). For each boundary edge (line 4), it extracts the fragment code (line 5) and builds the fragment with its interface implementation (line 6). At the end, it returns the complete FDG (line 8).

The **Fragment Dispatcher** launches instances of execution backends on each worker according to the devices from the deployment configuration. It also sets up distributed communication, e.g., through MPI [30], as required by the fragment interfaces. Finally, it assigns fragments to devices based on the DP and sends the fragments to the workers.

5.2 Executing fragments

The workers use a set of *execution backends* to take the fragment code and run it. Some backends (e.g., a DNN engine) produce executable machine code for a given device (e.g., GPUs) by translating the fragment implementation into a computational graph, which enables code optimizations.

The communication between fragments is also handled by the execution backends. For example, a DNN engine

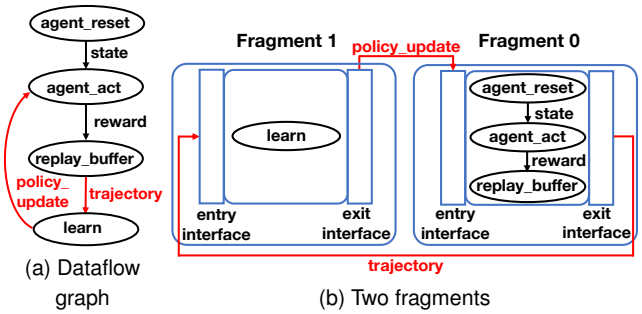


Fig. 5: Example of FDG generation for MAPPO

	MSRL	RLlib	WarpDrive
PPO	207	347 (+68%)	400 (+93%)
A3C	267	428 (+60%)	*n/a*

Tab. 4: Lines of code for the RL algorithm implementations

uses communication operators as part of its computational graph, automatically selecting suitable implementations (e.g., NCCL [39] for GPU collective communication).

Each worker has two components:

The **Fragment Optimizer** optimizes fragments that have been received for a given execution backend. To avoid the overhead of executing multiple instances of a replicated fragment, the optimizer attempts to *fuse* instances represented as computational graphs: it exploits the support of DNN engines to process data in a SIMD fashion by batching tensors from multiple fragment instances.

The Optimizer performs this transformation on the fragment's AST before submitting it to the DNN engine. It locates the AST nodes of tensors and merges their data. It then computes the new tensor shape to create a single tensor that can be processed by other data-parallel operators.

The **Executor backends** execute the fragments on a given target device: a DNN engine (MindSpore) executes computational graph on GPUs or CPUs; a CUDA job scheduler runs CUDA kernels on GPUs; a Python interpreter executes Python fragments on CPU cores; and a container scheduler can run arbitrary compute containers on CPU cores.

6 Evaluation

Our experimental evaluation answers the following questions: (i) what is the training performance that MSRL with FDGs achieves compared to existing RL systems with fixed execution strategies (§6.2)?; (ii) how does MSRL benefit from choosing different distribution policies (§6.3)?; and (iii) how well does MSRL scale in terms of the number of agents and the amount of training data (§6.4)?

6.1 Experimental set-up

Implementation. We implement MSRL in 11,700 lines of Python and C++ code. It uses CUDA 11.03, cuDNN 8.2.1, OpenMPI 4.0.3, and the MindSpore DNN framework 1.8.0 [19] as a GPU-based execution backend. The source code is available at https://github.com/mindspore-lab/mindrl.

Cluster	CPU cores *#nodes × #per node*	GPUs *#nodes × #per node*	Interconnects *intra-, inter-node*
Azure VMs NC24s_v2	Intel Xeon E5-2690 16×24, 448 GB	NVIDIA P100 16×4	PCIe 10 GbE
Local cluster	Intel Xeon 8160 4×96, 250 GB	NVIDIA V100 4×8	NVLink 100 Gbps IB

Tab. 5: Testbed configuration

MSRL uses the following distribution policies from Appendix A: DP-SingleLearnerCoarse; DP-SingleLearnerFine; DP-MultiLearner; DP-GPUOnly; and DP-Environments.

Baseline comparisons. For comparison, we use RLlib [34] of Ray V2.0, as a representative distributed RL system, and WarpDrive V1.6 [23], as a single-GPU system that accelerates the full RL training loop. Note that the implementations of RLlib Flow [26] and PodRacer [16] are unavailable.

RL algorithms. We focus on three popular algorithms: (1) *proximal policy optimization* (PPO) [47]; its multi-agent version, (2) *multi-agent PPO* (MAPPO) [57]; and (3) *asynchronous advantage actor-critic* (A3C) [31].

Tab. 4 compares the lines of code for the algorithm implementations. The RLLib and WarpDrive implementations require 68% and 93% more lines than MSRL, respectively, due to the hardcoded execution and distribution logic. This shows a benefit of MSRL's approach, which allows users to focus on the algorithm logic in their implementations.

For environments, we use two games (CartPole, HalfCheater) from the MuJoCo simulation engine [52], and two strategies (Spread, Tag) from the *multi-agent particle environment* (MPE) [28]. The policies use a 7-layer DNN.

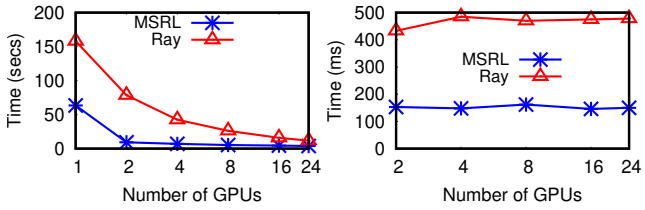
Testbeds. We conduct experiments on a *cloud* and a *local* cluster. The hardware details are given in Tab. 5: the cloud cluster has 16 VMs (with 64 GPUs); the local cluster has 4 nodes (with 32 GPUs). All nodes run Ubuntu Linux 20.04.

Metrics. For PPO, we measure (i) the training time to reach a given reward and (ii) the time per episode. For MAPPO, as the problem size increases with agents, we report (i) training time against the problem size and (ii) training throughput.

6.2 Performance with FDGs against baselines

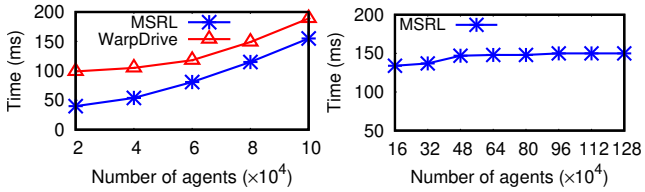
We investigate the performance impact that MSRL's FDG abstraction incurs compared to RL systems that only support hardcoded parallelization and distribution approaches.

Distributed training. We compare MSRL with DP-SingleLearnerCoarse to Ray [34] using PPO and A3C on the local cluster. For Ray, both algorithms are implemented using RLlib-Flow [26] and tuned based on RLlib's public PyTorch implementation [44]. DP-SingleLearnerCoarse is equivalent to the distribution approach implemented by RLlib-Flow's PPO and A3C implementations.



(a) Episode time vs. GPUs (PPO) (b) Episode time vs. GPUs (A3C)

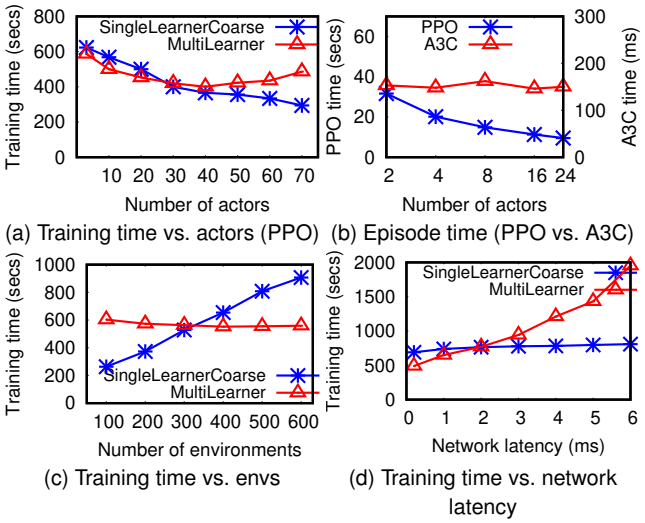
Fig. 6: Performance comparison with Ray



(a) Episode time vs. agents (1 GPU)

(b) Episode time vs. agents (n GPUs)

Fig. 7: Performance comparison with WarpDrive (PPO)



(a) Training time vs. actors (PPO) (b) Episode time (PPO vs. A3C)

(c) Training time vs. envs (d) Training time vs. network latency

Fig. 8: Impact of parameters on distribution policies

For PPO, we distribute 320 environments evenly among the actors, i.e., each actor interacts with $320/\#actors$ environments. A single learner trains the DNN. For A3C, one learner performs gradient optimization with gradients collected asynchronously from actors. Each actor interacts with one environment and computes gradients locally. We measure the time per episode, which is dominated by actor and environment execution. Since the DNN training/inference time is negligible, the fact that MSRL and Ray use different DNN frameworks (MindSpore vs. PyTorch) has low impact.

Fig. 6a shows the time per episode for PPO. MSRL's time with 1 GPU is 2.5× faster than Ray's, because Ray's CPU actor interacts with all environments sequentially. As the number of GPUs increases, both systems reduce episode time, because each actor interacts with fewer environments. With 24 GPUs, it takes 3.9 s for MSRL to execute an episode compared to 11.4 s for Ray (3× speed-up). When actors interact with multiple environments, MSRL combines DNN inference into one operation through FDG fusion, increasing GPU parallelism. It also uses fragments to execute environment steps in parallel by launching multiple processes.

Fig. 6b shows the time per episode for A3C. Both systems exhibit constant time with more GPUs, because the workload of each GPU executing an actor remains unchanged. MSRL executes actors 2.2× faster than Ray: since its distribution policy exploits customized asynchronous send/receive operations from the DL engine, it can avoid further data copies between GPUs and CPUs. In contrast, Ray must copy data to the CPU to communicate asynchronously.

In addition, MSRL generates the FDG that can be translated into a computational graph by the DL engine, enabling code optimizations and efficient execution. By leveraging code templates, MSRL generates optimized fragment code by directly manipulating the FDG AST.

GPU only training. Next, we use MSRL to deploy PPO with distribution policy DP-GPUOnly, which fuses the training loop into a single GPU fragment and replicates it for distributed execution. We use the *simple tag* MPE environment [28], which simulates a predator-prey game in which chaser agents are rewarded for catching runner agents. We train different numbers of agents, thus increasing the number of environments, on the local cluster and measure the training time per episode. We compare against WarpDrive [23], which performs single-GPU end-to-end RL training.

Fig. 7a shows the training time on 1 GPU. Compared to WarpDrive, MSRL is 1.2–2.5× faster when ranging from 20,000 to 100,000 agents. MSRL's DL engine (MindSpore) compiles fragments to computational graphs, exploiting more parallelization and optimization opportunities than WarpDrive's hand-crafted CUDA implementation.

While WarpDrive cannot scale to more than 1 GPU, Fig. 7b shows MSRL's performance when using up to 16 GPUs (each GPU trains 80,000 agents). Initially, training time increases from 138 ms to 150 ms due to the increased computation on a single worker (i.e., up to 640,000 agents). After that, training time is stable, and it is limited by communication bandwidth (InfiniBand, NVLink).

Conclusions: MSRL's FDG abstraction provides distribution policies for PPO and A3C that are tailored to their bottlenecks, e.g., enabling parallel environment execution and aggressively parallelizing GPU execution. Ray is limited by the distribution approach of its RLlib library; WarpDrive's manual CUDA implementation prevents it from exploiting more sophisticated compiler optimizations.

6.3 Trade-offs between distribution policies

Next, we explore the trade-offs between different distribution policies when changing RL algorithms and resources.

Actors. We measure PPO's training time with two distribution

policies, DP-SingleLearnerCoarse and DP-MultiLearner. We use a reward of 3,000 with 200 environments.

Fig. 8a shows the training time with 2 to 70 actors. DP-MultiLearner outperforms DP-SingleLearnerCoarse with fewer than 30 actors, but DP-SingleLearnerCoarse scales better after that, converging faster with more actors. Since DP-SingleLearnerCoarse only has 1 learner, its training batch size is fixed. Adding more actors therefore only distributes environment execution. In contrast, DP-MultiLearner fuses actors and learners into single fragments. With more actors, it also adds learners, reducing the batch size for each learner. This adds randomness to the training, affecting convergence [17].

Next, we compare two algorithms, PPO and A3C, under the same distribution policy DP-SingleLearnerCoarse.

Fig. 8b shows the time per episode for up to 24 actors. For PPO, the time decreases with the actor count; in contrast, A3C's time stays constant. Adding actors in PPO increases the parallelism of environment execution, thus reducing the workload per actor; for A3C, each actor only interacts with one environment, which makes its workload independent of the actor count. To reduce the episode time for A3C, a new distribution policy could be written that distributes the actor among multiple devices, combining data- or task-parallelism.

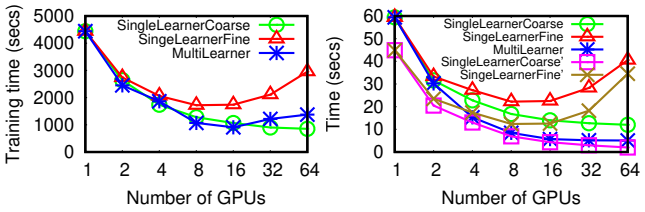
Environments. We explore how changing the number of environments affects the choice of distribution policy. When an agent interacts with more environments in parallel per episode, it trains with more data, improving training performance.

Fig. 8c shows the training time with 50 actors with 100–600 environments under DP-SingleLearnerCoarse and DP-MultiLearner. DP-MultiLearner scales better than DP-SingleLearnerCoarse with more than 320 environments: DP-SingleLearnerCoarse's training time increases with more environments, because its actors send trajectories to the learner, adding communication overhead; DP-MultiLearner only communicates gradients, having a fixed overhead.

Network latency. We examine the behavior of DP-SingleLearnerCoarse and DP-MultiLearner with PPO under different network latencies. We change network latency in our cloud cluster using the Linux traffic control (`tc`) tool from 0.2 ms to 6 ms. We use 400 environments and 50 actors.

As Fig. 8d shows, DP-MultiLearner is more sensitive to network latency than DP-SingleLearnerCoarse, and its training time increases with higher latency: since DP-MultiLearner uses Mindspore's data parallel model [19] to broadcast, aggregate and update gradients, it transmits many small tensors. This makes it a more suitable choice for cluster with low latency (< 2 ms); DP-SingleLearnerCoarse transmits the trajectory and DNN model weights as large tensors, performing data transmissions less frequently.

Cluster size. Finally, we study the performance of PPO under 3 distribution policies when increasing the GPU count: DP-SingleLearnerCoarse and DP-SingleLearnerFine use a single learner but apply different synchronization granularities; DP-MultiLearner scales to multiple learners using data-parallelism.



(a) Training time vs. GPUs (b) Episode time vs. GPUs

Fig. 9: Impact of GPU count on distribution polices

We use a constant 320 Mujoco HalfCheetah [4] environments.

Fig. 9a shows the training time in the cloud cluster to reach a reward of 4,000 with up to 64 GPUs; Fig. 9b reports the time per episode. With 64 GPUs, DP-SingleLearnerCoarse achieves the best speed-up in training time (5.3×). It maintains local copies of the DNN model at the actor and learner, and only actors send the batched states and rewards to the learner at the end of each episode (i.e., after 1,000 steps). This reduces the overhead with more GPUs compared to DP-SingleLearnerFine, whose actor fragments must communicate with the learner at each step.

DP-MultiLearner exhibits a different behavior: with 16 GPUs, it achieves better performance than either DP-SingleLearnerCoarse and DP-SingleLearnerFine, because it distributes policy training: it trains smaller trajectory batches on each GPU and aggregates the gradients from all GPUs. Instead, DP-SingleLearnerFine and DP-SingleLearnerCoarse gather all batches and train them using 1 learner.

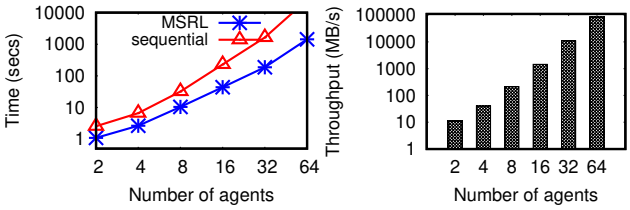
With more than 16 GPUs, DP-MultiLearner performs worse than DP-SingleLearnerCoarse: batches become smaller, making the gradient aggregation less efficient compared to training a large batch. Although DP-MultiLearner trains each episode faster than DP-SingleLearnerCoarse (see Fig. 9b), it requires more episodes to reach a similar reward value.

Note that DP-SingleLearnerCoarse and DP-SingleLearnerFine use the original PPO implementation with 1 learner [47], which limits scalability due to the centralized policy training (❸ in Fig. 1). To ignore this bottleneck in the algorithm, Fig. 9b also reports only the policy training time (labelled DP-SingleLearnerCoarse' and DP-SingeLearnerFine'). Now, MSRL scales better: when moving from 32 to 64 GPUs, performance increases by 25%.

Conclusions: As hyper-parameters, network properties or GPU counts change, the differences between distribution policies in terms of synchronization granularity and frequency of impact performance. MSRL's ability to allow users to switch between distribution policies at deployment time means that they can achieve the best performance in different scenarios without changing the algorithm implementation.

6.4 Scalability

Finally, we investigate how MSRL's design scales with the number of deployed agents for a MARL algorithm and of environments, thus increasing training data. We want to validate



(a) Training time per episode (b) Training throughput

Fig. 10: Scalability with agent count (MAPPO)

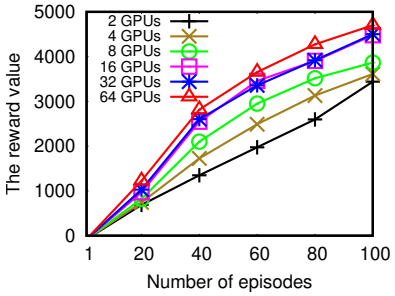


Fig. 11: Statistical efficiency with environment count (PPO)

if MSRL's approach introduces scalability bottlenecks.

Agents. We use MAPPO with the MPE *simple spread* environment [28], in which n agents learn to cover n landmarks while avoiding collisions. Agents must also process global observations on how far the closest agent is to each landmark. This results in $O(n^3)$ observations with n agents, quickly growing in computational cost and GPU memory usage [28]. We deploy on the cloud cluster using DP-Environments: each GPU trains 1 agent, and 1 worker executes all environments.

Fig. 10a shows the training time per episode for up to 64 GPUs against a sequential baseline (1 GPU). Due to its cubic complexity, the time increases both for the baseline and MSRL. With distributed training, MSRL's time grows more slowly than the baseline: with 32 agents, MSRL improves performance by 58×; with 64 agents, the baseline exhausts GPU memory, while MSRL trains one episode in 23.8 mins.

Fig. 10b compares the throughput with different agent numbers. Throughput is measured as the amount of data trained per second (in MB/s). Adding more agents (i.e., GPUs) significantly improves throughput, and the margin becomes larger with more GPUs: the throughput with 64 agents is over 7,600× higher than with 2 agents, as multiple GPUs train agents in parallel, processing more observations per GPU.

Environments. We observe the effect of more environments on statistical efficiency, i.e., the episodes needed to reach a given reward. We use 10 environments per CPU, adding more workers in the cloud cluster using DP-SingleLearnerCoarse.

Fig. 11 shows the reward along with the number of episodes for different environment counts. More environments lead to a higher reward: as more CPUs execute environments, the larger use of trajectories per episode yields a higher reward.

Conclusions: FDGs do not deteriorate scalability. MSRL scales to a large number of data-intensive agents, handling the increase in communication between fragments without bottlenecks. A larger amount of data generated by more environments also improves the statistical efficiency of training.

7 Related Work

DNN compilation. XLA [56] is a domain-specific compiler that accelerates the linear algebra of DNN models. JAX [12] uses just-in-time (JIT) compilation to transform vectorized Python programs to GPU or TPU code. TVM [6] is an automated end-to-end optimizing compiler for DNN training. These approaches focus on DNN training and inference workloads with regular computation/communication patterns. In contrast, RL algorithms exhibit more complex control and data flow in their training loops.

DNN auto-parallelization. Alpa [60] and Unity [53] automatically parallelize and distribute DNN training using data/operator/pipeline parallelism. They search for effective distributed execution plans using dynamic or integer linear programming. In future work, we want to explore the use of optimization techniques to generate an optimal distribution policy for a given RL algorithm. Since an FDG has more heterogeneity than DNN dataflows, the search space is substantially larger and based on more complex cost models.

Dataflow and actor systems. Spark [58] and Naiad [36] express programs as dataflow graphs, sharding data across workers. They provide high-level APIs to express computation as a single homogeneous dataflow. In contrast, FDGs allow different dataflow models to be integrated into a single distributed computation, as governed by distribution policies.

Ray [34] offers a general actor-based platform for distributed computing. To support RL algorithms, it uses domain-specific libraries (RLlib/RayFlow [25, 26]) that hardcode distribution strategies, preventing it from switching strategies based on e.g., hardware properties. Instead, MSRL proposes FDG, a higher-level abstraction for parallelizing and distributing RL training loops, which decouples RL algorithms from their execution through explicit distribution policies.

8 Conclusions

While DNN systems have mature dataflow abstractions that improve execution performance, similar abstractions for RL systems have been under-explored. We described MSRL, a system that supports the flexible parallelization and distribution of RL algorithms using *fragmented dataflow graph*. Accounting for the heterogeneous nature of RL training loops, MSRL separates the algorithm from its execution by using *distribution policies* that allocate dataflow fragments to GPUs and CPUs. Our experiments showed how distribution policies generalize existing RL systems without overhead.

Acknowledgments. We thank the anonymous reviewers and our shepherd, Yibo Zhu, for their helpful comments.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2016.

[2] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18:153:1–153:43, 2017.

[3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *CoRR*, abs/1912.06680, 2019.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.

[5] Michael Chang, Sidhant Kaushik, S. Matthew Weinberg, Tom Griffiths, and Sergey Levine. Decentralized reinforcement learning: Global decision-making via local economic transactions. In *Proceedings of the 37th International Conference on Machine Learning, ICML*, 2020.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2018.

[7] Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre M. Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2020.

[8] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. SEED RL: scalable and efficient deep-rl with accelerated central inference. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[9] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: scalable distributed deep-rl with importance weighted actor-learner architectures. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, 2018.

[10] Python Software Foundation. Process-based parallelism, 2022. [Online; accessed 10-December-2021].

[11] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021.

[12] Roy Frostig, Matthew Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. In *Systems for Machine Learning*, 2018.

[13] Sven Gronauer and Klaus Diepold. Multi-agent deep reinforcement learning: a survey. *Artificial Intelligence Review*, pages 1–49, 2021.

[14] Sergio Guadarrama, Anoop Korattikara, Oscar Ramirez, Pablo Castro, Ethan Holly, Sam Fishman, Ke Wang, Ekaterina Gonina, Neal Wu, Efi Kokiopoulou, Luciano Sbaiz, Jamie Smith, Gábor Bartók, Jesse Berent, Chris Harris, Vincent Vanhoucke, and Eugene Brevdo. TF-Agents: A library for reinforcement learning in tensorflow. https://github.com/tensorflow/agents, 2018. [Online; accessed 25-June-2019].

[15] Abhishek Gupta, Justin Yu, Tony Z. Zhao, Vikash Kumar, Aaron Rovinsky, Kelvin Xu, Thomas Devlin, and Sergey Levine. Reset-free reinforcement learning via multi-task learning: Learning dexterous manipulation behaviors without human intervention. In *IEEE International Conference on Robotics and Automation, ICRA 2021*, 2021.

[16] Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable reinforcement learning. *CoRR*, abs/2104.06272, 2021.

[17] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*, 2017.

[18] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alexander Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Çaglar Gülçehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. *CoRR*, abs/2006.00979, 2020.

[19] Huawei. Mindspore. https://www.mindspore.cn/en, 2020.

[20] Huawei. Mindspore all gather operator. https://www.mindspore.cn/docs/en/r1.7/api_python/ops/mindspore.ops.AllGather.html?highlight=allgather, 2022. [Online; accessed 18-May-2022].

[21] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, 2021.

[22] Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. In *Advances in Neural Information Processing Systems 12, [NIPS Conference]*, 1999.

[23] Tian Lan, Sunil Srinivasa, and Stephan Zheng. Warpdrive: Extremely fast end-to-end deep multi-agent reinforcement learning on a GPU. *CoRR*, abs/2108.13976, 2021.

[24] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2014.

[25] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael I. Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, 2018.

[26] Eric Liang, Zhanghao Wu, Michael Luo, Sven Mika, and Ion Stoica. Distributed reinforcement learning is a dataflow problem. *CoRR*, abs/2011.12719, 2020.

[27] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR*, 2016.

[28] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems*, 2017.

[29] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter R. Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2020.

[30] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[31] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, 2016.

[32] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML*, 2016.

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nat.*, 518(7540):529–533, 2015.

[34] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2018.

[35] Paul Muller, Shayegan Omidshafiei, Mark Rowland, Karl Tuyls, Julien Pérolat, Siqi Liu, Daniel Hennes, Luke Marris, Marc Lanctot, Edward Hughes, Zhe Wang, Guy Lever, Nicolas Heess, Thore Graepel, and Rémi Munos. A generalized training approach for multiagent learning. In *8th International Conference on Learning Representations, ICLR*, 2020.

[36] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13*, 2013.

[37] Ranjit Nair, Milind Tambe, Makoto Yokoo, David V. Pynadath, and Stacy Marsella. Taming decentralized pomdps: Towards efficient policy computation for multiagent settings. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 2003.

[38] NVIDIA. CUDA Toolkit: Develop, optimize and deploy gpu-accelerated apps, 2022. [Online; accessed 10-December-2021].

[39] NVIDIA. NCCL: Nvidia collective communications library, 2022. [Online; accessed 10-December-2021].

[40] NVIDIA. Nvlink and nvswitch. https://www.nvidia.com/en-au/data-center/nvlink, 2023. [Online; accessed 10-Sep-2022].

[41] OpenAI. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt, 2022.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2019.

[43] Chao Qu, Shie Mannor, Huan Xu, Yuan Qi, Le Song, and Junwu Xiong. Value propagation for decentralized networked deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS*, 2019.

[44] Ray. Ray ppo setting. https://github.com/ray-project/ray/blob/master/rllib/tuned_examples/ppo/halfcheetah-ppo.yaml, 2020. [Online; accessed 10-Sep-2022].

[45] Michael Schaarschmidt, Sven Mika, Kai Fricke, and Eiko Yoneki. RLgraph: Modular Computation Graphs for Deep Reinforcement Learning. In *Proceedings of Machine Learning and Systems, MLSys*, 2019.

[46] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[47] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[48] Tom Shanley. *Infiniband*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[49] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nat.*, 529(7587):484–489, 2016.

[50] Peng Sun, Jiechao Xiong, Lei Han, Xinghai Sun, Shuxing Li, Jiawei Xu, Meng Fang, and Zhengyou Zhang. Tleague: A framework for competitive self-play based distributed multi-agent reinforcement learning. *CoRR*, abs/2011.12895, 2020.

[51] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[52] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[53] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Patrick S. McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2022.

[54] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander Sasha Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom Le Paine, Çaglar Gülçehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy P. Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019.

[55] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8:229–256, 1992.

[56] XLA and TensorFlow teams. XLA: Optimizing compiler for machine learning, 2022. [Online; accessed 10-December-2021].

[57] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and YI WU. The surprising effectiveness of ppo in cooperative multi-agent games. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2022.

[58] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

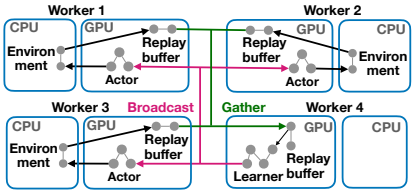
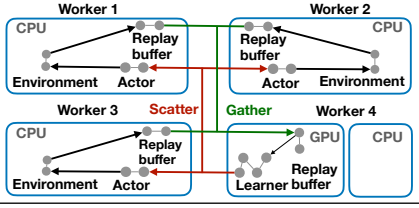
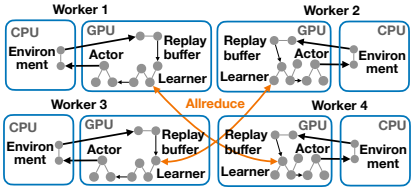
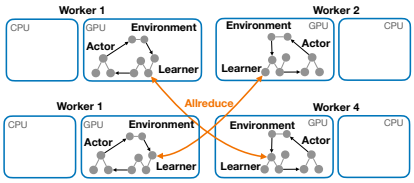
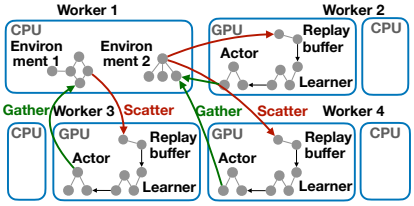
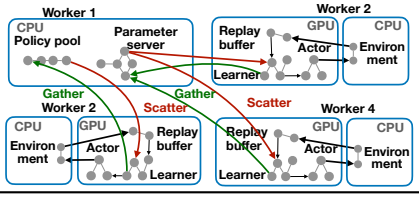
[59] Kaiqing Zhang, Zhuoran Yang, Han Liu, Tong Zhang, and Tamer Basar. Fully decentralized multi-agent reinforcement learning with networked agents. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, 2018.

[60] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, and Joseph E Gonzalez. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. *arXiv preprint arXiv:2201.12023*, 2022.

[61] Ming Zhou, Ziyu Wan, Hanjing Wang, Muning Wen, Runzhe Wu, Ying Wen, Yaodong Yang, Weinan Zhang, and Jun Wang. Malib: A parallel framework for population-based multi-agent reinforcement learning. *CoRR*, abs/2106.07551, 2021.

[62] Matthieu Zimmer, Claire Glanois, Umer Siddique, and Paul Weng. Learning fair policies in decentralized cooperative multi-agent reinforcement learning. In *Proceedings of the 38th International Conference on Machine Learning, ICML*, 2021.

A Supported Distribution Policies

Distribution policy	Deployment	Description
[DP-SingleLearnerCoarse] replicate: (*actor*, *env*) split: (*learner*) e.g., Acme [18], Sebulba [16]		DP-SingleLearnerCoarse replicates the actor and environment fragments: W1–W3 co-locate 1 GPU fragment with an actor for DNN policy inference and 1 CPU fragment for the environment execution. A single GPU fragment with a learner performs policy training (W4), gathering batched training data, training the policy and broadcasting updates.
[DP-SingeLearnerFine] replicate: fused *actor/env* split: *learner* e.g., SEED RL [8]		DP-SingleLearnerFine fuses the actor and environment into 1 fragment (W1–W3) but handles policy inference at the learner (W4), i.e., actors do not contain DNNs. W4 executes policy inference and training in 1 GPU fragment; W1–3 only have CPU fragments. W4 scatters actions to W1–W3 and gathers data for policy training.
[DP-MultiLearner] replicate: fused *actor/learner*, *env*		DP-MultiLearner performs data-parallel training with multiple learners, supporting fully decentralised MARL training [5,43,59,62]. DP-MultiLearner co-locates 2 fragments: a GPU fragment that fuses the actor and learner, accelerating policy inference, training and replay buffer management, and a CPU fragment for environment execution.
[DP-GPUOnly] replicate: fused *actor/learner/env*		DP-GPUOnly fuses the training loop into 1 GPU fragment. To enable communication among GPU fragments, DP-GPUOnly uses `Allreduce` operators compiled into the computational graph with NCCL2 [39]. DP-GPUOnly is a distributed implementation of the single-node systems (e.g., WarpDrive [23]).
[DP-Environments] replicate: fused *actor/learner* split: *env* e.g., MALib [61]		DP-Environments has a dedicated worker for environment execution. W1 has CPU fragments to execute environment instances on multiple CPU cores; W2–W4 fuse the actor and learner to accelerate policy inference and training. W1 gathers the inferred actions and scatters the states and rewards.
[DP-Central] replicate: fused *actor/learner*, *env* split: *param server/policy pool*		DP-Central supports a central *policy pool* [61] or *parameter server* [24] on a separate worker (W1). W2–W4 co-locate GPU fragments for policy inference and training and CPU fragments for environment execution.