MSRL: Distributed Reinforcement Learning with Dataflow Fragments

Huanzhou Zhu*  
Imperial College London

Bo Zhao*  
Imperial College London

Gang Chen  
Huawei Technologies Co., Ltd.

Weifeng Chen  
Huawei Technologies Co., Ltd.

Yijie Chen  
Huawei Technologies Co., Ltd.

Liang Shi  
Huawei Technologies Co., Ltd.

Peter Pietzuch  
Imperial College London

Lei Chen  
Hong Kong University of Science and Technology

Abstract

Reinforcement learning (RL) trains many agents, which is resource-intensive and must scale to large GPU clusters. Different RL training algorithms offer different opportunities for distributing and parallelising the computation. Yet, current distributed RL systems tie the definition of RL algorithms to their distributed execution: they hard-code particular distribution strategies and only accelerate specific parts of the computation (e.g. policy network updates) on GPU workers. Fundamentally, current systems lack abstractions that decouple RL algorithms from their execution.

We describe MindSpore Reinforcement Learning (MSRL), a distributed RL training system that supports distribution policies that govern how RL training computation is parallelised and distributed on cluster resources, without requiring changes to the algorithm implementation. MSRL introduces the new abstraction of a fragmented dataflow graph, which maps Python functions from an RL algorithm’s training loop to parallel computational fragments. Fragments are executed on different devices by translating them to low-level dataflow representations, e.g. computational graphs as supported by deep learning engines, CUDA implementations or multi-threaded CPU processes. We show that MSRL subsumes the distribution strategies of existing systems, while scaling RL training to 64 GPUs.

1 Introduction

Reinforcement learning (RL) solves decision-making problems in which agents continuously learn policies, typically represented as deep neural networks (DNNs), on how to act in an unknown environment [48]. RL has achieved remarkable outcomes in complex real-world settings: in game play, AlphaGo [46] has defeated a world champion in the board game Go; in biology, AlphaFold [20] has predicted three-dimensional structures for protein folding; in robotics, RL-based approaches have allowed robots to perform tasks such as dexterous manipulation without human intervention [14].

The advances in RL come with increasing computational demands for training: AlphaStar trained 12 agents using 384 TPUs and 1,800 CPUs for more than 44 days to achieve grandmaster level in StarCraft II game play [50]; OpenAI Five trained to play Dota 2 games for 10 months with 1,536 GPUs and 172,800 CPUs, defeating 99.4% of human players [3]. Distributed RL systems must therefore scale in terms of agent numbers, the amount of training data generated by environments, and the complexity of the trained policies [3, 20, 50].

Existing designs for distributed RL systems, e.g. SEED RL [9], ACME [17], Ray [33], RLlib [24] and Podracer [15] hardcode a single strategy to parallelise and distribute an RL training algorithm based on its algorithmic structure. For example, if an RL algorithm is defined as a set of Python functions for agents, learners and environments, a system may directly invoke the implementation of an agent’s act() function to generate new actions for the environment.

While integrating an RL algorithm’s definition with its execution strategy simplifies the design and implementation, it means that systems fail to parallelise and distribute RL algorithms in the most effective way:

1) When parallelising the training computation, most RL systems only accelerate the DNN computation on GPUs or TPUs [9, 17, 24] using current deep learning (DL) engines, such as PyTorch [38], TensorFlow [13] or MindSpore [18]. Other parts of the RL algorithm, such as action generation, environment execution and trajectory sampling, are left to be executed as sequential Python functions on worker nodes, potentially becoming performance bottlenecks.

Recently, researchers have investigated this unrealised acceleration potential: Podracer [15] uses the JAX [11] compilation framework to vectorise the Python implementation of RL algorithms and parallelise execution on GPUs and TPUs; WarpDrive [22] can execute the entire RL training loop on a GPU when implemented in CUDA; and RLlib Flow [25] uses parallel dataflow operators [54] to express RL training. While these approaches accelerate a larger portion of RL algorithms, they force users to define algorithms through custom APIs e.g. with a fixed set of supported computational operators.
(2) For the distribution of the training computation, current RL systems are designed with particular strategies in mind, i.e. they allocate algorithmic components (e.g. actors and learners) to workers in a fixed way. For example, SEED RL [9] assumes that learners perform both policy inference and training on TPU cores, while actors execute on CPUs; ACME [17] only distributes actors and maintains a single learner; and TLeague [47] distributes learners but co-locates environments with actors on CPU workers. Different RL algorithms deployed in different resources will exhibit different computational bottlenecks, which means that a single strategy for distribution cannot be optimal.

We observe that the above challenges originate from the lack of an abstraction that separates the specification of an RL algorithm from its execution. Inspired by how compilation-based DL engines use intermediate representations (IRs) to express computation [7, 52], we want to design a new distributed RL system that (i) supports the execution of arbitrary parts of the RL computation on parallel devices, such as GPUs and CPUs (parallelism); and (ii) offers flexibility how to deploy parts of an RL algorithm across devices (distribution).

We describe MindSpore Reinforcement Learning (MSRL), a distributed RL system that decouples the specification of RL algorithms from their execution. This enables MSRL to change how it parallellises and distributes different RL algorithms to reduce iteration time and increase the scalability of training. To achieve this, MSRL’s design makes the following contributions:

(1) Execution-independent algorithm specification (§3). In MSRL, users specify an RL algorithm by implementing its algorithmic components (e.g. agents, actors, learners) as Python functions in a traditional way. This 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.

(2) Fragmented dataflow graph abstraction (§4). From the RL algorithm implementation, MSRL constructs a fragmented dataflow graph (FDG). An FDG is a dataflow representation of the computation that allows MSRL to map the algorithm to heterogeneous devices (CPUs and GPUs) at deployment time. The graph consists of fragments, which are parts of the RL algorithm that can be parallellised and distributed across devices.

MSRL uses static analysis on the algorithm implementation to group its functions into fragments, instances of which can be deployed on devices. The boundaries between fragments are decided with the help of user-provided partition annotations in the algorithm implementation. Annotations specify synchronisation points that require communication between fragments that are replicated across devices for increased data-parallelism.

(3) Fragment execution using DL engines (§5). For execution, MSRL deploys hardware-specific implementations of fragments, permitting instances to run on CPUs and/or GPUs. MSRL supports different fragment implementations: CPU implementations use regular Python code, and GPU implementations are generated as compiled computation graphs of DL engines (e.g. MindSpore or TensorFlow) if a fragment is implemented using operators, or they are implemented directly in CUDA. MSRL fuses multiple data-parallel fragments for more efficient execution.

(4) Distribution policies (§6). Since the algorithm implementation is separated from execution through the FDG, MSRL can apply different distribution policies to govern how fragments are mapped to devices. MSRL supports a flexible set of distribution policies, which subsume the hard-coded distribution strategies of current RL systems: e.g. a distribution policy can distribute multiple actors to scale the interaction with the environment (like Acme [17]); distribute actors but move policy inference to learners (like SEED RL [9]); distribute both actors and learners (like Sebulba [15]); and represent the full RL training loop on a GPU (like WarpDrive [22] and Anakin [15]). As the algorithm configuration, its hyperparameters or hardware resources change, MSRL can switch between distribution policy to maintain high training efficiency without requiring changes to the RL algorithm implementation.

We evaluate MSRL experimentally and show that, by switching between distribution policies, MSRL improves the training time of the PPO RL algorithm by up to $2.4 \times$ as hyperparameters, network properties or hardware resources change. Its FDG abstraction offers more flexible execution without compromising training performance: MSRL scales to 64 GPUs and outperforms the Ray distributed RL system [33] by up to $3 \times$.

2 Distributed Reinforcement Learning

Next we give background on reinforcement learning (§2.1), discuss the requirements for distributed 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 unknown environment. The agent’s goal is to learn a policy that maximises the cumulative reward based on the feedback from the environment (see Fig. 1): 1 policy inference: an agent performs an inference computation on a policy, which maps the environment’s state to an agent’s action; 2 environment execution: the environment executes the actions, generating trajectories, which are sequences of (state, reward) pairs produced by the policy; 3 policy training: finally, the agent improves the policy by training it based on the received reward.

RL algorithms that train a policy fall into three categories: (1) value-based methods (e.g. DQN [32]) use a deep neural network (DNN) to approximate the value function, i.e. a map-
Beginning of the expected return to perform a given action in a state. Agents learn the values of actions and select actions based on these estimated values; (2) policy-based methods (e.g. Reinforce [51]) directly learn a parametrised policy, approximated by a DNN, for selecting actions without consulting a value function. Agents use batched trajectories to train the policy by updating the parameters in the direction of the gradient that maximises the reward; and (3) actor–critic methods (e.g. PPO [45], DDPG [26], A2C [31]) combine the two by learning a policy to select actions (actor) and a value function to evaluate selected actions (critic).

Multi-agent reinforcement learning (MARL) has multiple agents, each optimising its own cumulative reward when interacting with the environment or other agents (see Fig. 1). A3C [31] executes asynchronously on separate environment instances; MAPPO [53] extends PPO to a multi-agent setting in which agents share a global parametrised policy. During training, each agent maintains a central critic that takes joint information (e.g. state/action pairs) as input, while learning a policy that only depends on local state.

2.2 Requirements for distributed RL systems

The benefits of RL come at the expense of its computational cost [37]. Complex settings and environments require the exploration of large spaces of actions, states and DNN parameters, and these spaces grow exponentially with the number of agents [35]. Therefore, RL systems must be scalable by exploiting both the parallelism of GPU acceleration and a large number of distributed worker nodes.

There exists a range of proposals how RL systems can parallelise and distribute RL training: for single-agent RL, environment execution (2 in Fig. 1), policy inference and training (1+2) can be distributed across workers [15, 17, 33], potentially using GPUs [9, 15]; for MARL, agents can be distributed [24, 25, 33, 43] and exchange training state using communication libraries [28, 36]. In general, environment instances can execute in parallel [15, 31] or be distributed [8] to speed up execution.

Not a single of the above strategies for parallelising and distributing RL training is optimal, both in terms of achieving the lowest iteration time and having the best scalability, for all possible RL algorithms. For different algorithms, training workloads and hardware resources, the training bottlenecks shift: e.g. our measurements show that, for PPO, environment execution (2) takes up to 98% of execution time; for MuZero [44], a large MARL algorithm with many agents, environment execution is no longer the bottleneck, and 97% of time is spent on policy inference and training (1+3).

Instead of hardcoding a particular approach for parallelising and distributing RL computation, a distributed RL system design should provide the flexibility to change its execution approach based on the workload. This leads us to the following requirements that our design aims to satisfy:

1) Execution abstraction. The RL system should have a flexible execution abstraction that enables it to parallelise and distribute computation unencumbered by how the algorithm is defined. While such execution abstractions are commonplace in compilation-based DL engines [4, 7, 11, 42], they do not exist in current RL systems (see §2.3).

2) Distribution strategies. The RL system should support different strategies for distributing RL computation across worker nodes. Users should be permitted to specify multiple distribution strategies for a single RL algorithm, switching between them based on the training workload, without having to change the algorithm implementation. Distribution strategies should be applicable across classes of RL algorithms.

3) Acceleration support. The RL system should exploit the parallelism of devices, such as multi-core CPUs, GPUs or other AI accelerators. It should support the full spectrum from fine-grained vectorised execution to coarse-grained task parallel tasks on CPUs cores. Acceleration should not be restricted to policy training and inference only (1+2) but cover the full training loop, including environment execution (2) [22].

4) Algorithm abstraction. Despite decoupling execution from algorithm specification, users expect familiar APIs for defining algorithms around algorithmic components [10, 12, 21], such as agents, actors, learners, policies, environments etc. An RL system should therefore provide standard APIs, e.g. defining the main training loop in terms of policy inference (1), environment execution (2) and policy training (3).

2.3 Design space of existing RL systems

To understand how existing RL systems support the requirements above, we survey the design space of RL systems. Existing designs can be categorised into three types, function-based, actor-based and dataflow-based (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 using direct function calls. For example, Acme [17] and SEED RL [9] organise algorithms as actor/learner functions; RLGraph [43] uses a component abstraction, and users register Python callbacks to define functionality. Distributed execution is delegated to backend engines, e.g. TensorFlow [13], PyTorch [38], Ray [33].

(b) **Actor-based** RL systems execute algorithms through message passing between a set of (programming language) actors deployed on worker nodes (see Fig. 2b). Ray [33] defines algorithms as parallel tasks in an actor model. Tasks are distributed among nodes using remote calls. Defining control flow in a fully distributed model, however, is burdensome. To overcome this issue, RLlib [24] adds logically centralised control on top of Ray. Similarly, MALib [56] offers a high-level agent/evaluator/learner abstraction for population-based MARL algorithms (e.g. PSRO [34]) using Ray as a backend.

(c) **Dataflow-based** RL systems define algorithms through data-parallel operators, which are mapped to GPU kernels or distributed tasks (see Fig. 2c). Operators are typically predefined, and users must choose from a fixed set of APIs. For example, Podracer [15] uses JAX [11] to compile subroutines as dataflow operators for distributed execution on TPUs. WarpDrive [22] defines dataflow operators as CUDA kernels, and executes the complete RL training loop using GPU thread blocks. RLlib Flow [25] uses distributed dataflow operators, implemented as iterators with message passing. Tab. 1 considers how well these approaches satisfy our four requirements from §2.2:

**Execution abstraction.** Function- and actor-based systems execute RL algorithms directly through implemented (Python) functions and user-defined (programming language) actors, respectively. This prevents systems from applying optimisations how RL algorithms are parallelised or distributed. In contrast, dataflow-based systems execute computation using pre-defined/complied operators [15, 25] or CUDA kernels [22], which offers optimisation opportunities.

**Distribution strategies.** Most function-based systems adopt a fixed strategy that distributes actors to parallelise the environment execution (1+2 in Fig. 1) with only a single learner. Actor-based systems distribute stateful actors and stateless tasks across multiple workers, often using a greedy scheduler without domain-specific planning.

Dataflow-based systems typically hardcode how dataflow operators are assigned to workers. Podracer [15] provides two strategies: Anakin co-hosts an environment and an agent on each TPU core; Sebulba distributes the environment, learners and actors on different TPU cores; and RLlib Flow [25] shards dataflow operators across distributed Ray actors.

**Acceleration support.** Most RL systems only accelerate DNN policy inference and training (1+2). Some dataflow-based systems (e.g. Podracer [15] and WarpDrive [22]) also accelerate other parts of training, requiring custom dataflow implementations: e.g. Podracer can accelerate the environment execution (2) on TPU cores; WarpDrive executes the entire training loop (1+2) on a single GPU using CUDA.

**Algorithm abstraction.** Function-based RL systems make it easy to provide intuitive actor/learner/env APIs. Actor-based RL systems often provide lower-level harder-to-use APIs for distributed components (e.g. Ray’s get/wait/remote [33]); higher-level libraries (e.g. RLlib’s PolicyOptimizer API [24]) try to bridge this gap. Dataflow-based systems come with fixed dataflow operators, requiring users to rewrite algorithms. For example, JAX [11] provides the vmap operator for vectorisation and pmap for single-program multiple-data (SPMD) parallelism.

In summary, there is an opportunity to design a new RL system that combines the usability of function-based systems with the acceleration potential of dataflow-based systems. Such a design requires a new execution abstraction that retains the flexibility to apply different acceleration and distribution strategies on top of an RL algorithm specification.

---

<table>
<thead>
<tr>
<th>Type</th>
<th>System</th>
<th>Execution</th>
<th>Distribution</th>
<th>Acceleration</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function-based</td>
<td>SEED RL [9]</td>
<td>Python functions</td>
<td>environment only</td>
<td>DNNs</td>
<td>actor/learner/env</td>
</tr>
<tr>
<td></td>
<td>Acme [17]</td>
<td>Python components</td>
<td>delegated to backend</td>
<td></td>
<td>agent</td>
</tr>
<tr>
<td></td>
<td>RLGraph [43]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Actor-based</td>
<td>Ray [33]</td>
<td>task (stateless)</td>
<td>local scheduler</td>
<td>DNNs</td>
<td>Python functions</td>
</tr>
<tr>
<td></td>
<td>RLlib [24]</td>
<td>actor (stateful)</td>
<td>global scheduler</td>
<td></td>
<td>with Ray API [33]</td>
</tr>
<tr>
<td></td>
<td>MALib [56]</td>
<td></td>
<td>RPC</td>
<td></td>
<td>agent/actor/learner/env</td>
</tr>
<tr>
<td></td>
<td>WarpDrive [22]</td>
<td>predefined</td>
<td>sharded dataflow</td>
<td>DNNs</td>
<td>Operator API</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dataflow operators</td>
<td>operators with Ray tasks [33]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fragmented</td>
<td>MSRL</td>
<td>heterogeneous</td>
<td>dataflow partitioning</td>
<td>funcs/operators/</td>
<td>agent/actor/learner/env</td>
</tr>
<tr>
<td>dataflow graph</td>
<td></td>
<td>dataflow</td>
<td></td>
<td>DNNs/envs</td>
<td></td>
</tr>
</tbody>
</table>

Tab. 1: Design space of distributed RL systems
Algorithm 1: MAPPO algorithm in MSRL

```python
class MAPPOAgent(Agent):
    def act(self, state):
        return self.actors.act(state)
def learn(self, sample):
    return self.learner.learn(sample)

class MAPPOActor(Actor):
    def act(self, state):
        action = self.actor_net(state)
        return action

class MAPPOLearner(Learner):
    def learn(self, sample):
        reward, new_state = MSRL.env_step(action)
        return reward, new_state

class MAPPOTrainer(Trainer):
    def train(self, episode):
        for i in range(self.duration):  # @MSRL. fragment (type='Step', ops=['AllGather'],
            state = MSRL.env_reset()  # data=[state])
            for j in range(self.iter):
                reward, new_state = MSRL.agent_act(state)
                # @MSRL. fragment (type='AllGather', ops=['Step'], data=[state])
                sample = MSRL.env_step(action)  # a learner samples from the
                loss += self.mappo_net_train(action, state, adv, r)
            return loss / self.iter

    def learn(self, sample):
        # @MSRL. fragment (type='Learner', ops=['AllGather'], data=[sample])
        return loss / self.iter

mappo_algorithm_config = {
    'agent': {'num': 4, 'type': 'MAPPOAgent',
              'actor': {'num': 1, 'type': 'MAPPOActor',
                        'policy': 'MAPPOActorNet', 'env': True},
              'learner': {'num': 1, 'type': 'MAPPOLearner',
                          'policy': 'MAPPOLearnerTrain',
                          'params': {'name': 'MPE'}},
    'workers': [198.168.152.19, 198.168.152.20, [...],
                'GPUS_per_worker': 2},
    'distribution_policy': 'Single_learner_coarse'}

mappo_deployment_config = {
    'actor': {'num': 1, 'type': 'MAPPOActor',
              'policy': 'MAPPOActorNet', 'env': True},
    'learner': {'num': 1, 'type': 'MAPPOLearner',
                'policy': 'MAPPOLearnerTrain',
                'params': {'name': 'MPE'}}
```

3 MSRL API

Next, we introduce MSRL’s API and give an example of how it can implement the multi-agent PPO (MAPPO) algorithm [53]. The API separates the RL algorithm’s logic from deployment and execution considerations.

A user expresses an RL algorithm through familiar concepts, such as agents, actors, learners, trainers and environments. An agent consists of actors and learners: actors collect data from the environment, and learners manage the training policy. A trainer provides the training loop logic, specifying the interaction between actors and learners for each agent.

MSRL offers two types of APIs: (i) a component API defines the behaviour of each algorithmic component, defined through abstract classes. For example, `Actor.act()` specifies the interaction between actors and environments; `Learner.learn()` implements the training logic of the neural network; a training loop is defined through `Trainer.train()`, and (ii) an interaction API allows components to interact. For example, an actor submits the collected training data using `MSRL.replay_buffer_insert()`; a learner samples from the replay buffer using `MSRL.replay_buffer_sample()`.

MSRL determines how the algorithm is deployed through two configurations: (1) an algorithm configuration defines the logical components (i.e. agents, actors, learners etc.) and their hyper-parameters; and (2) a deployment configuration refers to a distribution policy and the devices.

Example. Alg. 1 sketches the MAPPO implementation. Its policies and DNNs are omitted for simplicity. `MAPPOAgent` (line 1) defines the agent behaviour: it interacts with the environment through `MAPPOActor` (line 6), and performs the policy training with `MAPPOLearner` (line 13). It then collects experiences from the environment with a given policy (lines 8–10), which is used to optimise the DNN model (lines 15–21). To define behaviour, `MAPPOAgent`, `MAPPOActor` and `MAPPOLearner` inherit from base classes, overriding abstract methods with the algorithm’s logic.

`MAPPOTrainer` defines the training loop (line 23) and uses MSRL’s interaction API. After entering the loop (line 25), it resets the environment using MSRL.env.reset(). It then uses `MSRL.agent_act()` to collect the experience (line 29) from `MAPPOActor`, and saves it in the replay buffer (line 30). Finally, the trainer calls `MSRL.agent_learn()` to execute the `learn()` function from `MAPPOLearner`.

Partition annotations. Despite defining an RL algorithm in a classical way with actors, learners and trainers, MSRL must be able to decompose the algorithm and distribute computation across devices. Since auto-distributing code is a hard problem [1, 2, 41], MSRL relies on user annotations in the code to help identify suitable partitioning boundaries.

The partition annotations in lines 9, 11, 27, 32, 34 denote possible boundaries in the algorithm for parallel and distributed execution. Each annotation specifies the type of code fragment and what dependent data must be transferred at the boundary when computation is distributed. Code between two consecutive partition annotations thus becomes a self-contained fragment, which can be assigned to a device.

As an example, consider the fragment defined by the annotations in lines 9 and 11: the first annotation stipulates that an action can be received from the previous fragment before interacting with the environment; the second annotation sends the reward and new state to the next fragment on another device. As we explain in the next section, this enables MSRL to distribute the environment simulation across devices.

The algorithm configuration (lines 35–43) is a Python dictionary that defines how to instantiate the algorithmic components (lines 35). It specifies their number, type and hyper-parameters, which MSRL requires for execution (see §5). In
this example, the training uses 4 agents, each with 1 actor and 1 learner; and each actor interacts with 32 environments.

The deployment configuration (lines 44–47) specifies how the computation is executed through a distribution policy. With the help of the partition annotations, the distribution policy governs the partitioning of the algorithmic components and their distribution. We discuss partitioning and distribution policies in §4 and §6, respectively.

4 Translation to Fragmented Dataflow Graphs

We now describe how MSRL uses fragmented dataflow graphs to decouple the parallelisation and distribution of RL algorithms from their implementation.

4.1 Fragmented dataflow graphs

MSRL must partition an RL algorithm so that it can be distributed across workers. Since it is challenging to parallelise and distribute the algorithm implementation directly, MSRL introduces a new execution abstraction by representing the algorithm as a fragmented dataflow graph (FDG): each node in the FDG is a potentially data-parallel fragment of code, which can be deployed on workers with GPUs and CPUs. FDGs thus allow MSRL to execute the RL computation on different devices depending on the allocation of fragments. MSRL distributes fragments to devices on remote workers, which then exchange data over the network; or fragments may be co-located on a single worker and use shared memory communication.

Fig. 3 gives a high-level overview of an FDG, which consists of multiple dataflow fragments (dashed square nodes). A fragment encompasses the dataflow graph of the corresponding code with two interfaces: an entry and an exit interface. These interfaces implement the functionality for sharing data with other fragments: if fragments are located on the same worker, they can directly share data structures; if they belong to different workers, they can use network communication operators. Depending on the communication method, the interface may be blocking (e.g. when using shared data structures) or non-blocking (e.g. for asynchronous communication).

Each fragment must be allocated to a device. Depending on how a fragment’s code is implemented, fragments require specific hardware resources for execution. For example, the actor and the policy fragments in Fig. 3 are implemented using the data-parallel operators of a DL engine, and thus require a GPU for execution; the environment fragment has a native Python implementation that executes on a CPU.

MSRL parallelises the execution of fragment code by creating multiple instances of it and assigning each instance to a separate device. All such replicated fragments share the same interface, allowing MSRL to use the communication operation from the partition annotation to synchronise data (see §3). For example, if the fragment denoted by the partition annotations in lines 32 and 34 in Alg. 1 is replicated, the instances synchronise using the AllGather method.

4.2 Deployment using fragment dataflow graphs

When deploying an RL algorithm as an FDG, two dimensions impact execution performance (see Fig. 4):

(1) Fragment granularity refers to a fragment’s code size, which affects device utilisation. A small fragment may under-utilise a GPU; a large one may exhaust GPU memory.

The fragment granularity also determines the ratio between computation and communication. The frequency and amount of data exchanged during synchronisation between fragments is a major source of overhead, often limiting scalability. In general, coarser fragments require less synchronisation, which reduces communication overhead, but they diminish opportunities for parallelism. For example, multiple fragments may exchange policy parameters (e.g. weights) or the replay buffer (i.e. trajectories) frequently at each step; alternatively, they may batch data from multiple steps and communicate at each episode to increase network efficiency.

FDGs can thus subsume the existing execution strategies of current RL systems. Based on the partition annotations, an FDG may fuse an actor and environment into one CPU-based fragment, and a learner into a GPU-based fragment (e.g. as proposed by Acme [17]); alternatively, it can create a coarser-grained GPU fragment by moving the DNN from the CPU fragment to the learner, which accelerates policy inference (as done by SEED RL [9]); at the coarsest granularity, an even larger fragment may contain the actor, learner, DNNs and environment, which accelerates the whole training loop (as done by WarpDrive [22] or Anakin [15]).

(2) Fragment co-location refers to the mapping of fragments to devices on the same worker. Co-locating fragments avoids network communication (Ethernet or InfiniBand) for synchronisation. Instead, it uses intra-node communication (NVLink...
Algorithm 2  Fragmented dataflow graph generation

1: function GENERATE_FDG(algorithm)
2: FDG ← \{\}
3: DFG ← generate_dfg(algorithm)
4: boundary ← annotation_parser(annotations)
5: common_node ← label_common_node(DFG, boundary)
6: for node in common_node do
7:   sub_graph ← graph_traversal(node, DFG)
8:   fragment ← generate_interface(sub_graph, boundary)
9:   FDG ← FDG ∪ fragment
10: return FDG

Fig. 5: Example of FDG construction for MAPPO

Fig. 6: Overview of the MSRL architecture

or PCIe), which reduces latency and increases throughput. Whether co-location is feasible depends on the available resources (e.g. the number of GPUs per node) and fragment constraints (e.g. the resource needs of fragments).

Choosing the right trade-off between fragment granularity and co-location is key to achieving good performance. The FDG abstraction allows MSRL to make different trade-offs using a set of distribution policies. Next, we describe how MSRL transforms an RL algorithm into an FDG.

4.3 Building fragmented dataflow graphs

MSRL generates an FDG by partitioning the algorithm’s dataflow graph. Based on the user-specified partition annotations (§3), it splits the dataflow graph into fragments, which then exchange the data specified by the annotations.

Alg. 2 shows the algorithm to generate an FDG. First, it statically analyses the RL algorithm to construct a dataflow graph, and then it partitions the graph into fragments.

Recall that the partition annotations denote the boundaries between code segments and describe the data that must be transferred when the code is partitioned. Correspondingly, in the partitioned fragments, the nodes that represent the data specified by the partitioned annotation must be at the boundary of fragments. We refer to these nodes as common nodes in the dataflow graph. MSRL creates fragments by partitioning the graph at the common nodes.

Consider the partitioning of the MAPPO algorithms from Alg. 1 with the two annotations in lines 32 and 34, Fig. 5a shows its simplified dataflow graph, with the data described by the two annotations highlighted in red. By splitting the graph at the common nodes, it can be partitioned into two fragments (see Fig. 5b). In each fragment, the common nodes (circled in red) are connected to interface implementations, which are generated for a given communication method.

To perform the partitioning, Alg. 2 first parses the partition annotations to record the communication method and corresponding data for each fragment boundary (line 4) and uses this information to label the common nodes (line 5). It then traverses the dataflow graph from each common node (line 7). The traversal terminates when it reaches another common node or a leaf node, and the constructed subgraph forms a new fragment. The algorithm also duplicates the common nodes in the original dataflow graph and fragment graph. Since the subgraph between two common nodes can only belong to one fragment, the algorithm removes the subgraph from the original dataflow graph after constructing the fragment, removing it from the following search. Finally, it adds the implementation of the communication interfaces and connects the common nodes with the interfaces (line 8).

If the user does not provide any partition annotations, MSRL, by default, partitions the algorithm along the algorithmic components, i.e. actors, learners, environments etc. Each component’s input and output values are the common nodes in the dataflow graph. The communication method is chosen by analysing the dataflow between the components.

5 MSRL Architecture

We describe MSRL’s architecture, which follows a coordinator/worker design (see Fig. 6): (1) a user submits the RL algorithm implementation to a coordinator; (2) the coordinator generates the FDG and dispatches fragments to workers; and (3) workers generate executable code for the received fragments, which can be run by e.g. a DL engine.

5.1 Generating and dispatching fragments

The coordinator has two components:

The FDG Generator uses the FDG generation algorithm (Alg. 2) to partition the RL algorithm based on the partition annotations and to insert code for the interfaces.

To ensure the code is executable by a given DL engine, a user must implement parts of the RL algorithm using appropriate dataflow operators, such as Mul(), Div() etc. For the generated fragment interfaces, MSRL maintains a mapping from the operations defined by the partition annotations to the corresponding dataflow operators. For example, the AllGather annotation in Alg. 1 maps to a comm.AllGather operator [19] in the MindSpore DL engine.

The generated code is synthesised as part of the run()
5.2 Executing fragments on workers

Each worker also has two components:

The Code Generator produces executable code that is specific to a given execution environment, such as DL engine, for the fragments received from the coordinator. Based on the fragment’s AST, it also synthesises the interface implementations. Finally, it converts the AST to Python code using the unparse() function from the Python AST library.

When a worker is assigned multiple fragment replicas on a single GPU, it uses several CUDA streams concurrently, one for each fragment. This approach, however, incurs an overhead launching and scheduling streams and also leads to extra memory copies between the CPU and GPU. MSRL avoids this by fusing co-located fragments into a single one.
This leverages the support of DL engines to process data in a SIMD fashion: by batching tensors from multiple replicated fragments into a single tensor, they can be processed by the DL engine's data-parallel operators.

To fuse fragments, the Code Generator locates the AST node for the tensor in the fragment. Since the Python AST represents a tensor's shape and data as a list stored in the elts field, the Code Generator can create the appropriate tensor shape for a batched operator.

The Executor launches the generated fragment code on different devices based on their implementations. For example, code implemented using MindSpore operators can run on a GPU controlled by the MindSpore DL engine; pure Python fragments execute on CPU cores. For efficient GPU execution, DL engines such as MindSpore compile code into a computational graph with GPU kernels. This enables optimisations: e.g. independent operators can be scheduled in parallel, improving performance over sequential execution.

Communication between fragments is also handled by MindSpore operators, which automatically select a suitable implementation based on the device type and network topology. It may use NCCL [36] for multi-GPU collective communication and MPI [29] for InfiniBand traffic between workers.

### 6 Distribution Policies

A distribution policy governs how MSRL executes an RL algorithm by specifying the FDG partitioning and the mapping of fragments to workers and devices. In general, there is not a single policy that is optimal in all cases: the performance and applicability of a policy 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 switch easily between policies, either for the same RL algorithm or when using different algorithms.

Tab. 2 describes MSRL's six default policies, showing the trade-off between them. These policies largely follow the hard-coded distribution strategies of existing RL systems; further policies can be defined easily by expert users:

- **DP-A (Single learner/coarse)** replicates the actor and environment fragments but relies on a single learner. The DNN policy is replicated across the actors and learner, which only requires coarse synchronisation. This policy is therefore suitable for expensive environments that need scaling out, but small DNN models that can be synchronised in a batched fashion.

- **DP-B (Single learner/fine)** 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-A, it relies on fine-grained synchronisation: training data is exchanged at each step, instead of being batched per episode. For good performance, DP-B therefore requires high bandwidth connectivity between workers.

- **DP-C (Multiple learners)** 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-A 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-C is communication efficient.

- **DP-D (GPU only)** fuses the full RL training loop into a single GPU fragment. It is only applicable if the environment has a GPU implementation. Since the full RL training loop is accelerated, DP-D achieves the highest performance by eliminating the overhead of CPU/GPU switches.

- **DP-E (Environments)** dedicated one or more workers for the execution of environments. It is best suited when the RL training job executes computationally expensive CPU environments, e.g. complex physics simulations, which requires many dedicated CPU cores for execution.

Finally, **DP-F (Central)** introduces a separate fragment for a centralised policy pool [56] or parameter server [23]. It can support CTDE-based MARL algorithms (similar to Ray [33] and RLlib [24]) by workers sending local policy updates to a parameter server; or population-based MARL (similar to MALib [56]) by a worker maintaining a population of policies, selecting the optimal one and broadcasting it to other workers.

### 7 Evaluation

Our experimental evaluation answers the following questions: “what are the trade-offs between distribution policies?” ($\S$7.2); “what is MSRL’s performance compared to other systems?” ($\S$7.3); and “how well does MSRL scale?” ($\S$7.4).

#### 7.1 Experimental set-up

**Implementation and test-bed.** We implement MSRL on top of MindSpore [18]. The code is available at https://github.com/mindspore-ai/reinforcement. We conduct the experiments on both a cloud and a local cluster. As shown in Tab. 3, the cloud cluster has 16 VMs; the local cluster has 4 nodes. All machines run Ubuntu Linux 20.04 with CUDA 11.03, cuDNN 8.2.1, OpenMPI 4.0.3 and MindSpore 1.8.0.

**RL algorithms and environments.** We answer the above questions by focusing on three widely-used RL algorithms, Proximal Policy Optimization (PPO) [45], its multi-agent version, Multi-Agent PPO (MAPPO) [53], and Asynchronous Advantage Actor Critic (A3C) [30]. The policy uses a seven-layer DNN. As environments, we use three games from MuJoCo [49], an advanced physics simulation engine, and the Multi-Agent Particle Environment (MPE) [27], a set of mixed cooperative-competitive environments for MARL.

**Distribution policies.** With the above algorithms, we use the following distribution policies from Tab. 2: DP-A (Single
We investigate the impact of the number of actors (Single learner/coarse) and DP-C (Multiple learners), DP-D (GPU only) and DP-E (Environments).

**Metrics.** For PPO, we measure (i) the training time to reach a given reward value and (ii) the time for each episode. For MAPPO, as the problem size increases with the number of agents due to agent competition, we report training time against the problem complexity and the training throughput.

### 7.2 Trade-offs between distribution policies

First we observe the impact of changes to the RL workload and hardware resources on different distribution policies.

**Actors.** We investigate the impact of the number of actors used when training PPO to a reward of 3,000 with 200 environments. We report the trade-off between two policies, DP-A (Single learner/coarse) and DP-C (Multiple learners).

Fig. 7a shows the training time with 2 to 70 actors. DP-A scales better than DP-C with more actors: it converges faster with a higher actor count. DP-C achieves the best performance with 40 actors, and its performance deteriorates after that but it outperforms DP-A with fewer than 30 actors.

Since DP-A only has 1 learner, its training batch size is fixed. Increasing the number of actors therefore only distributes environment execution. In contrast, DP-C fuses actors and learners into single fragments. With more actors, it also adds learners and thus reduces the batch size for each learner. This adds randomness to the training [16] and affects converge speed. In general, DP-C needs further hyper-parameter tuning (e.g. of the learning rate) for better performance [16].

Next we compare the behaviour of two algorithms, PPO and A3C, under the same distribution policy DP-A. Fig. 7b shows the time per episode for up to 24 actors. (Note that A3C requires at least 2 actors.) Under DP-A, PPO’s episode time decreases as the number of actors increases; in contrast, A3C’s episode time stays constant with the actor count.

In PPO, adding actors increases the parallelism degree of environment execution and therefore reduces the workload per actor; in A3C, each actor only interacts with one environment, making its workload independent of the actor count. To reduce the time per episode for A3C, a new distribution policy could be written that distributes the actor among multiple devices, e.g. exploiting data- or task-parallelism.

**Environments.** We investigate the impact of changing the number of environments. When an agent interacts with more environments in parallel within one episode, it trains with more data, thus potentially improving training performance. We fix the number of actors to 50.

Fig. 7c shows the training time as we vary the number of environments from 100 to 600 under DP-A and DP-C. DP-C scales better than DP-A with more environments, and there is a cross-over point with 320 environments. The training time with DP-A increases with more environments but remains stable for DP-C. This is because DP-A’s actors send trajectories to the learner, which increases in communication overhead with more environments. In contrast, DP-C only communicates gradients and its communication overhead is fixed. Therefore the right distribution policy must be selected based on the environment count.

**Network latency.** We also examine the behaviour of DP-A and DP-C when deployed with PPO on clusters with different network latencies. We change the 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.

Fig. 7d reports training time. DP-C is more sensitive to network latency than DP-A. Its training time increases rapidly with higher network latency, while remaining stable for DP-A. Since DP-C leverages Mindspore’s data parallel model [18] to broadcast, aggregate and update gradients, it repeatedly transmits many small tensors; DP-A transmits the trajectory and DNN model weights as compact large tensors, performing data transmissions less frequently. However, DP-C transmits less data than DP-A and thus is more appropriate in clusters with low network latencies (below 2 ms).

**Cluster size.** Finally, we compare the performance of PPO under 3 distribution policies when increasing the number of GPUs (Fig. 8): DP-A and DP-B use a single learner but apply different synchronisation granularities; DP-C scales to multiple learners using data-parallelism. We use a fixed number of 320 Mujoco HalfCheetah [5] environments.

Fig. 8a shows the training time to reach a reward of 4,000 in the cloud cluster with up to 64 GPUs; Fig. 8b reports the time per episode. Compared to 1 GPU, all distribution policies reduce training time with more GPUs. With 64 GPUs, DP-A achieves the highest speed-up in training time (5.3×). Compared to that, DP-B exhibits a longer training time, which is lowest for 16 GPUs. DP-A 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 a large number of GPUs compared to DP-B, whose actor fragments must communicate with the learner at each step.

DP-C exhibits a different behaviour: with 16 GPUs, it achieves better performance than either DP-A and DP-B. This is because it distributes policy training: it trains smaller batches of trajectories on each device and aggregates the gradients from all devices to update the policy. Instead, DP-B and DP-A gather all batches and train them using 1 learner.

While DP-C is the best policy choice for 16 GPUs, it performs worse than DP-A for a larger cluster. With more GPUs, trajectory batches become smaller and aggregating the trained gradients becomes less efficient compared to training a single large batch. Therefore, although DP-C trains each episode faster than DP-A (see Fig. 8b), it requires more episodes to reach a similar reward value.

Note that DP-A and DP-B use the original implementation of the PPO algorithm [45], which uses a single learner to train the policy. As a result, scalability is constrained by the centralised policy training (6 in Fig. 1). To ignore this bottleneck in the algorithm, we also exclude the policy training time in Fig. 8b, labelled DP-A’ and DP-B’. Here, MSRL continues scaling even with large GPU numbers: when moving from 32 to 64 GPUs, performance increases by 25%.

In Figs. 8c and 8d, we repeat the same experiment on the local cluster, which has faster GPU and network connectivity (NVLINK, InfiniBand). When comparing Figs. 8c and 8a, we can see that DP-A now always performs better than DP-C, irrespective of the GPU count. Since the local cluster supports faster GPU communication, DP-A’s overhead is lower than in the cloud cluster. DP-A’s larger batches become more effective than DP-C’s training/aggregation over smaller batches.

**Conclusions:** As the GPU count, hyper-parameters or network properties change, the differences between the synchronisation granularity and frequency of distribution policies impact performance. MSRL’s ability to select a distribution policy at deployment time allows users to achieve the best performance in different configurations without having to change the algorithm implementation.

### 7.3 Performance against baseline systems

We want to put the absolute performance of MSRL into perspective by comparing against other RL systems that hard-code particular parallelisation and distribution strategies. For the comparison, we choose Ray [33], as a representative distributed RL training system, and WarpDrive [22], as a single-GPU system that parallels the complete RL training loop.

**Distributed training.** We compare against Ray [33] using PPO and A3C with DP-A on the local cluster. Both algorithms are implemented and tuned based on Ray’s public PyTorch-based implementation [40]. We measure the time per episode, which is dominated by actor and environment execution. Since the time spent on DNN training/inference is negligible, the fact that MSRL and Ray use different DL frameworks (MindSpore vs. PyTorch) has low impact.

Fig. 9a shows PPO’s time per episode. In this experiment, we distribute 320 environments evenly among the actors, i.e. each actor interacts with 320/#actors environments, and a single learner trains the DNN. As shown in the figure, MSRL’s episode time with 1 GPU is 2.5× faster than Ray’s because Ray executes the actor on the CPU, which then interacts with all environments sequentially. As the number of GPUs increases, both systems reduce episode time as each actor interacts with fewer environments. With 24 GPUs, it takes 3.85 s for MSRL to execute an episode compared to 11.38 s for Ray (3× speed-up). When actors interact with multiple environments, MSRL combines DNN inference into one operation through FDG fusion, exploiting GPU parallelism better. It also uses fragments to execute environment steps in parallel by launching multiple processes.

Fig. 9b shows A3C’s time per episode. Here, 1 learner performs gradient optimisation with gradients collected asynchronously from actors. Each actor interacts with 1 environment and computes gradients locally. The figure shows that both systems’ time per episode remains constant with more GPUs. Each GPU is mapped to one actor, and its workload remains unchanged. Again, MSRL executes actors 2.2× faster than Ray: since its distribution policy exploits asynchronous send/receive operations from NCCL, MSRL avoids further data copies between GPU and CPU. In contrast, Ray copies data to the CPU to communicate asynchronously.

**GPU only training.** Next, we deploy PPO with distribution policy DP-D, which fuses the training loop into a single GPU
fragment and replicates it for distributed execution. We use PPO with the MPE simple tag environment [27], which simulates a predator-prey game in which chaser agents are rewarded for catching runner agents. Runner agents are penalised for being caught by a chaser or colliding with runners.

After adapting the simple tag implementation to GPUs, 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 [22], which performs single-GPU end-to-end RL training.

Fig. 10a shows the training time on 1 GPU. Compared to WarpDrive, MSRL is $1.2 \times -2.5 \times$ faster when ranging from 20,000 to 100,000 agents. MSRL’s DL engine (MindSpore) compiles fragments to computational graphs, exploiting more parallelisation and optimisation opportunities.

Fig. 10b shows how MSRL scales to multiple GPUs (which is unsupported by WarpDrive) when each GPU trains 80,000 agents. The training time increases from 138 ms to 150 ms from 160,000 to 960,000 agents. Beyond that, it remains stable – the available connectivity bandwidth (NVLink, InfiniBand) now bounds the communication overheads between GPU fragments and devices.

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

7.4 Scalability

We explore how MSRL’s design scales as we increase the number of deployed agents for a MARL algorithm and the number of environments (thus increasing the amount of training data). We want to validate if MSRL’s approach introduces scalability bottlenecks.

Agents. We increase the number of agents for the MAPPO MARL algorithm with the MPE simple spread [27] environment, where $n$ agents interact and learn to cover $n$ landmarks (rewards) while avoiding collisions (penalties). In addition to training local observations, agents must also train global observations on how far the closest agent is to each landmark. This results in $O(n^3)$ observations with $n$ agents, thus quickly growing the computation complexity and GPU memory footprint [27]. We deploy the MAPPO agents on the cloud cluster using DP-A: each GPU on a worker trains 1 agent, and a dedicated worker executes all environments.

Fig. 11a shows the training time per episode for up to 64 GPUs against a sequential baseline (1 GPU). Due to its cubic complexity, the training time increases sharply both for the baseline and MSRL. With distributed training, MSRL’s training time grows more slowly than the baseline: when training 32 agents, MSRL improves the performance by $58 \times$; with 64 agents, the baseline exhausts GPU memory, while MSRL trains one episode in 23.8 mins.

Fig. 11b compares the throughput when training different numbers of agents. The throughput is measured as the amount of data trained per second (in MB/s). Increasing the agent number (i.e. GPU devices) significantly improves throughput, and the margin becomes larger with more GPUs: the throughput when training 64 agents is over 7,600 $\times$ higher than training 2 agents. This is because multiple GPUs train agents in parallel, and more agents result in more training data (i.e. a larger size of observations) per GPU.

Environments. Next we increase the number of environments and observe the impact on the statistical efficiency, i.e. the number of episodes to reach a given reward value. We assign 10 environments to each CPU and add more devices in the cloud cluster using DP-A. We expect MSRL to achieve higher statistical efficiency with more training data.

In Fig. 12, we show the reward along with the number of episodes for different environment counts. As expected, increasing the number of environments leads to a higher reward value. As more CPUs execute environments, more trajectories are used per episode, achieving a higher reward.

Conclusions: MSRL’s FDG abstraction does not deteriorate scalability. MSRL scales with a large number of data-intensive agents, handling the increase in communication between fragments without bottlenecks. As expected, a larger amount of data generated by more environments also increases the statistical efficiency of the training process.
8 Conclusions

We described MindSpore Reinforcement Learning (MSRL), a system that supports the flexible parallelisation and distribution of RL algorithms on multiple devices. MSRL separates the algorithm implementation from its execution through the abstraction of fragmented dataflow graphs (FDGs), which enable MSRL to support diverse distribution policies for allocating code fragments to devices. Our experimental results show the trade-offs when using different distribution policies.

References


