SADMA: Scalable Asynchronous Distributed Multi-Agent Reinforcement Learning Training Framework

Sizhe Wang*, Long Qian*, Cairun Yi, Fan Wu, Qian Kou, Mingyang Li, Xingyu Chen, and Xuguang Lan†

Xi'an Jiaotong University, Shaanxi 710049, China
{wangsizhe,qianlongym,yicairun,wf_fixer, xjtukouqian,limingyang,}@stu.xjtu.edu.cn
{chenxingyu_1990,xglan}@xjtu.edu.cn

Abstract. Multi-agent Reinforcement Learning (MARL) has shown significant success in solving large-scale complex decision-making problems while facing the challenge of increasing computational cost and training time. MARL algorithms often require sufficient environment exploration to achieve good performance, especially for complex environments, where the interaction frequency and synchronous training scheme can severely limit the overall speed. Most existing RL training frameworks, which utilize distributed training for acceleration, focus on simple single-agent settings and are not scalable to extend to large-scale MARL scenarios. To address this problem, we introduce a **S**calable **A**synchronous **D**istributed **M**ulti-**A**gent RL training framework called **SADMA**, which modularizes the training process and executes the modules in an asynchronous and distributed manner for efficient training. Our framework is powerfully scalable and provides an efficient solution for distributed training of multi-agent reinforcement learning in large-scale complex environments. Code is available at https://github.com/sadmaenv/sadma.

Keywords: Multi-agent Reinforcement Learning · Distributed Training · Large Scale Multi-Agent Training.

1 Introduction

Multi-Agent Reinforcement Learning (MARL) has achieved remarkable success in many real-world decision-making problems that involve multi-agent systems across various domains, such as multi-player strategy games[3,17], network routing[20], and autonomous driving[4]. However, due to the involvement of interactions and cooperation among multiple agents, MARL environments tend to be extremely complex. Existing algorithms often require a lot of interactions

* Equal contributions
† Corresponding author

Table 1. Comparison of SADMA with other multi-agent reinforcement learning libraries.

Library	Parallel Env	Distributed Support	Distributed Backend	Flexible Resource Allocation	Asynchronous Training
PyMARL	✓	×	-	-	×
MARLlib	✓	✓	Ray	×	×
SADMA	✓	✓	ZeroMQ	✓	✓

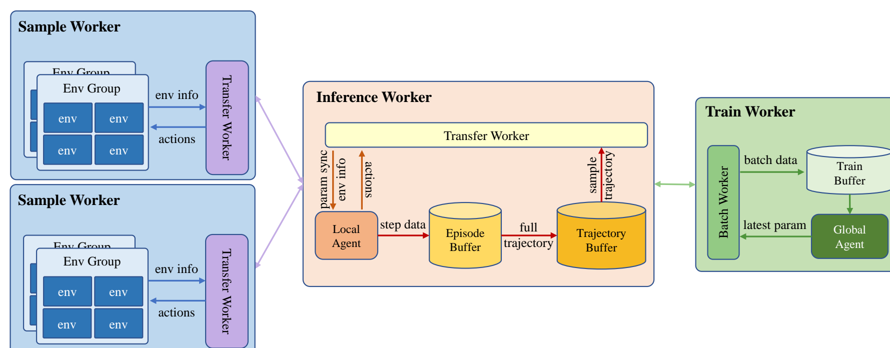


Fig. 1. Overall framework of SADMA. Our framework consists of five modules, each of which runs on a separate process and can be deployed anywhere in the cluster in a distributed manner with a unified data transfer interface. The sample worker is responsible for managing the execution of environmental entities, the inference worker generates actions based on the environment information, and the train worker updates network parameters using sampled data and sends the latest parameters to the inference worker.

with the environment to learn effective strategies[22]. However, as the number of agents increases, the speed of interaction with environments may severely decrease due to the escalating complexity of inference within environments. This results in the need for extensive training time, particularly in intricate environments.

In order to reduce training time, some researchers have suggested running multiple instances of the environment in parallel using multiprocessing techniques to improve the sample efficiency. Existing multi-agent reinforcement learning libraries such as PyMARL [16] and PyMARL2 [8] utilize python's multiprocess programming technique to realize environment parallelism. However, when confronted with large-scale complex multi-agent environments, environment instances may run very slowly and consume a lot of resources. Limited by the computational resources of a single machine, simple environment parallelism still cannot fulfill the demand. Although there are some relatively accessible open-source distributed RL algorithm libraries, such as RLlib [11], which utilize Ray [14] as the distributed framework, their focus is on single-agent RL algo-

rithms, with limited support for MARL algorithms. Recently, MARLlib [9] was introduced as an extension of RLlib to provide support for multi-agent reinforcement learning algorithms. However, MARLlib mainly focuses on the encapsulation and integration of algorithms and environments, and does not provide a diverse and flexible distributed training scheme for multi-agent reinforcement learning algorithms. In addition, MARLlib only implements a distributed sampling process, which is not able to utilize the distributed GPU resources to make full use of cluster computing resources to accelerate training. To the best of our knowledge, there is no open source unified framework for distributed training of multi-agent reinforcement learning algorithms that enables flexible and scalable deployment for different cluster resource configurations.

In the context of large-scale MARL training, effectively harnessing cluster computing resources to expedite training becomes a critical concern. However, existing MARL algorithm libraries suffer from various problems and limitations when facing large-scale distributed training demands. To address these challenges, we propose **S**calable **A**synchronous **D**istributed **M**ulti-**A**gent RL training framework called **SADMA**. Our framework hopes to address the problem of efficient acceleration of distributed training in large-scale multi-agent reinforcement learning training scenarios. Our framework leverages multiprocessing techniques and the lightweight asynchronous messaging library ZeroMQ (ZMQ) [7] to construct a flexible distributed training framework. We specifically analyze the training time-consuming problem in multi-agent reinforcement learning, and use efficient distributed parallel sampling and asynchronous training to further reduce the waiting time during the training process. To enable flexible distributed deployment and asynchronous training, we decouple and modularize the MARL training process. In addition, we design efficient and unified data transfer interfaces for cross-process and cross-machine communication, bringing flexibility and scalability to distributed deployments. Our framework adapts to different cluster configurations, allowing flexible deployment of modules in clusters to fully utilize computing resources. Furthermore, our framework is easy to use and deploy, and with the help of deployment systems such as Kubernetes (K8s) [12], it is possible to efficiently run distributed training tasks in large-scale clusters. Our framework offers the following advantages:

- **Scalability**. Our framework is designed to be deployed in large-scale clusters, allowing efficient utilization of computational resources thus achieving significant training acceleration.
- **Modularization**. We modularize the MARL training process to simplify algorithms construction and provide support for distributed training.
- **Asynchronicity**. We decouple the modules to run the sampling and training process asynchronously, reducing the waiting time to achieve higher resource utilization and runtime speeds.
- **Flexibility**. Modules can be combined and flexibly deployed anywhere in the cluster to adapt to various cluster resource configurations.

2 Related work

The success of Deep Reinforcement Learning(RL) is inseparable from huge data and computing power, which leads to huge demand for distributed learning that can speed up overall training and improve computational resource utilization.

Due to the structured computation pattern of RL algorithms, some successful RL methods are proposed for improving sample and training efficiency. Early algorithms improve sampling efficiency by interacting with multiple environments simultaneously, such as Advantage Actor-Critic (A2C) [10], which aggregates sample data and then performs SGD [15] iterations, using the updated strategy to continue collecting new samples. Asynchronous Advantage Actor-Critic (A3C) [13] uses multiple independent actors, each holding a policy copy, to perform environment simulation sampling, action inference, and gradient computation, respectively. GA3C [2], which is a hybrid CPU/GPU version of the A3C, introduces a separate learner component that uses the GPU for action generation and learning in an asynchronous implementation. These algorithms make previous non-distributed DRL methods distributed using one machine.

Based on these efficient algorithms, some frameworks extend to distributed training on multiple machines. Among them, IMPALA (Importance Weighted Actor-Learner Architecture) [6] uses a GA3C-like architecture where parallel actors communicate with environments, collect trajectories, and send them to the learners for parameter updating. Since gradient calculation is put on the learners' side, which can be accelerated with GPUs, the framework is claimed to scale to thousands of machines without sacrificing data efficiency. Based on IMPALA, SEEDRL (Scalable, Efficient, Deep-RL) [5] achieves further performance improvements through a centralized inference architecture and an optimized communication layer. The communication between learners and actors is mere states and actions reducing latency.

On the basis of these algorithms, a series of open-source distributed RL libraries and frameworks have been produced, some of which provide support for multi-agent reinforcement learning. RLlib [11] integrates a large number of RL algorithms into a high-performance, scalable distributed algorithm framework based on Ray [14]. However, its framework lacks the flexibility to control the details of distributed training to achieve targeted performance optimization, and code packaging is complex. MAlib [23] also develops a framework for distributed MARL algorithms based on population training with Ray. MARLlib [9] is a new distributed MARL library that combines the core advantages of Ray and RLlib, but does not provide a flexible distributed deployment scheme to effectively utilize distributed computing resources. There is still no flexible and scalable distributed training framework for multi-agent reinforcement learning algorithms. Our training framework is different from existing algorithm libraries in that we modularize the training process and run the modules asynchronously to achieve higher training efficiency; we use ZMQ, a high-performance asynchronous messaging library, instead of ray to build a flexible and scalable distributed deployment scheme.

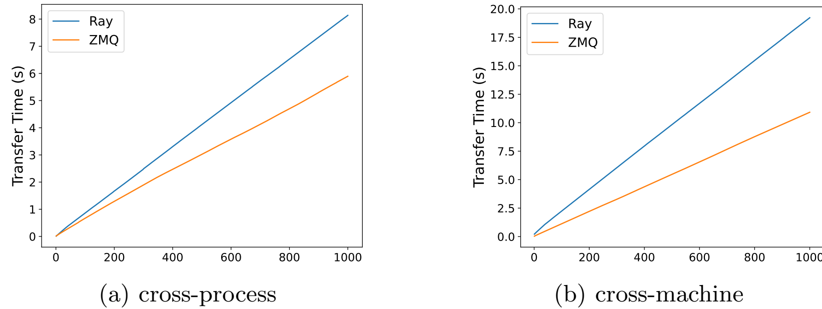
3 Framework Design

In this section, we specifically describe the overall design and characteristics of SADMA. The overall framework is shown in Fig. 1. Based on multi-process techniques and the asynchronous messaging library ZMQ, we build a unified data transfer interface to facilitate efficient communication between processes to achieve compatibility with cross-process and cross-machine data transfer. We modularize the multi-agent reinforcement learning training process and employ asynchronous execution to reduce the waiting time in the original synchronous training process as much as possible. We perform a number of specific performance optimizations for sampling and training to further increase runtime speed and improve resource utilization. Our framework has great scalability and flexibility to adapt to different cluster resource configurations, and is able to support large-scale MARL training and effectively utilize cluster resources to reduce training time.

3.1 High-Performance Data Transfer Scheme

Cross-Process Data Transfer In parallel computing, multiprocess programming has become a common way to fully utilize multi-core processors and improve program performance. However, for multi-agent reinforcement learning training tasks, frequent data transfers between processes are usually necessary, which requires efficient data transfer mechanisms to reduce the waiting time. We use cross-process data transfer scheme for modules running on the same machine. To handle CPU-side data, we build pipes for inter-process data transfer. To avoid unnecessary data copying, we utilize shared memory for frequently read and written data, such as replay buffers. As for GPU-side data, which are shared across processes, we exclusively transfer pointers to circumvent resource-intensive data transfers.

Cross-Machine Data Transfer To achieve high-performance distributed training, improving the efficiency of cross-machine data transfer is paramount. Existing distributed reinforcement learning training frameworks typically employ Ray, an open-source unified compute framework, as their communication scheme. However, despite its user-friendliness, Ray is not lightweight and efficient enough. Since large-scale multi-agent environments may involve hundreds of agents with large observation dimensions, there is a high demand for distributed training data transfer. In pursuit of efficiency, we employ ZMQ, a lightweight and high-performance messaging library, to facilitate more expedited cross-machine data transfer. With the advantages of flexibility, reliability, high-performance and lightweight, ZMQ is ideally suited to handle the frequent data transfer requirements in large-scale multi-agent reinforcement learning training. Recently some researchers have also evaluated the speed benefits of ZMQ for communication [1]. We conduct a comparative analysis of the cross-machine transfer efficiency between ZMQ and Ray for various data sizes. Fig. 2 shows that ZMQ is more



(a) cross-process (b) cross-machine

Fig. 2. Cross-process and cross-machine data transfer speed comparison between ZMQ and Ray. We tested the time consumption at different transfer data sizes. We created arrays of shape (N, N, N) and varied N from 1 to 1000, with the horizontal axis representing the size of N.

efficient than Ray for both cross-process and cross-machine communication, especially for cross-machine transfers. This supports our choice of ZMQ as the efficient data transfer scheme.

Unified Transfer Interface Design In order to flexibly adapt to single-machine and cross-machine data transfer, we unified the data transfer interface. We wrap these two data transfer schemes into a unified interface at the code level, which decides which scheme should be used for the current transfer based on the configuration. This unified design can bring many benefits. First, it simplifies programming at the code level so that users do not need to care about the tedious communication approach, and only need to set it in the configuration file. Second, it enhances the flexibility of our distributed training framework, enabling modules to be easily deployed across the distributed cluster without the need for custom communication implementations for each module.

3.2 Modular Design

In order to support distributed training, we modularize the training process so as to fully utilize the computational resources to run each module asynchronously to reduce the waiting time and improve the overall training speed. Our overall modularized design is shown in Fig. 1. The detailed functions of each module are described below.

Transfer Worker. In order to facilitate inter-module data transfer, we develop the transfer worker based on the unified transfer interface. The transfer Worker is responsible for cross-machine or cross-process data transfer between modules. It operates within a separate process and employs multiple sub-threads to concurrently handle inter-module data transfer operations, eliminating data

waiting times that could otherwise impact the main program's execution. Benefiting from the unified transfer interface design, transfer worker can flexibly adapt to both single-machine and cross-machine inter-process communication scenarios. In combination with the transfer Worker, other modules enable high-performance and flexible data transfers that can be deployed in clusters of various resource configurations for efficient and convenient distributed training.

Sample worker. Large-scale MARL tasks typically require massive amounts of interaction data for training. However, due to limitations in the inference speed of the environments, running a single environment for data sampling cannot efficiently provide the required training data in a timely manner. The prevailing approach is to parallelize multiple environments using multiprocessing techniques to accelerate the data sampling process. Hence, we construct the sample worker which is responsible for managing the interactions of multiple parallel environments.

Each sample worker contains a designated number of parallel environments that utilize multiprocessing techniques to fully leverage computing resources. In order to achieve better scalability and resource utilization, we separate and asynchronously execute environment interaction and policy inference. The sample worker refrains from conducting policy inference and instead focuses solely on managing the execution of environment instances. When interacting with the environment, the sample worker transfers all the environmental information to the inference worker via the transfer worker, and then the inference worker performs policy inference based on the transferred data and returns the corresponding actions to the sampler worker. After receiving the actions, the sample worker executes actions for corresponding environments and collects the information returned from the environment and transfers them to the inference worker again. All parallel environments interact synchronously.

Separating the environment execution from the policy inference can bring benefits to distributed training. Since policy inference involves neural network computation, which generally requires GPU, while environment execution only requires CPU, we can flexibly allocate computing resources to the inference worker and sampler worker for different cluster configurations to fully utilize the cluster resources.

Inference Worker. The inference worker is responsible for policy inference to provide actions for interacting with the environment. The inference worker creates the episode buffer for each environment it is responsible for, since complete episodes are often used as training data in MARL algorithms such as QMIX. The environment information transferred from the sample worker and the policy inference data are saved in the corresponding episode buffer after each interaction step. When an episode is finished, the full trajectory will be stored in the trajectory buffer in memory. In order to save unnecessary memory consumption, the number of episodes stored in the trajectory buffer can be less than the num-

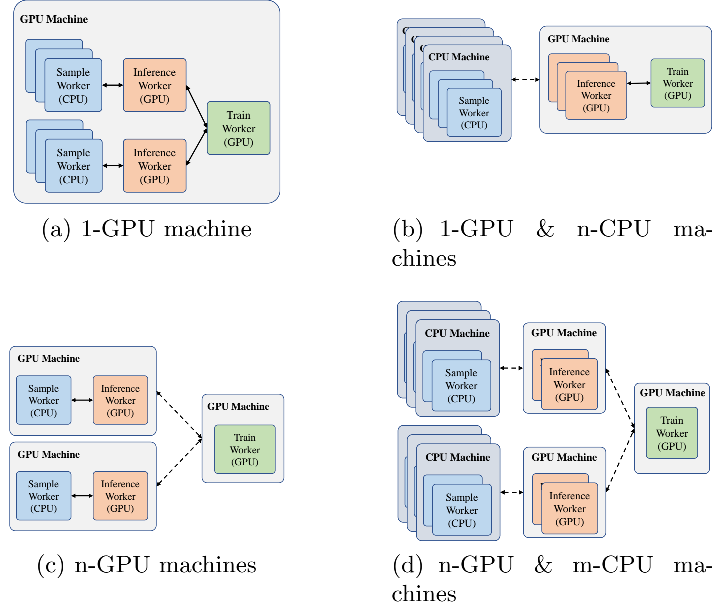
ber of environments. The data in the trajectory buffer is ready and just waiting to be used for training.

In the case of on-policy algorithms such as IPPO [18], when the trajectory buffer reaches its capacity, it will no longer accept new trajectories from the episode buffer. If there is no free space in the trajectory buffer at the end of an episode in the episode buffer, the inference worker will be temporarily stopped until the data in the trajectory buffer are consumed. This limits the generation of outdated data, which can impact the performance of the on-policy algorithm. Although our framework runs asynchronously, there are still synchronization constraints for the On-policy algorithm, which allows us to approach the training bottleneck speed with the guaranteed quality of the sampled data.

Batch worker. In the context of multi-agent reinforcement learning algorithms, the typical requirement is to train on a batch of episodes at once. To support large-scale distributed sampling, we allow the setup of multiple distributed inference workers. Once these episodes are collected, they must be organized into a batch and prepared for training on the GPU. Consequently, there is a need to transfer the data collected in the trajectory buffer to the train worker. To address this data management challenge, we introduce the batch worker that collaborates with the train worker.

The batch worker's role is to consolidate episode data from each inference worker into a batch and then transfer this batched data onto the GPU, placing it into the train buffer. The train worker, in turn, retrieves the training data directly from the GPU using data addresses. The batch worker runs asynchronously on a separate process without affecting the train worker. It operates as a data processing module, preparing data required for the upcoming training in advance while the train worker focuses on policy updates. This approach eliminates the need for the train worker to wait for data to be processed, resulting in significant time savings.

Train Worker The role of the train worker primarily centers around agent training and synchronization. With the presence of a batch worker, the train worker is relieved from the burden of tedious data processing. This streamlined workflow enables the train worker to devote undivided attention to agent training, resulting in improved efficiency. During each training step, the train worker acquires the pre-processed batch data from the train buffer and proceeds with the parameter update process. Subsequently, it dispatches the most recent parameters to each inference worker, ensuring parameter synchronization across the system. Both the train worker and the train buffer are located on the GPU, thus facilitating fast data communication. By design, the train worker is devoid of superfluous functions, guaranteeing its efficient execution of training tasks. This optimization enhances the performance and responsiveness of the train worker, ultimately accelerating the overall training process.



(a) 1-GPU machine

(b) 1-GPU & n-CPU machines

(c) n-GPU machines

(d) n-GPU & m-CPU machines

Fig. 3. Flexible distributed deployment in different cluster resource configurations. Solid arrows represent cross-process communication and dashed lines represent cross-machine communication

3.3 Flexible Resource Allocation

Existing libraries usually do not focus on performance issues in large-scale distributed training scenarios, but more on algorithm integration and packaging. The most important difference between our framerwork and theirs is that we design our framework for deployment and performance issues in distributed training scenarios. This gives our framework a greater advantage for training multi-agent reinforcement learning algorithms in large-scale complex scenarios.

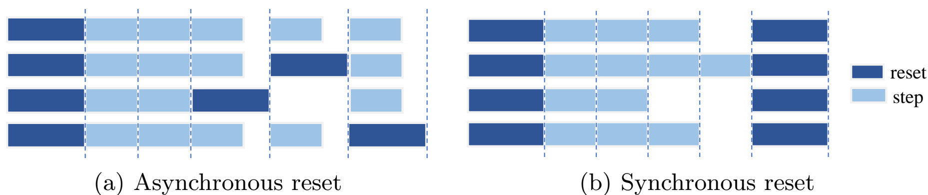
Benefiting from the modularized design and unified data transfer interface, each module can be flexibly combined with each other and assigned to different computing nodes in the cluster regardless of the hardware device restrictions. This facilitates deployment on clusters with different resource configurations. Our framework naturally adapts to different resource configurations and thus can fully utilize cluster resources to accelerate training. In the following we describe in detail several different configuration types supported by our framework, as shown in Fig. 3.

1-GPU Machine. Although our framework is designed for distributed training, considering that a single workstation may be sufficient for small-scale algorithm development, we still take into account the adaptation and runtime per-

formance on a single machine, as shown in Fig. 3(a). In the single-computer case, the data between modules does not need to be transferred based on the network, but only needs to be considered for cross-process transfer. We use inter-process pipelines and shared memory to realize the communication, which ensures the efficiency of training.

1-GPU & n-CPU Machines. When there are limited GPU computing nodes, for example, there is only one GPU node and the rest are CPU nodes, as shown in Fig. 3(b). In this case, there are two general approaches. One is to run only multiple parallel environments on the CPU node, because the environments only need CPU resources to run. Then both the training module and the inference module are placed on the GPU node as its involved in the computation of the neural network. Another approach is to run the parallel environments and the corresponding inference modules on the CPU nodes and the GPU nodes perform only the training. Each of these two approaches has its own advantages. For the first scheme, since only the environment instance is running on the CPU node, there is no need to design the cross-node transfer of model parameters, which is time-consuming when the model parameters are large. However, the CPU node needs to wait for the GPU node to send back the inference results at every environment interaction, hence there is a delay. For the second scheme, since both the inference module and the environment run on the CPU node, there is no environment interaction delay as in the first scheme, but the network inference using the CPU will be slower, and it needs to communicate with the GPU node periodically to synchronize the model parameters. Existing libraries usually focus on algorithm encapsulation and integration, and do not provide optional distributed training schemes. Our framework, on the other hand, can flexibly adapt to a variety of different configurations and only requires modification of the configuration file without additional code changes.

N-GPU Machines. When all the computing nodes in the cluster have GPUs, our framework can more fully utilize GPU resources to accelerate training unlike MARLlib which is unable to utilize distributed GPU resources. Since the modules can be freely combined, we can run the parallel environment and the corresponding inference modules on multiple GPU nodes, as shown in Fig. 3(c). Moreover, in order to accelerate inference and improve GPU utilization efficiency, we can run multiple parallel environments with multiple inference modules on a single GPU node, and try to place each inference module on a different GPU card so as to make the load of GPU cards as balanced as possible. The training modules can also be deployed on any node with spare resources. This flexible configuration scheme can fully utilize the resources of each computing node, and thus can show advantages when facing distributed training of large-scale complex tasks. Existing libraries, such as MARLlib, do not have the ability to utilize GPU resources across nodes, which limits their use in large-scale distributed training.



(a) Asynchronous reset

(b) Synchronous reset

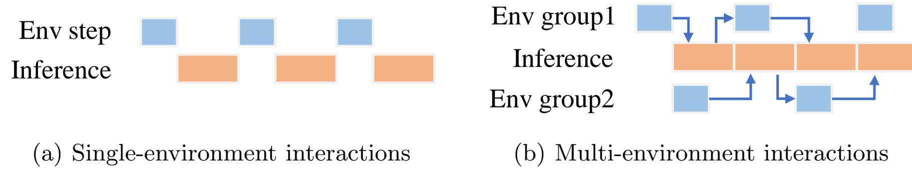
Fig. 4. Synchronous and asynchronous environment reset.

N-GPU & M-CPU Machines. When having multiple CPU nodes and GPU nodes, as shown in Fig. 3(d), our framework can flexibly deploy different computational tasks for different nodes' computational resources. We can run parallel environment instances on CPU nodes and then run inference modules on GPU nodes. And in order to improve resource utilization, we can group multiple CPU nodes, assign one GPU node to be responsible for inference, and run multiple inference module instances on this node for load balancing. For the training module, it can still be assigned to any GPU node with remaining resources. Moreover, when there are enough GPU nodes, the training module can also be deployed on a free GPU node and utilize multi-card accelerated training to cope with large-scale training data demand.

3.4 High-Performance Specific Optimization

To address the training process as well as our distributed framework, we use a series of targeted performance optimizations to improve the sampling speed and training speed. The specific optimization scheme is as follows.

Synchronous & Asynchronous Environment Reset. Our framework supports both synchronous and asynchronous environment reset modes. The difference between these two modes is shown in Fig. 4. When interacting with environments, we adopt batch inference to process information from multiple environments at the same time. In synchronous reset mode, we wait for the end of the episodes of all environments before resetting them, while in asynchronous reset mode, we reset an environment as soon as its episode ends. Both synchronous and asynchronous resets have their own advantages and disadvantages. When environment reset takes too long, adopting asynchronous reset mode may cause the overall interaction speed to decrease. This is because we need to wait for all environments to return information before performing policy inference, which can result in a long wait if an environment is being reset. In this case, synchronous reset mode may be better because it only needs to wait for the last environment to finish its episodes and spend one reset time to reset all environments, while asynchronous reset needs to wait for reset frequently. In contrast, asynchronous reset is more advantageous when the environment reset only consumes about the same amount of time as the environment execution.



(a) Single-environment interactions (b) Multi-environment interactions

Fig. 5. Illustration of single-environment and multi-environment group interactions. The blue rectangle represents the environment execution process and the orange rectangle represents the policy inference process.

Therefore, when environment reset consumes a lot of time, it is suitable to adopt synchronous reset mode, while when environment reset consumes little time, it is advantageous to adopt asynchronous reset mode. Our framework supports both modes to deal with different application scenarios.

Multi-Environment Group. The problem with synchronized environment interaction is that there is a sequential relationship between environment execution and policy inference. This results in sample workers and inference workers always having idle time and not being able to fully utilize compute resources. In order to achieve more efficient environment interaction, we use multi-environment groups to divide the environments managed in the sample worker into multiple groups, as shown in Fig. 5. When an environment group returns the environment information and waits for the inference worker's inference data, the rest of the environment groups perform the environment execution. When the inference worker finishes the inference, the rest of the environment groups will transfer the environment information immediately, so that the inference worker will not be in an idle state.

Batch Inference. Considering that multi-agent environments tend to have large observation dimensions due to a large number of agents and in order to speed up sampling, multiple sample workers are created in a distributed manner for sampling. However, if only one inference worker is allocated to these sample workers, since it is a synchronized environment interaction, it will result in the need to wait for the slowest environment to return observations. The existing practice of distributed reinforcement learning frameworks is to configure an inference worker for each sample worker to ensure efficient inference. However, since inference worker contains neural networks, too many inference workers can lead to a large amount of GPU resources consumption in the case of large-scale distributed training, which is not favorable for scalability. Therefore, we use the batch inference method to form a batch of multiple sample workers and assign an inference to be responsible for the interaction. This will bring flexibility and efficiency to the construction of the sampling process. This is not as fast as assigning an inference worker to each sample worker, since the time it takes for an

environment to return batch information depends on the slowest environment, and the inference time increases due to the increase in the amount of data, but it allows the user to be more flexible and fully utilize the distributed computing resources. We can control GPU usage by controlling the number of inference workers. This design brings powerful scalability to distributed sampling.

4 Experiments

In this section, we evaluate the performance of the proposed SADMA framework. We choose to compare it with PyMARL2, a classic single-computer multi-agent reinforcement learning algorithm library, and MARLlib, the latest MARL algorithm library with distributed support, as baselines. In order to fully demonstrate the advantages of our framework, we test the efficiency under different computational resource scenarios on single-machine and multi-machine settings respectively. Considering that PyMARL2 itself does not provide a distributed deployment program, we only compare it with MARLlib in the multi-computer scenario. We use two different hardware configurations: (1) System#1: 128-core workstation with 8 GPUs; (2) System#2: a four-node cluster with each node owning 64-core and 4 GPUs. All the GPUs mentioned are of the same model (NVIDIA RTX3090). We describe the specific experimental setup and results below.

4.1 Throughput Comparisons

Throughput measures the sampling speed of the framework, which affects the overall training efficiency. Faster sampling speed reduces waiting time by providing the sample data needed for training in time. We compare to baselines under different resource configurations for single and multiple machines settings.

We choose the classical multi-agent reinforcement learning environment SMAC for testing. To increase the complexity of the environment interaction, we choose 27m_vs_30m map in SMAC. Considering that multi-agent reinforcement learning includes on-policy algorithms like IPPO and off-policy algorithms like QMIX, these two categories of algorithms have different sampling processes, which may lead to different sampling speeds. We separately test the throughput under these two types of algorithms. In order to compare the sampling speed under different computational resource conditions, we gradually increase the CPU core utilization from 1 to 256 to test the throughput. To mitigate the impact of server performance fluctuations on the tests, we run the training program continuously for 5 minutes after it is fully initiated to calculate the average sampling speed.

Fig. 6 shows the comparison results. It can be seen that the sampling speed of our framework is excellent on both single and multiple machines. Since PyMARL2 is a serial training process and does not set up different processes for sampling and training, its sampling speed is slower. Benefiting from the asynchronous training process and targeted optimization schemes, our framework

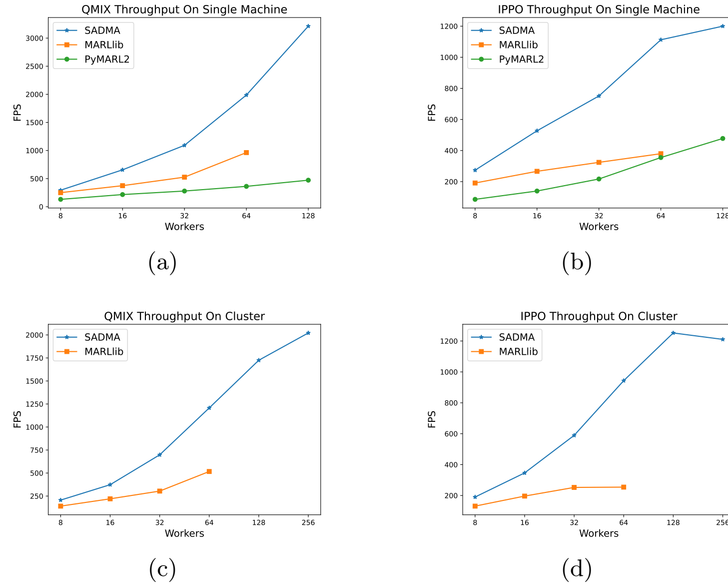


Fig. 6. Throughput comparison results. The horizontal coordinate represents the number of CPU cores used for sampling, with each core running one environment instance. Due to the high GPU memory usage of MARLlib, there is a memory overflow when setting the number of workers to 128, so we only tested to 64 workers.

can achieve higher sampling speeds. However, while MARLlib is capable of distributed parallel sampling, it does not have a targeted distributed optimization for the multi-agent reinforcement learning training process, resulting in poor performance and scalability.

4.2 Convergence Acceleration

Although our framework has a higher sampling speed, it remains a question whether this is effective in improving the wall times necessary for the algorithm to converge. Therefore, we compare the wall times of each framework to make the algorithm converge with the same resource configuration. For comparison with the convergence performance of the algorithm in PyMARL2, we test it on the single-machine configuration. We choose several maps in SMAC to test the convergence speed of each framework with the QMIX and IPPO algorithms. We uniformly set the number of workers to 8 and ensure that other parameters are consistent.

Fig. 7 and Fig. 8 show the convergence of the QMIX and IPPO algorithm in each framework on different maps respectively. It can be seen that our curves are always above PyMARL2 and MARLlib, which indicates that our framework is able to effectively utilize the computational resources to improve the convergence speed of the algorithms under the same resource configuration. This

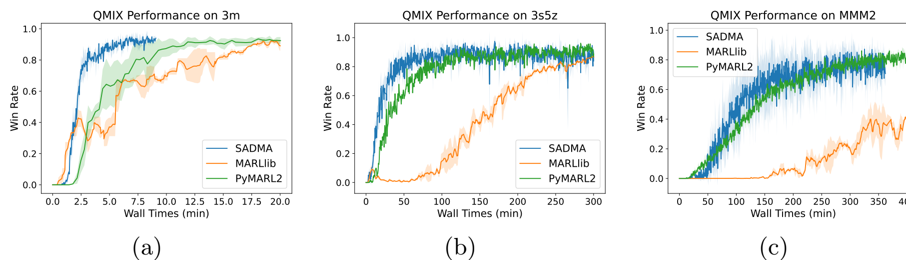


Fig. 7. Comparison of wall times required for convergence of QMIX algorithms with the same resource allocation.

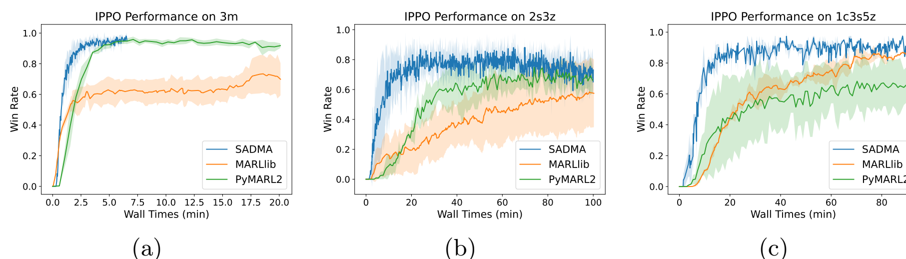


Fig. 8. Comparison of wall times required for convergence of IPPO algorithms with the same resource allocation.

demonstrates the efficiency of our framework. This is achieved by SADMA building an asynchronous training flow through efficient inter-process communication to increase the training speed. For off-policy algorithms such as QMIX, we decouple the environment sampling and training processes to run asynchronously, thus reducing the waiting time due to sampling.

4.3 Scalability Evaluation

In order to evaluate the scalability of SADMA for large-scale multi-agent environments, we constructed an environment containing 1225 agents based on the CityFlow [21] environment, as well as a replenishment environment containing 1000 agents [19]. Due to the large number of agents, running an episode of the environment takes a lot of time and memory. At the same time, single-step policy inference needs to compute the actions of all the agents, which leads to slower inference and larger GPU memory usage. In this case, SADMA can flexibly build a distributed training process similar to IMPALA, deploying the inference worker with the corresponding environment instances on the same compute node to ensure scalability. However, MARLlib cannot deploy the training flow flexibly.

Fig. 9 shows the sampling speedup obtained by allocating different computational resources. For the CityFlow environment containing 1000+ agents, it is

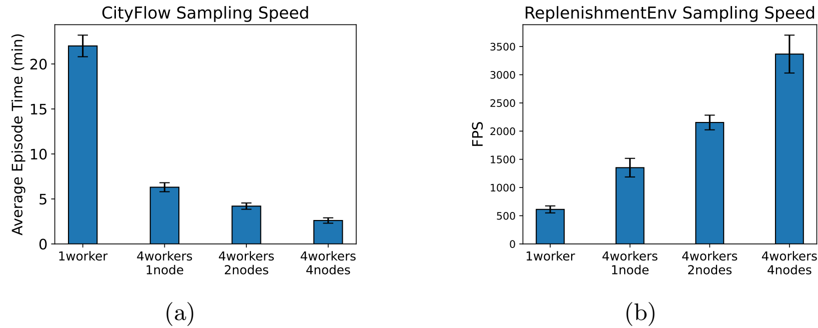


Fig. 9. Distributed sampling speed with different resource configurations. An episode in the CityFlow environment contains 242 steps.

too slow to be able to develop and evaluate algorithms. While our framework can easily implement distributed scaling to accelerate training, opening up the possibility of researching large-scale complex muti-agent environments. For the Replenishment environment, where millions of steps are often required for good performance, the efficiency of sampling and training can be further improved with distributed scaling. The distribution flexibility of SADMA brings powerful scalability and compatibility, which can theoretically scale to hundreds or thousands of computational nodes, providing an efficient solution for research on large-scale complex multi-agent tasks.

5 Conclusion

We propose SADMA, a scalable asynchronous distributed multi-agent reinforcement learning training framework. Our framework achieves asynchronous sampling and training by modularizing the multi-agent reinforcement learning training process. In order to adapt to the needs of large-scale multi-agent body reinforcement learning training, we build a high-performance unified data transfer interface based on ZMQ and multi-process techniques, which makes it easy to deploy each module at any location in the cluster to fully utilize the computational resources. We also optimize the training process to further improve the speed. We compare the training efficiency of existing algorithm libraries under different resource configurations, and the results show that our framework is more efficient and scalable to satisfy the needs of large-scale distributed training. We hope to use our framework to build a generalized distributed multi-agent reinforcement learning algorithm library to accelerate the algorithm research and application. In future work, we will further improve our framework and add more algorithms and environment support.

References

1. Ahmad, M.F.: Public opinion and persuasion of algorithmic fairness: assessment of communication protocol performance for use in simulation-based reinforcement learning training **16**, 1–10 (2023)
2. Babaeizadeh, M., Frosio, I., Tyree, S., Clemons, J., Kautz, J.: Reinforcement learning through asynchronous advantage actor-critic on a gpu. arXiv preprint arXiv:1611.06256 (2016)
3. Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., et al.: Dota 2 with large scale deep reinforcement learning. arXiv preprint arXiv:1912.06680 (2019)
4. Cao, Y., Yu, W., Ren, W., Chen, G.: An overview of recent progress in the study of distributed multi-agent coordination. IEEE Transactions on Industrial informatics **9**(1), 427–438 (2012)
5. Espeholt, L., Marinier, R., Stanczyk, P., Wang, K., Michalski, M.: Seed rl: Scalable and efficient deep-rl with accelerated central inference. arXiv preprint arXiv:1910.06591 (2019)
6. Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., et al.: Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In: International conference on machine learning. pp. 1407–1416. PMLR (2018)
7. Hintjens, P.: ZeroMQ: messaging for many applications. " O'Reilly Media, Inc." (2013)
8. Hu, J., Jiang, S., Harding, S.A., Wu, H., Liao, S.w.: Rethinking the implementation tricks and monotonicity constraint in cooperative multi-agent reinforcement learning. arXiv preprint arXiv:2102.03479 (2021)
9. Hu, S., Zhong, Y., Gao, M., Wang, W., Dong, H., Li, Z., Liang, X., Chang, X., Yang, Y.: Marllib: Extending rllib for multi-agent reinforcement learning. arXiv preprint arXiv:2210.13708 (2022)
10. Konda, V., Tsitsiklis, J.: Actor-critic algorithms. Advances in neural information processing systems **12** (1999)
11. Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J., Jordan, M., Stoica, I.: Rllib: Abstractions for distributed reinforcement learning. In: International conference on machine learning. pp. 3053–3062. PMLR (2018)
12. Luksa, M.: Kubernetes in action. Simon and Schuster (2017)
13. Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., Kavukcuoglu, K.: Asynchronous methods for deep reinforcement learning. In: International conference on machine learning. pp. 1928–1937. PMLR (2016)
14. Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M.I., et al.: Ray: A distributed framework for emerging {AI} applications. In: 13th USENIX symposium on operating systems design and implementation (OSDI 18). pp. 561–577 (2018)
15. Robbins, H., Monro, S.: A stochastic approximation method. The annals of mathematical statistics pp. 400–407 (1951)
16. Samvelyan, M., Rashid, T., De Witt, C.S., Farquhar, G., Nardelli, N., Rudner, T.G., Hung, C.M., Torr, P.H., Foerster, J., Whiteson, S.: The starcraft multi-agent challenge. arXiv preprint arXiv:1902.04043 (2019)
17. Vinyals, O., Babuschkin, I., Chung, J., Mathieu, M., Jaderberg, M., Czarnecki, W.M., Dudzik, A., Huang, A., Georgiev, P., Powell, R., et al.: Alphastar: Mastering the real-time strategy game starcraft ii. DeepMind blog **2**, 20 (2019)

18. de Witt, C.S., Gupta, T., Makoviichuk, D., Makoviychuk, V., Torr, P.H., Sun, M., Whiteson, S.: Is independent learning all you need in the starcraft multi-agent challenge? arXiv preprint arXiv:2011.09533 (2020)
19. Yang, X., Liu, Z., Jiang, W., Zhang, C., Zhao, L., Song, L., Bian, J.: A versatile multi-agent reinforcement learning benchmark for inventory management. arXiv preprint arXiv:2306.07542 (2023)
20. Ye, D., Zhang, M., Yang, Y.: A multi-agent framework for packet routing in wireless sensor networks. sensors **15**(5), 10026–10047 (2015)
21. Zhang, H., Feng, S., Liu, C., Ding, Y., Zhu, Y., Zhou, Z., Zhang, W., Yu, Y., Jin, H., Li, Z.: Cityflow: A multi-agent reinforcement learning environment for large scale city traffic scenario. In: The world wide web conference. pp. 3620–3624 (2019)
22. Zhang, K., Yang, Z., Başar, T.: Multi-agent reinforcement learning: A selective overview of theories and algorithms. Handbook of reinforcement learning and control pp. 321–384 (2021)
23. Zhou, M., Wan, Z., Wang, H., Wen, M., Wu, R., Wen, Y., Yang, Y., Yu, Y., Wang, J., Zhang, W.: Malib: A parallel framework for population-based multi-agent reinforcement learning. J. Mach. Learn. Res. **24**, 150–1 (2023)