MALib: A Parallel Framework for Population-based Multi-agent Reinforcement Learning

Ming Zhou[1]*, Ziyu Wan[1], Hanjing Wang[1], Muning Wen[1], Runzhe Wu[1], Ying Wen[1]†, Yaodong Yang[2], Weinan Zhang[1] and Jun Wang[2]

[1]Shanghai Jiao Tong University, [2]University College London

June 3, 2021

Abstract

Population-based multi-agent reinforcement learning (PB-MARL) refers to the series of methods nested with reinforcement learning (RL) algorithms, which produces a self-generated sequence of tasks arising from the coupled population dynamics. By leveraging auto-curricula to induce a population of distinct emergent strategies, PB-MARL has achieved impressive success in tackling multi-agent tasks. Despite remarkable prior arts of distributed RL frameworks, PB-MARL poses new challenges for parallelizing the training frameworks due to the additional complexity of multiple nested workloads between sampling, training and evaluation involved with heterogeneous policy interactions. To solve these problems, we present *MALib*, a scalable and efficient computing framework for PB-MARL. Our framework is comprised of three key components: (1) a centralized task dispatching model, which supports the self-generated tasks and scalable training with heterogeneous policy combinations; (2) a programming architecture named *Actor-Evaluator-Learner*, which achieves high parallelism for both training and sampling, and meets the evaluation requirement of auto-curriculum learning; (3) a higher-level abstraction of MARL training paradigms, which enables efficient code reuse and flexible deployments on different distributed computing paradigms. Experiments on a series of complex tasks such as multi-agent Atari Games show that *MALib* achieves throughput higher than 40K FPS on a single machine with 32 CPU cores; 5× speedup than RLlib and at least 3× speedup than OpenSpiel in multi-agent training tasks. MALib is publicly available at https://github.com/sjtu-marl/malib.

1 Introduction

Training intelligent agents that can adapt to a diverse set of complex environments and agents has been a long-standing challenge. A feasible way to handle these tasks is multi-agent reinforcement learning (MARL) [**?**], which has shown great potentials to solve multi-agent tasks such as real-time strategy games [**?**], traffic light control [**?**] and ride-hailing [**?**]. In particular, the PB-MARL algorithms combine deep reinforcement learning (DRL) and dynamical population selection methodologies (e.g., game theory [**?**], evolutionary strategies [**?**]) to generate auto-curricula. In such a way, PB-MARL continually generates advanced intelligence and has achieved impressive successes in some non-trivial tasks, like Dota2 [**?**], StrarCraftII [**?**] and Leduc Poker [**?**]. However, due to the intrinsic dynamics arising from multi-agent and population, these algorithms have intricately nested structure and are extremely data-thirsty, requiring a flexible and scalable training framework to ground their effectiveness.

The deployment of PB-MARL shares some common procedures with conventional (MA)RL, but many challenges still remain, including the auto-curricula learning process and exponential

*The first three authors are core developers. Ming contributed to the system architecture design and development; Ziyu contributed to the algorithm implementations; Hanjing contributed to the data server decoupling.

†Correspondence to: Ying Wen <ying.wen@sjtu.edu.cn>.

compositional sampling. PB-MARL inherently comprises heterogeneous tasks, including simulation, policy inference, policy training and policy support expansion. All of these tasks are coupled to the underlying mutable policy combinations, which further complicates the PB-MARL. Figure 1 presents a typical case in PB-MARL, the learning process of Policy Space Response Oracle (PSRO) [?]. At each iteration, the algorithm will generate some policy combinations and executes independent learning for each agent. Such a nested learning process comprises rollout, training, evaluation in sequence, and works circularly until the algorithm finds the estimated Nash Equilibrium. We note that these sub-tasks perform highly heterogeneous, i.e., the underlying policy combination is different from agent to agent. Furthermore, the evaluation stage involves tremendous simulations that cross fast-growing joint policy sets. Thus, we believe these requirements make distributed computing unavoidable for achieving high efficiency in executions.

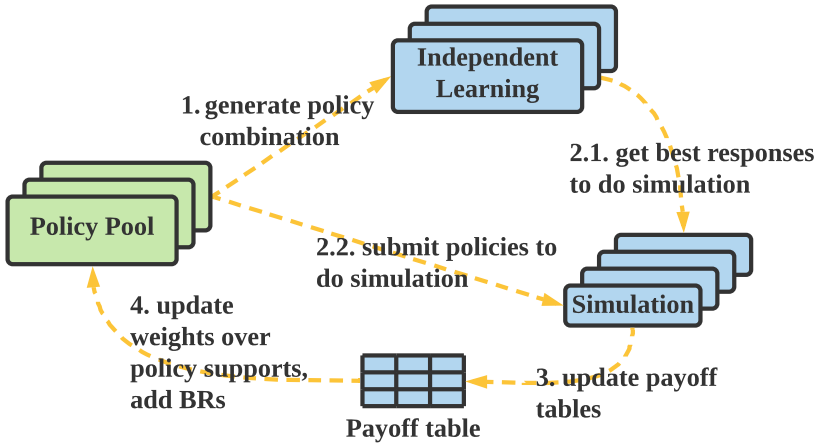


Figure 1: PSRO learning process.

Most of existing training frameworks are originally designed for single-agent RL, and few attempts have been made to miscellaneous types of training schema in MARL, especially PB-MARL. In single-agent DRL, to meet the requirements of distributed computing, there have been tremendous distributed frameworks proposed to solve the data processing problem, e.g., RLlib [?], SEED RL [?], Sample-Factory [?] and Acme [?]. Despite that single-agent RL methods can also be applied in multi-agent settings [?] by acting as independent learners [?], more effective training schemas like centralized training & decentralized execution [?, ?], self-play [?, ?, ?], population-based learning, etc. require the training performs in a non-independent style. Obviously, it requires extensive effort for the single-agent learning mode to handle such interactive requirements, and poses new challenges to the existing distributed reinforcement learning frameworks. Though works like OpenSpiel [?] have attempted to support self-play and PSRO, they merely consider the good PB-MARL abstractions and scalability. Therefore, we claim that existing frameworks cannot fully satisfy the new requirements brought by PB-MARL.

In this paper, we stress the necessity and unfulfilled requirements for deploying PB-MARL and provide our systematic solution. We summarize that the essentials of a MARL distributed framework should (1) provide a highly efficient controlling mechanism to support self-supervised auto-curricula training and heterogeneous policy interactions in PB-MARL; (2) provide a high-level abstraction of MARL training schema to simplify the implementation and a unifed and scalable training criteria. Based on the above analysis and comparisons, we present *MALib* as shown in Figure 2, a parallel framework that meets these requirements, to solve PB-MARL tasks in a high-parallelism style. We evaluate the performance of *MALib* from two distinct perspectives, i.e., system and algorithm. System performance is illustrated by the sample efficiency and throughput with increasing number of workers, while the algorithmic-side performance comparison is conducted over the reproduced algorithms with baselines, including typical PB-MARL algorithms and conventional MARL algorithms.

The main contributions of this work are summarized as follows:

1. We propose a centralized task dispatching model for PB-MARL, which achieves high flexibility in training tasks over auto-curricula policy combinations.
2. We propose an independent programming model, *Actor-Evaluator-Learner*, to improve the efficiency of data processing and asynchronous execution.
3. To improve the code reuse and compatibility of heterogeneous MARL algorithms, we abstract the MARL training schema, which also bridges the gap between single-point and multi-point optimization.
4. We also provide mainstream MARL algorithm implementations and verify system performance and algorithm performance on *MALib* with different system settings, including single machine and clusters.

2 Related Work

A fundamental challenge in MARL is that the agents tend to overfit other players [**?**], making it hard for the algorithms to achieve robust performance. To solve this problem, interacting with heterogeneous agents or diverse policies of co-players is unavoidable. PB-MARL is a feasible approach to solve this problem, and prior works include population-based training [**?**, **?**], self-play [**?**, **?**] and meta-game [**?**, **?**].

Another difficulty is the data processing, the same as all DRL tasks. For deep reinforcement learning, high throughput allows algorithms to achieve a faster convergence rate and high data efficiency. There are many distributed reinforcement learning algorithms/frameworks proposed in recent years [**?**, **?**, **?**]. Among them, a standard implementation is to design training and rollout workers in fully distributed control, i.e., the Actor-Learner model. Also, some of them try to mitigate the communication loads between CPU and GPU [**?**] to improve the single-machine performance. Despite the impressive successes in distributed RL, most of them require users to do extra parallel programming to fit their custom requirements. RLlib [**?**] solved this problem by building a DRL framework on the top of Ray [**?**], work in a logically centralized control manner. Furthermore, the solution to MARL tasks among these frameworks is to model the MARL tasks as single-agent tasks, which decreases the computing efficiency in more general MARL settings since it requires heterogeneous agent/policy interaction in the training process.

Apart from the efforts in DRL, there are also tremendous works that integrate distributed computing techniques into deep learning architectures. Frameworks like Pytorch [**?**] and Horovod [**?**] implemented their distributed training logic over MPI [**?**]. General distributed tools like Ray [**?**] and Fiber [**?**] provide a universal API for building distributed applications, which relieve users from parallel programming such as MPI and OpenMP [**?**]. There are also some works that focus on MARL implementation and abstraction [**?**, **?**], but most of them are too narrow to fit general MARL tasks, focusing on a specific domain [**?**]. As we claimed in the aforementioned content, the distributed computing support for MARL is necessary to the wider access of this exciting area.

Table 1: Comparison between *MALib* and existing distributed reinforcement learning frameworks from three dimensions.

Framework	**Single-Agent**	**Multi-Agent**	**Population Management**
RLlib [**?**]	✓	✓	×
SeedRL [**?**]	✓	×	×
Sample-Factory [**?**]	✓	✓	×
MALib	✓	✓	✓

To meet the distributed computing requirements of PB-MARL, we built our *MALib* on top of Ray and provided an efficient training framework. Table 1 presents the comparison between *MALib* and exiting distributed reinforcement learning framework from three dimensions, i.e., single-agent RL support, multi-agent RL support and population management. Despite some of

them support multi-agent RL algorithms, they are essentially independent learning, or require users' extra efforts to implement algorithms in other training paradigms. Furthermore, a key dimension for PB-MARL is the population management, i.e., maintaining a policy pool for each agent, support policy expansion, update policy distribution in auto-curriculum learning, etc. *MALib* considered these requirements and gives corresponding implementations.

3 Parallel Programming Abstractions for PB-MARL

In this section, we will give an introduction to our framework from three components: the *Centralized Task Dispatching* model, the *Actor-Evaluator-Learner* model and the abstractions of MARL training paradigms, as shown in Figure 2. With these key implementations, we tense *MALib* serve for PB-MARL in auto-curriculum learning task schedule, execution in high performance and implementation with high code reuse.

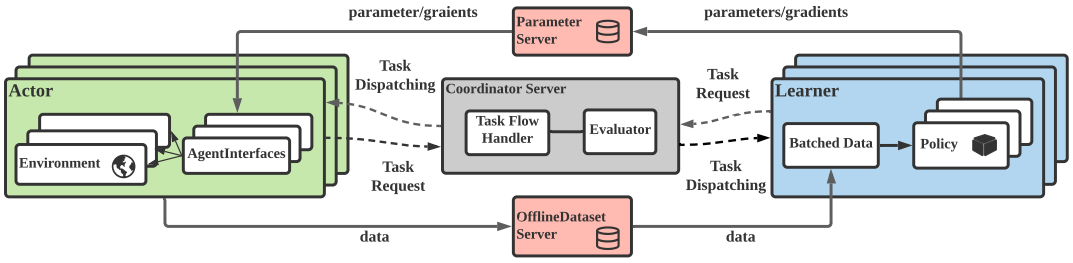


Figure 2: Overview of the *MALib* architecture. The *Coordinator Server* schedules the learning tasks; workers like *Actor* and *Learner* work in parallel and data dependencies are decoupled by *Parameter* and *OfflineDataset* servers. Actor is responsible for rollout/simulation tasks with k environments each, and Leaner is responsible for the optimization of a policy pool. The collected experiences are processed before being sent to the *OfflineDataset* server. After Learner/Actor completes tasks, it will send a task request to *Coordinator* server for evaluation or promote the generation of next learning stage.

3.1 Centralized Task Dispatching Model

As introduced in Section 2, parallelizations for RL in previous work can be roughly classified into the Fully Distributed Control (FDC) [**?**, **?**, **?**] and the Hierarchical Parallel Task (HPT) model [**?**] fixed training task flow and policy interaction manners. Though these frameworks have abstractions for RL tasks, the extraordinary types of MARL training schema limit their performance, so users have to make extra efforts for customization. Furthermore, the PB-MARL algorithms like PSRO [**?**] and AlpahRank [**?**] require mutation in policy combination and policy space expansion in auto-curricula, which are ignored in previous frameworks.

We propose the *Centralized Task Dispatching (CTD)* model to meet these requirements in PB-MARL. Figure 3 presents the comparisons between previous parallel control model and our CTD model. This figure borrows the flowcharts from RLlib to better explain our design in parallel task control. The same as RLlib, we implemented the CTD model on top of Ray [**?**], which allows Python-implemented tasks to be naturally distributed over a large cluster.

The CTD model considers both of the advantages from FDC and HPT models. Specifically, the CTD has a centralized controller **D** to update the description of underlying policy combinations iteratively and generate new learning tasks, then dispatches them to working processes (**A**, **B** and **C**). The working processes in CTD work independently but do not coordinate with each other like in FTD. Furthermore, the working processes also work in a semi-passive manner, i.e., they will send task requests to **D** after completing tasks, which differs from the HPT model where the working process is fully passive. In fact, the semi-passive execution can be highly performant since the working processes will not handle the centralized controller all the time, so that **D** can

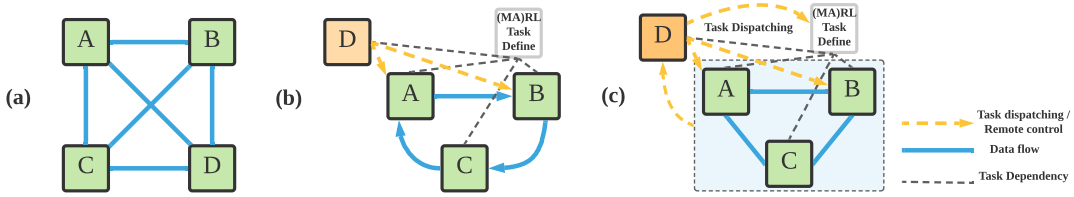


Figure 3: We abstract parallel processes as **A**∼**D** in this figure. Processes perform autonomous control in (a) *Fully Distributed Control Model (FDC)* and centralized control in (b) the *Hierarchical Parallel Task Model (HPT)* and (c) our *Centralized Task Dispatching Model (CTD)*. **D** represents the centralized control process which is responsible for task dispatching. The working processes (**A**∼**C**) in CTD execute in semi-passive manner, performing higher parallelism than the fully-passive execution in HPT.

work in highly parallelism to process more tasks to make sure the system run in high efficiency, especially for Python, which has a global interpreter lock. In our implementation, we modeled **D** as the *Coordinator Server*, and working processes **A**, **B** and **C** could be Actors, Learners and decoupled data servers.

Defining the Task Graph. The execution logic of the CTD model can be formulated as a closed-loop task graph for PB-MARL. In the beginning, we initialize a set of policy pools $\mathcal{P} = \{\mathcal{P}_i \mid i = 1, \ldots, M\}$ for each agent, while some of them can share the same policy pool. Then, each agent $\{a_j \mid j = 1, \ldots, N\}$ from the environment will choose one policy π_{a_j} from its belonging policy pool $\mathcal{P}_i$ to form policy combinations. We formulate a policy combination at intermediate training stage t as $\text{Comb}_t = \Pi_j^N \pi_{a_j}$. Based on the generated Comb_t, the coordinator will dispatch rollout tasks and training tasks to working processes. In general, the amount of tasks is determined by specific algorithms, e.g., PSRO generates N training tasks for N agents if the nested RL algorithm performs independent learning, or $m \leq N$ tasks for some centralized learning algorithms. Let $X(\theta_k)$ represent the collected data from rollout workers with policy parameters θ_k. Then for each rollout iteration, we have:

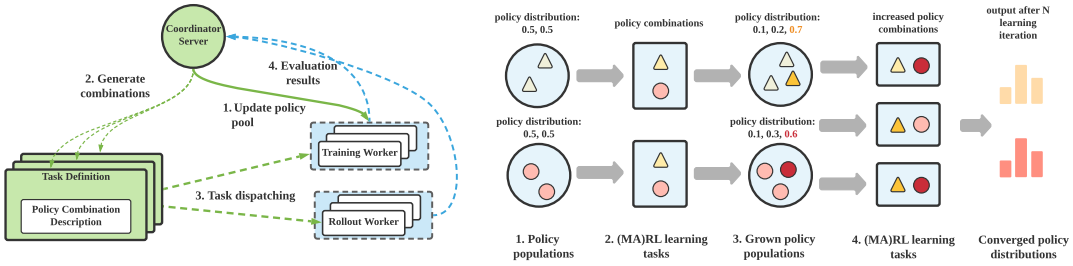
$$X(\theta_k) \sim P(s, a, s' \mid a \sim \text{Comb}_t(\theta_k)),$$

where θ_k represents the policy parameters at iteration k. For a given policy combination, a batch of collected data will be stored as $\mathcal{D} = \{X(\theta_k) \mid k = 1, \ldots, h\}$, and two parallel evaluation tasks for rollout and training will executed periodically as

$$\text{Eval}_{\text{rollout}} = f\left(X(\theta_k \leftarrow \theta')\right), \text{ and } \theta' = \underset{\theta}{\text{argmax}}\, \text{Eval}_{\text{train}} = f(X' \sim \mathcal{D} \mid \theta).$$

Until the global evaluator from the coordinator server reports staged stopping based on either one of them, then a new policy combination Comb_{t+1} will be produced with $\text{Comb}_{t+1} \sim G(\text{Eval}_{\text{rollout}}, \text{Eval}_{\text{train}})$, where G could be a specific evaluation function from PB-MARL algorithm like PSRO. Figure 4 shows the execution of circled task graph.

Decoupling the Task-Data Flow. The data flow mentioned here denotes the flow from data collecting to data sampling, also parameters push & pull. In general, a data flow is private to a policy combination, and the corresponding working processes perform in high efficiency as one-to-many (one learner to multiple actors). Although such mode has shown many advantages in prior distributed frameworks [**?**, **?**, **?**, **?**], it limits the parallelism in the PB-MARL case since the data dependencies could be many-to-many here, i.e., each rollout task is corresponding to policies from multiple learning processes whose learning paces differ to each one. We decouple the data flow from task execution using Parameter Server [**?**] and OfflineDataset Server, i.e., *Parameter Server* for parameters synchronous between Actors and Learners, OfflineDataset Server for data saving and sampling, as shown in Figure 2.

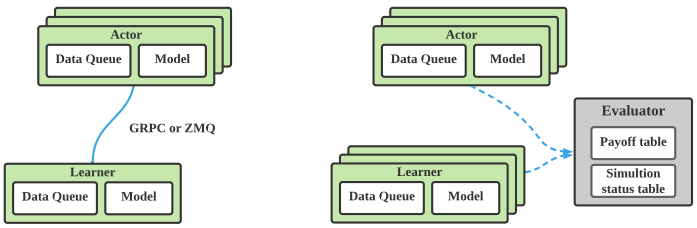


(a) execution graph **(b) population-based MARL learning process**

Figure 4: (a) The task execution graph of PB-MARL in *MALib*; (b) The auto-curriculum learning paradigm of PB-MARL. In general, a PB-MARL algorithm will start from an initialized policy set and the distribution over the policies (**step 1**), then find new policies (**step 2**) via (MA)RL algorithms. After that, the policy set will be updated with new policies, also the policy distribution (**step 3**), and follow with (MA)RL tasks over expanded policy combinations (**step 4**). This process will be cyclically executed until meeting the convergence such as an estimated Nash Equilibrium, then output the final policy distribution.

3.2 Actor-Evaluator-Learner Model

In single RL, it is common to decouple the training and rollout tasks [**?**, **?**, **?**, **?**] using the *Actor-Learner* model, where the Learner operates policy training and the Actor operates data collecting. Furthermore, the evaluation program interleaves the Learners and Actors. However, in the case of PB-MARL, such a design appears to be inadequate since there is no centralized evaluator to integrate evaluations of multiple learning tasks which is required by PB-MARL tasks. Thus, we propose the *Actor-Evaluator-Leaner* to meet this requirement, as shown in Figure 5. The *Evaluator* is nested with a payoff table to record the evaluation results of each policy combination. In general, the evaluation of a PB-MARL algorithm is built on top of the table, e.g., PSRO generates policy distribution over existing policies by estimating a Nash Equilibrium with this table.



(a) Actor-Learner model **(b) Actor-Evaluator-Learner model**

Figure 5: Comparison between Actor-Learner and Actor-Evaluator-Learner model.

3.3 Abstractions for MARL Training Paradigms

PB-MARL is nested with (MA)RL algorithms. In this section, we introduce a set of learners abstracted for different MARL paradigms. Our initial implementation offers five learners to meet the basic requirements, which also considers the asynchronous and synchronous training styles and some implementations of their respective MARL algorithms. Most previous works have tried to apply modular design to (MA)RL algorithm implementation and training. Still, they focus on the algorithm types that originate from RL, not the training paradigms, i.e.,

value-based or policy-gradient-based learners [**?**]. Though such a modular design presents a completed implementation logic to users, it has low reuse because of the nested training logic in algorithm implementation. We list four typical training paradigms here and present the corresponding abstractions.

Table 2: *MALib*'s training abstraction supports various of (MA)RL and PB-MARL algorithms.

Algorithm Family	**Independent**	**Centralized**	**Asynchronous**
Value Based	DQN [**?**]	QMIX [**?**]	Gorrila [**?**]
Actor Critic	A2C [**?**]	MAAC [**?**]	IMPALA [**?**]
Population-based Training	PSRO [**?**]	×	Pipeline-PSRO [**?**]

Independent Learning. It is basically equivalent to single-agent deep reinforcement learning, with little or no need to change the original framework. The independent learner serves for independent learning algorithms like DQN [**?**] and PPO [**?**]. In this learning paradigm, one learner for one policy, policy learning is independent to other agents (policies).

Centralized Learning. In MARL, centralized learning means the learning requires shared information from other agents. In MADDPG [**?**], the shared information indicates the state-action pairs from all agents, and each agent has its individual critic, which accepts the shared information to perform policy optimization. Another variant is QMIX [**?**], which differs from MADDPG in that all agents share a global critic to do simultaneous policy optimization. Gradient shared algorithms like Networked-agent learning [**?**] could also be framed as centralized learning. Though its authors claimed that their method is fully decentralized, it is explained from the view of optimization.

Asynchronous Learning. Methods like IMPALA [**?**] use multiple Learners to update policy networks in asynchronous manner. We provide an implementation to support this paradigm. Furthermore, this learning interface can be coped with the former two as a nested implementation.

Synchronous Learning. Synchronous learning interface makes the Actors and Learner work in sequence, like the conventional *Actor-Learner* model, i.e., support multiple Actors work for one Learner. Furthermore, this learning interface can be coped with the Independent and Centralized learning interfaces as a nested implementation.

We demonstrate the exiting (MA)RL algorithm families within our abstractions in Table 2. The synchronous learning dimension is eliminated from the table since it is the default training manner for conventional (MA)RL algorithms. Furthermore, we show the aforementioned training paradigms in Figure 6.

4 Evaluation

Special efforts are paid to optimize the learning performance of multi-agent tasks in the design of *MALib*. In this section, metrics including data throughput, sampling efficiency and training time are reported for a comprehensive understanding of *MALib* system performance. All the experiment results listed are obtained with one of the following hardware settings: (1) *System# 1*: A 32-core computing node with dual graphics cards. (2) *System# 2*: A two-node cluster with each node owns 128-core and a single graphics card. All the GPUs mentioned are of the same model (NVIDIA RTX3090). We present the primary results here, and readers can find more details in Appendix B.

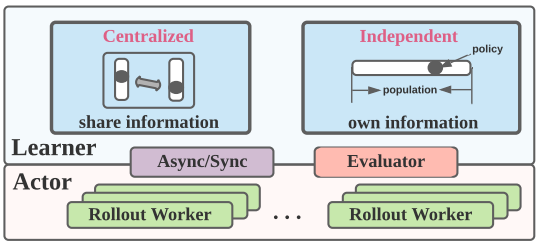


Figure 6: Depiction of scopes of independent/centralized and async/sync training diagrams. Independent and centralized paradigms have differences in data flow dependency: whether to share information among agents. And async and sync paradigms have differences in the task flow dependency: whether to strictly interleaving rollout and training.

4.1 Throughput Comparisons

We conduct the throughput evaluation on *MALib* and compare the results with some of the existing SOTA distributed RL frameworks, i.e. RLlib [**?**], Sample-Factory [**?**] and SEED RL [**?**]. Specially, Sample-Factory and SEED RL are highly tailored for TPU and GPU instances. Therefore, we repeat the group of experiments, with only CPU instances and GPU acceleration activated correspondingly.

Environment. As the environment for throughput comparison, we adopt the multi-agent version of Atari games (MA-Atari) from PettingZoo [**?**], which is a collection of 2D video games with more than one agent in each game. In our experiments, we use the two-players Pong game, with RGB-image frames in $12 \times 12 \times 3$ resolution.

Baselines and Settings. We choose IMPALA for SEED RL and RLlib, APPO for Sample-Factor and *MALib*. The evaluation of training throughput was conducted in different worker configurations over three minutes of continuous training, considering performance fluctuations caused by environment reset, data concatenation and other factors like threading lock. For each worker, we fixed the number of environments as 100. The number of workers ranges from 1 to 128 to compare the upper bound and bottleneck in parallelism performance of different frameworks.

Figure 7 shows the results of comparison on System# 1. Note that due to resource limitation, with only CPU cores, RLlib failed to launch with more than 32 workers while that threshold for GPU-accelerated RLlib is 8 workers on the same node. Despite of the extra abstraction layer introduced for tackling PB-MARL problems, the CPU version of *MALib* outperforms other frameworks in the conventional MA-Atari environment for the scalability to different size of worker groups. And *MALib* achieves comparable performance with Sample-Factory in the GPU-acceleratd settings, which is a framework specially tailored for training conventional RL algorithms on single GPU node.

4.2 Algorithm Implementation and Performance

We have implemented a series of algorithms in *MALib* as listed in Appendix A, including independent learning algorithms like DQN, PPO and SAC, along with MARL algorithms like MADDPG and QMIX. Furthermore, all of these algorithms can be applied to Population-based Training (PBT), self-play, which have been supported by *MALib*. For the algorithmic performance evaluation, we focus on the convergence rate, which is derived from sample efficiency and training time consumption. As for the evaluated algorithms, we use PSRO training for PB-MARL, MADDPG and QMIX for conventional MARL algorithms. With the consideration of fairness, we make the algorithm implementation from different frameworks consistent in network settings. Appendix B presents the details.

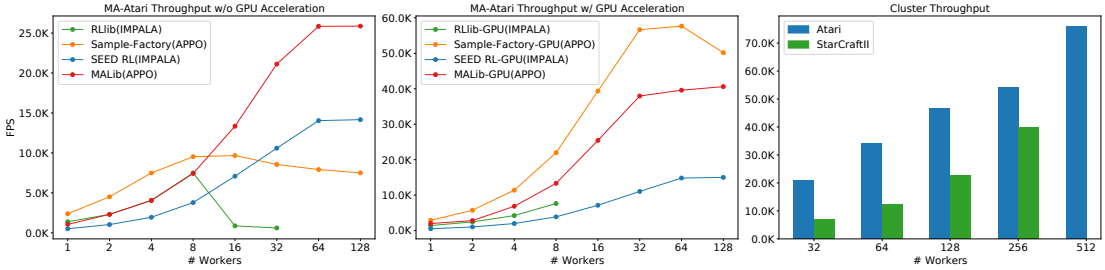
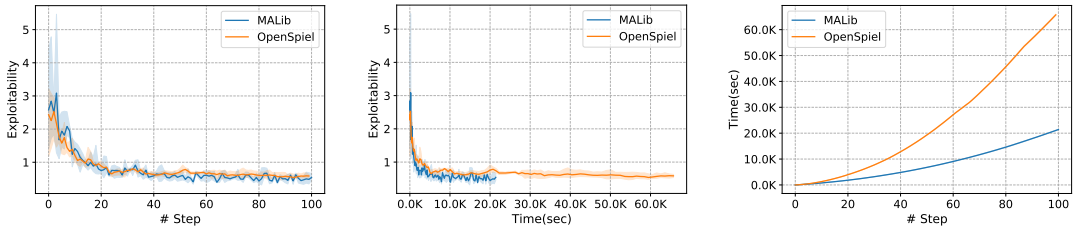


Figure 7: Throughput comparison among the existing RL frameworks and MALib. Due to resource limitation(32 cores, 256G RAM), RLlib fails under heavy loads(CPU case: #workers>32, GPU case: >8). MALib outperforms other frameworks with only CPU and achieves comparable performance with the highly tailored framework Sample-Factory with GPU despite higher abstraction introduced. To better illustrate the scalability of *MALib*, we show the MA-Atari and SC2 throughput on system#2 under different worker settings, the 512-workers group on SC2 fails due to resource limitation.



(a) (b) (c)

Figure 8: Comparisons of PSRO between *MALib* and OpenSpiel. (a) indicates that *MALib* achieves the same performance on exploitability as OpenSpiel; (b) shows that the convergence rate of *MALib* is 3× faster than OpenSpiel; (c) shows that *MALib* achieves a higher execution efficiency than OpenSpiel, since it requires less time consumption to iterate the same learning steps, which means *MALib* has the potential to scale up in more complex tasks that need run for much more steps.

Table 3: *MALib*'s implementation result of population-based methods.

Algorithm	**Time(sec)**	**Population Size**
Fictitious Self-Play	7562	60
PSRO(fictitious play)	4862	40
PSRO(α-rank)	**1776**	**21**

Population-based training for Leduc Poker. We compared the PSRO algorithm with OpenSpiel's [**?**] implementation on Leduc Poker, a common benchmark in Poker AI. In order to stress how the evaluation effects the running time, we change the meta-solver to fictitious play, an approximation of Nash Equilibrium with tractable solving time on a large payoff matrix. To get a relatively accurate empirical payoff, the number of simulations was sat to be 2000 for each policy combination and the learning iteration was limited to 100 steps, i.e., the maximum of population size is 100.

We evaluate the convergence of PSRO with exploitability [**?**]. As shown in Figure 8b, we cut 70% execution time while maintaining exploitability with a similar quality as OpenSpiel(Figure 8). Moreover, it indicates that *MALib* is more capable of complicated games since the executing time of OpenSpiel grows much faster than *MALib*. Other methods like Fictitious Self-Play [**?**] (FSP) and Self-Play [] (SP) were implemented and evaluated in the same environment settings. In Table 3, we compare the execution time required and population size expanded when the exploitability goes to 0.5. The results show that PSRO outperforms the other two methods.

Specifically, SP fails to converge since this Poker game is not a purely transitive game, and FSP is more time-consumption than PSRO. It might be that PSRO considers the interactions and meta-game between different policies in populations, and solves it to approximate the Nash Equilibrium of the underlying game, which results in faster convergence rate. Furthermore, when an exact meta-game solver such as the LP-solver or α-rank, PSRO will converge in shorter time and smaller population size.

MADDPG for MPE. Multi-agent Particle Environments [**?**] (MPE) is a typical benchmarking environment for MARL algorithms. It offers plenty of scenarios covering cooperative and competitive tasks. We compared MADDPG with RLlib implementation on seven scenarios under different worker settings, covered cooperative, competitive, and mixed cooperative-competitive tasks.

The experiments were conducted under different worker settings to compare the convergence performance. Figure 9 shows the results on *simple adversary*, it indicates that *MALib*'s implementation performs more steadily than RLlib's, especially when the worker number increases, RLlib implementation show high variance and fail to converge. We point out that the difference in implementations between *MALib* and RLlib is that the former executes learning in a fully asynchronous and the RLlib's implementation executes learning in sequential. Despite the data sampling executes in parallel, RLlib requires extra efforts on tuning to solve the data starvation in the training stage.

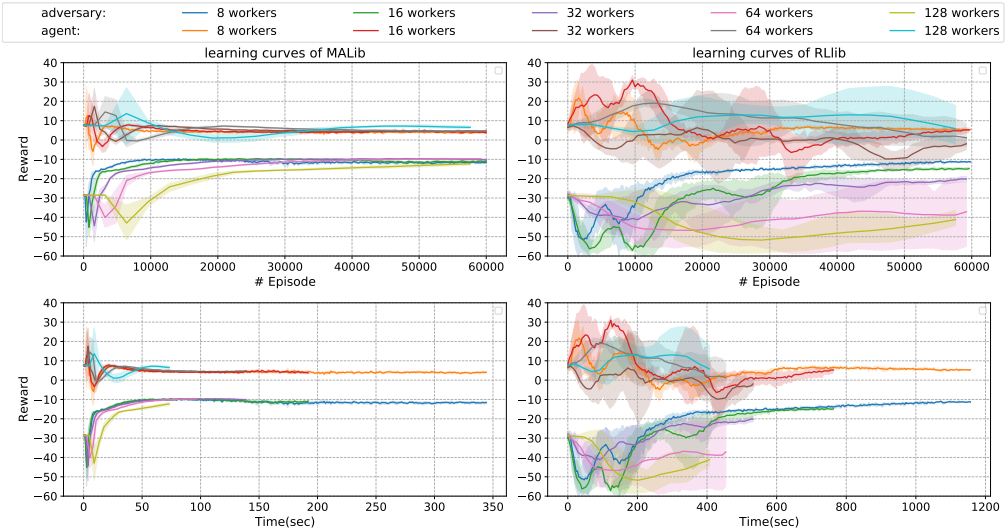


Figure 9: Comparisons of MADDPG in *simple adversary* under different rollout worker settings. Figures in the top row depict each agent's episode reward w.r.t. the number of sampled episodes, which indicate that *MALib* converges faster than RLlib with equal sampled episodes. Figures in the bottom row show the average time and average episode reward at the same number of sampled episodes, which indicates that *MALib* achieves 5× speedup than RLlib.

5 Conclusion

In this paper, we introduced *MALib* for PB-MARL to boost the research from high efficient training and code implementation, also compared with novel distributed RL frameworks in system and algorithm performance. Despite that *MALib* is currently built upon some of the low-level stacks from Ray [**?**] and lacks further optimization for GPU, it achieved higher throughput than previous work on multi-agent Atari tasks, especially in the CPU-only setting. For the algorithm performance, *MALib* shows at least 3x speedup with limited simulation parallelism on PB-MARL tasks compared to the existing library and achieved state-of-the-art scores on MPE tasks 5x faster than the parallel implementation in RLlib. System development is a long-term work, we plan to

further improve MALib by optimizing the usage of the heterogeneous computing resources and testing the performance on much larger scale clusters in the future.

Acknowledgments

The SJTU team is supported by "New Generation of AI 2030" Major Project (2018AAA0100900), Shanghai Municipal Science and Technology Major Project (2021SHZDZX0102), National Natural Science Foundation of China (62076161, 61632017) and Shanghai Sailing Program (21YF1421900). We also thank Zhicheng Zhang, Hangyu Wang, Ruiwen Zhou, Weizhe Chen, Minghuan Liu, Yunfeng Lin, Xihuai Wang, Derrick Goh and Linghui Meng for many helpful discussions, suggestions and comments on the project, and Ms. Yi Qu for her help on the design work.

A Algorithm Library

We have integrated a set of popular (MA)RL algorithms. Table 4 gives an overview of these algorithms and tags them according to 1) training interface introduced in Section 3.3, 2) execution mode, and 3) the supported PB-MARL algorithms. The training interfaces could be **Independent** or **Centralized** which are corresponding to independent learning and centralized learning respectively. The execution mode could be Async (asynchronous) or Sync (synchronous). In the initial implementation, we provided three PB-MARL algorithms support, they are Policy Space Response Oracle [**?**] (PSRO), Fictitious Self-play [**?**] (FSP), Self-play [**?**] (SP) and Population-based Training [**?**] (PBT).

Table 4: Implemented algorithms in *MALib*.

Algorithm	Training Interface	Execution Mode	PB-MARL Support
DQN [**?**]	Independent	Async/Sync	PSRO/FSP/SP
Gorilla [**?**]	Independent	Async	PSRO/FSP/SP
A2C [**?**]	Independent	Sync	PSRO/FSP/SP
A3C [**?**]	Independent	Async	PSRO/FSP/SP
SAC [**?**]	Independent	Async/Sync	PSRO/FSP/SP
DDPG [**?**]	Independent	Async/Sync	PSRO/FSP/SP
PPO [**?**]	Independent	Sync	PSRO/FSP/SP
APPO	Independent	Async	PSRO/FSP/SP
MADDPG [**?**]	Centralized	Async/Sync	PBT
QMIX [**?**]	Centralized	Async/Sync	PBT
MAAC [**?**]	Centralized	Async/Sync	PBT

B Additional Results

B.1 MADDPG for MPE

We implemented MADDPG in *MALib* with the same configuration as RLlib, i.e., both of the actor and critic uses three layers of 64-units fully-connect network. The experiments were conducted in seven scenarios introduced in PettingZoo [**?**], with different worker settings (ranges from 1 to 128), as listed below.

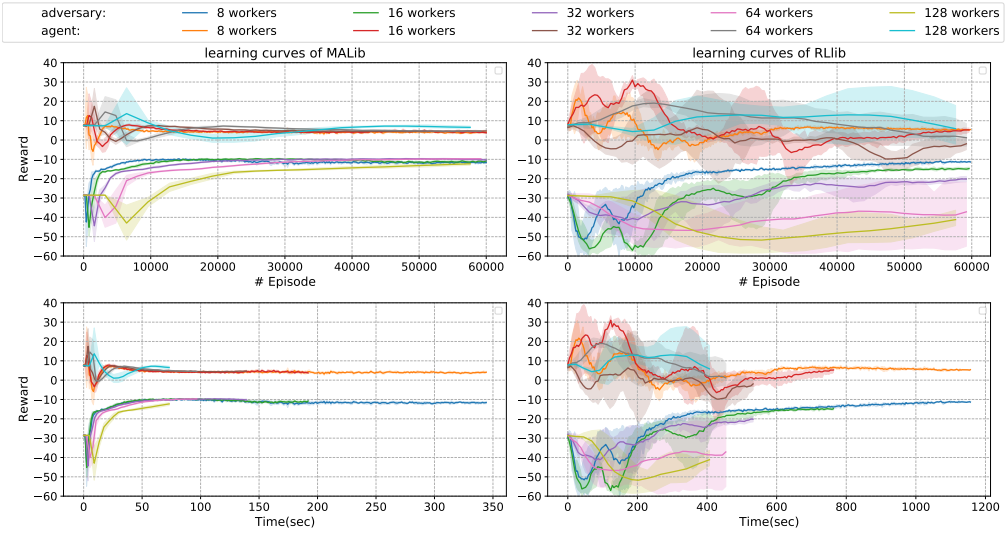


Figure 10: Comparisons of MADDPG in *simple adversary* under different rollout worker settings.

Simple Adversary. There are 1 adversary, 2 good agents and 2 landmarks in this scenario. All agents can observe landmarks and other agents. One landmark is tagged as the 'target landmark'. Good agents are rewarded based on the distance to the target landmark, i.e., closer to the target landmark higher reward, but also receive negative reward based on how close the adversary is to the target landmark. For the adversary, it is rewarded based on distance to the target, but it doesn't know which landmark is the target landmark. In this scenario, good agents have to learn to 'split up' and cover all landmarks to deceive the adversary. Figure 10 shows the comparison from the converged reward and time-consumption.

Simple Crypto. There are 2 good agents (Alice and Bob) and 1 adversary (Eve) in this scenario. Alice must sent a private 1 bit message to Bob over a public channel. Alice and Bob are rewarded +2 if Bob reconstructs the message, but are rewarded -2 if Eve reconstruct the message (that adds to 0 if both teams reconstruct the bit). Eve is rewarded -2 based if it cannot reconstruct the signal, zero if it can. Alice and Bob have a private key (randomly generated at beginning of each episode) which they must learn to use to encrypt the message. Figure 11 shows the comparison from the converged reward and time-consumption.

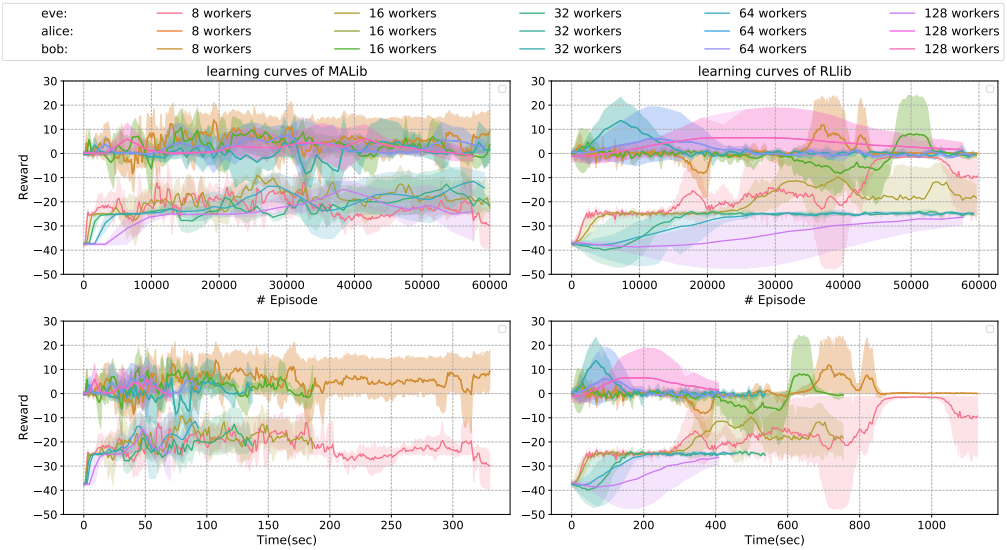


Figure 11: Comparisons between learning curves of MALib and RLlib when running MADDPG in *simple crypto* environment under different rollout worker settings.

Simple Push. There are 1 good agent, 1 adversary, and 1 landmark in this scenario. The good agent is rewarded based on the distance to the landmark. The adversary is rewarded if it is close to the landmark, and if the agent is far from the landmark (the difference of the distances). Thus the adversary must learn to push the good agent away from the landmark. Figure 12 shows the comparison from the converged reward and time-consumption.

Simple Reference. There are 2 agents and 3 landmarks of different colors in this scenario. Each agent wants to get closer to their target landmark, which is known only by the other agents. Both agents are simultaneous speakers and listeners. Locally, the agents are rewarded by their distance to their target landmark. Globally, all agents are rewarded by the average distance of all the agents to their respective landmarks. The relative weight of these rewards is controlled by the local_ratio parameter. Figure 13 shows the comparison from the converged reward and time-consumption.

Simple Speaker Listener. This scenario is similar to simple_reference, except that one agent is the 'speaker' and can speak but cannot move, while the other agent is the listener (cannot speak,

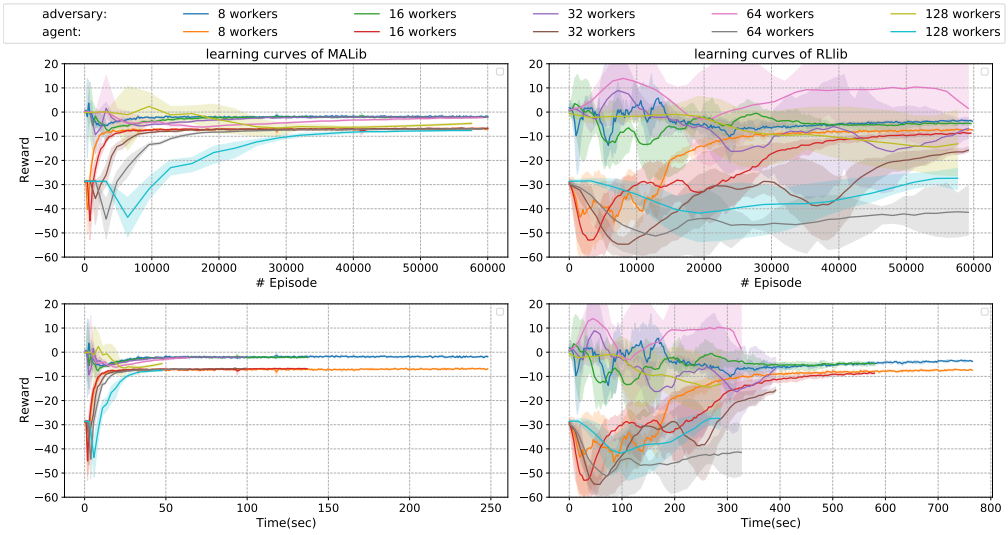


Figure 12: Comparisons between learning curves of *MALib* and RLlib when running MADDPG in *simple push* under different rollout worker settings.

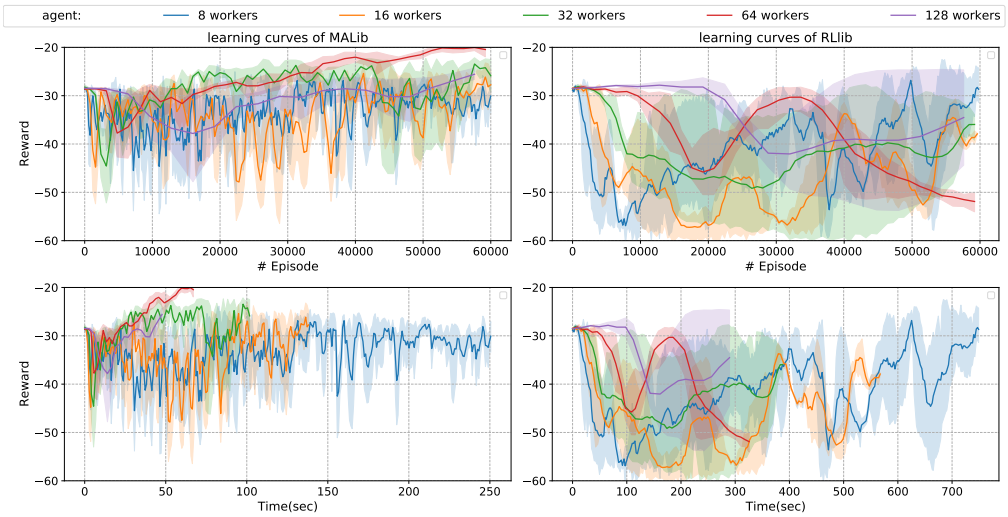


Figure 13: Comparisons between learning curves of MALib and RLlib when running MADDPG in *simple reference* under different rollout worker settings.

but must navigate to correct landmark). Figure 14 shows the comparison from the converged reward and time-consumption.

Simple Spread. There are 3 agents, 3 landmarks in this scenario. Agents must learn to cover all the landmarks while avoiding collisions. More specifically, all agents are globally rewarded based on how far the closest agent is to each landmark (sum of the minimum distances). Locally, the agents are penalized if they collide with other agents (-1 for each collision). The relative weights of these rewards can be controlled with the local_ratio parameter. Figure 15 shows the comparison from the converged reward and time-consumption.

Simple Tag. There are 1 good agent, 3 adversaries and 2 obstacles in this scenario. Good agent is faster and receive a negative reward for being hit by adversaries (-10 for each collision). Adversaries are slower and are rewarded for hitting good agents (+10 for each collision). Obstacles block the way. Figure 16 shows the comparison from the converged reward and time-consumption.

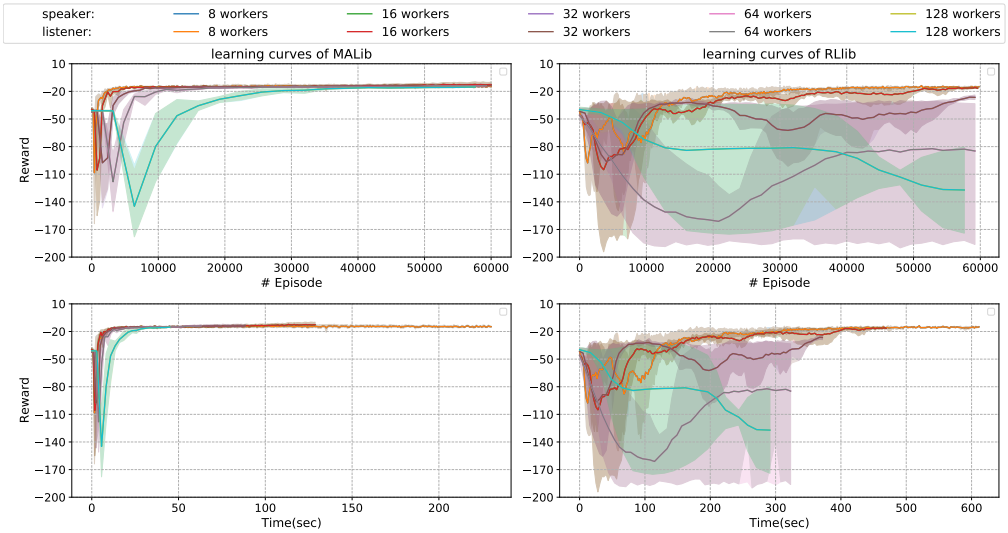


Figure 14: Comparisons between learning curves of MALib and RLlib when running MADDPG in *simple speaker listener* under different rollout worker settings.

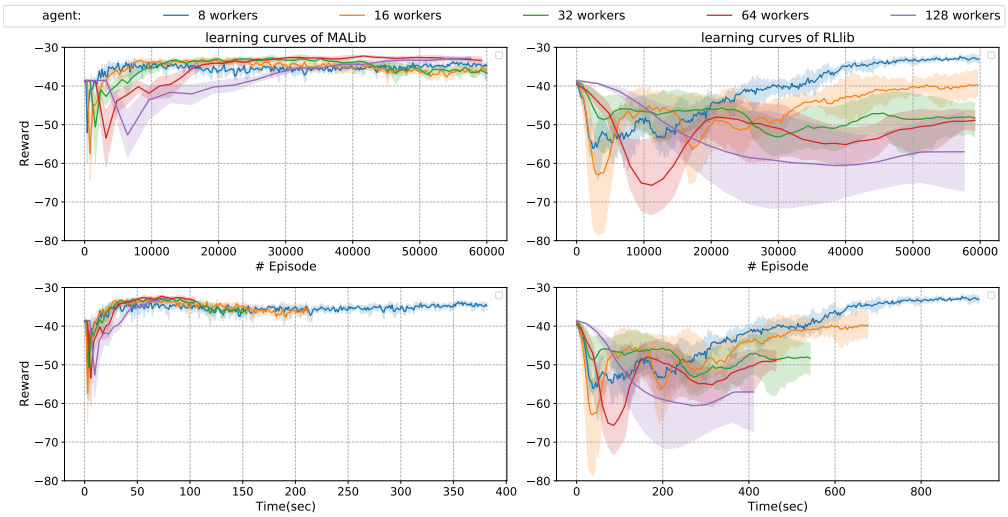


Figure 15: Comparisons between learning curves of MALib and RLlib when running MADDPG in *simple spread* under different rollout worker settings.

In summary, *MALib*'s implementation performs more steadily than RLlib in different worker settings, and achieved faster convergence rate.

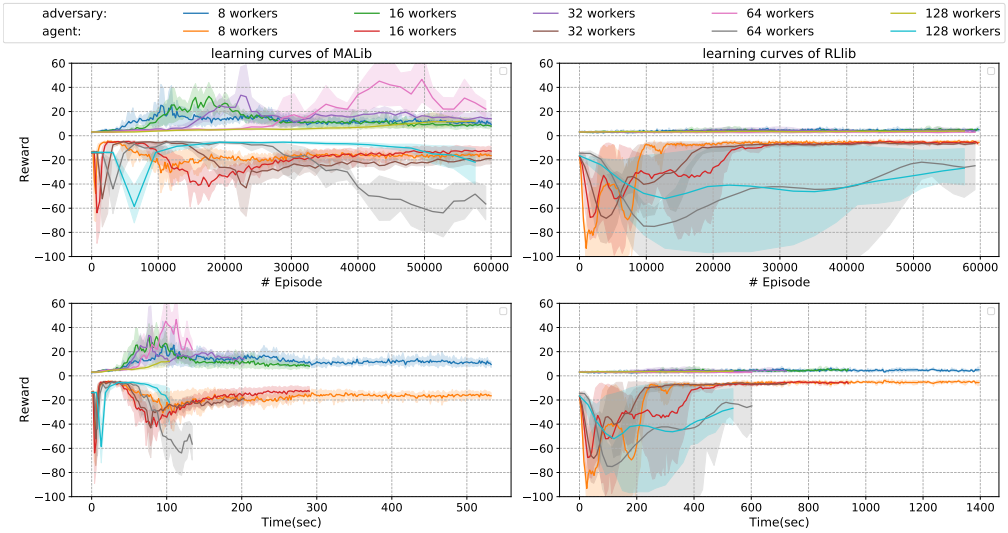


Figure 16: Comparisons between learning curves of MALib and RLlib when running MADDPG in *simple tag* under different rollout worker settings.