IBIMA Publishing
Journal of Software & Systems Development
https://ibimapublishing.com/articles/JSSD/2024/340710/
Vol. 2024 (2024), Article ID 340710, 11 pages, ISSEN: 2166-0824
https://doi.org/10.5171/2024.340710

Research Article

Towards Efficient Memory and Type Safe Modelling of Multi-agent Games: Development of Amfiteatr - Simulation and Reinforcement Learning Tool Written in Rust

Robert JAROSZ

Military University of Technology, Warsaw, Poland
robert.jarosz@wat.edu.pl

Received date:1st april 2024; Accepted date:3 July 2024; published date: 17 September 2024

Academic Editor: Cristian Bucur

Abstract

This paper presents conception and architecture description of memory and type safe library as a tool to simulate multi-agent conflict situations with reinforcement learning support. The presented work discusses the creation of computer models compatible with the conception of games in extensive form. The document presents an approach addressing generic game theory problems with asymmetric players using different information sets and policy types while maintaining type safety. The conception of library is compared to popular PettingZoo library marking community standard for multi agent-reinforcement learning problem. The discussed comparison addresses information flow inside models and performance. The possibility of distributing models across network connected hosts is briefly discussed. At the end of the paper, the current state of the library is discussed and direction for future development is pointed.

Keywords: simulation, reinforcement-learning, Rust, multi-agent,

Introduction

Recent years have brought numerous successful research studies in machine learning algorithms, including reinforcement learning (RL) subarea. A characteristic feature of reinforcement learning is the use of experience gained from repeated simulation, that stands in contrast with supervised learning algorithms, which consume arbitrarily the provided training data. The technique of RL has proven to be very successful in optimizing strategies in various games and problems. That trait makes RL suitable for optimizing game policy and robotic control. Examples of reinforcement learning solutions surpassing human level include mastering classic games reported by (Silver et al., 2016 and Silver et al., 2017) features of self-learning algorithms for Go and chess respectively. The effectiveness of deep reinforcement learning was also shown playing Atari games by Mnih et al., (2013 and 2015). The most popular current libraries providing game environments to research on RL agents are Python libraries *Gymnasium* (Towers *et al.*, 2023) dedicated to modelling single player games and their multiplayer counterpart *PettingZoo* (Terry *et al.*, 2021). This paper presents *Amfiteatr* - an alternative library to build conflict situation models with support for reinforcement learning. The main contribution of a developed library is the usage of safety features and better performance of Rust language to help build effective and correct simulation and reinforcement learning models. Beside distinct implementation language, library features different design of entity communication model.

Choice of Implementation Language

Developed library is aimed to maximize execution performance preserving model safety. For this purpose, the language Rust (Klabnik, Nichols et al., 2024) was selected. Rust is a multi-paradigm, low level, and high control language; it is also a direct competitor of C++ in the system languages niche. It guarantees memory-safety and thread-safety despite not having a garbage collector, what is achieved by novel data ownership model. Differently to C++, the programmer can be sure that no dereferencing of null pointer nor accessing out of range will happen during the program run. Thread safety is an important requirement in simulating multi-agent models, as dedicating separate threads for every agent is a natural approach. Rust supports template implementation using powerful procedural macros and *trait* system allowing generic early-bound function implementation. It also supports creating objects with late bound functions when performance can be traded for runtime flexibility. The combination of safety and performance features has allowed Rust to gain growing share in system development and data science backend. These qualities were decisive in technology selection for the presented library.

Game Theory Model

Library is designed to model conflict problems presentable as game in extensive form (Binmore, 2007) (Ameljańczyk, 1980). N-player game in extensive form can be represented by graph $G = (\mathcal{S}, \mathcal{A}, \Phi)$, where $\mathcal{S}$ is the set of possible game states, $\mathcal{A}$ is a set of possible actions and Φ is function mapping each action (graph edge) $a \in \mathcal{A}$ to an ordered pair of states $(s', s'') \in \mathcal{S}^2$ stating that a can be performed in state s' leading game to state s''. Each of the game states is either intermediate state when one player can perform action, or terminal state when the game is finished. Randomness can be modelled by introducing additional player called *chance* that performs his actions with random distribution.

In games with imperfect information, some game states are indistinguishable from each other from the point of view of some agent; a set of such states is called *information set*. States in the same information set share known parameters of game state but differ in parameters that are not known to a player. One can tell what current information set is, but not in which state precisely. The set of information sets of player $n \in \mathcal{N}$ will be denoted W_n. It is worth to note that information set being a set of states is mathematical construction; practical implementation does not have to physically include a collection of states but simply not include all data or use its approximation.

Players select their actions following their *policy* or *strategy.* Common *policy* is defined as a function of state $\pi: \mathcal{S} \rightarrow \mathcal{A} \cup \perp$ returning action a or *none* symbol when no action is possible. Since the library presented in this paper uses the term *information sets* when relating to states viewed by an agent, it is more appropriate to define policy formally as a function of information set: $\pi_n: W_n \rightarrow \mathcal{A} \cup \perp$.

Game defines *payoff* function that returns game result evaluation. Typically, the domain for *payoff* function is the set of possible states $\mathcal{S}$. Some games define *payoff* only for final game states (e.g., classical chess), while others track players' scores dynamically. The latter presents a more generic approach as the first one can be transformed into it by defining payoffs in intermediate states as zero or some other neutral value. Intermediate payoffs allow the calculation of *rewards* as the difference between payoffs evaluated after and before acting.

Infrastructure Abstraction

Game model is built similarly like reinforcement learning solution described by Sutton and Barto (2018) consisting of:

- Environment - a game controlling entity responsible for collecting *agents'* actions, translating game's state and producing observations and rewards for *agents*.
- Agents - entities representing players in game, they observe changes in information set, perform actions and collect rewards. Agents may represent players with differently specified goals and, depending on game, may benefit from cooperation.

The core principle of the presented design is the thread separation of agents and environment; communication between entities is done via thread synchronisation mechanisms or network protocols. Although the diagram of data flow between agents and environment presented on Figure 1 seems analogous to one-agent problem, existence of multiple agents introduces several issues to be addressed.

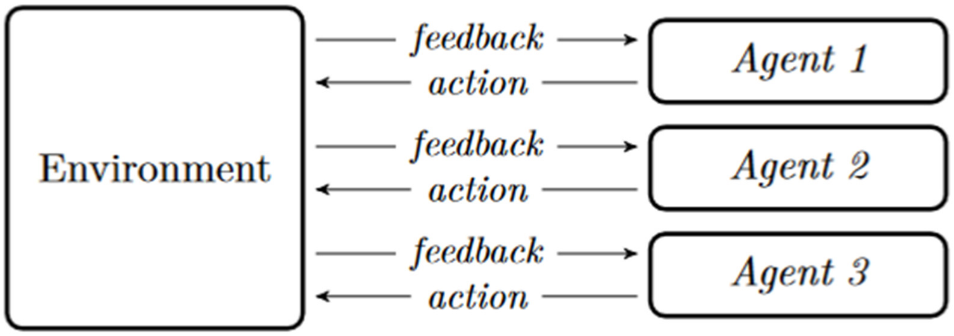

Fig 1: Model scheme of reinforcement learning

Firstly, when creating a multi-agent game model, one must take player ordering in consideration - to which the agent sends control and who expects action at certain moment from. Turn-based games define their own order; however, some games may expect simultaneous agent movement. Simultaneous actions can be modelled as ordered sequence. However, in order to preserve fairness, observations are distributed only after every agent authorized to perform action in time step has done so. Currently, only turn-based (one player at time) game flow is supported and simultaneous model support is to be implemented in the future.

Secondly, exact observation may differ across agents. Depending on game, not every action must have fully visible consequences for other players, these consequences may be observed later or even not be observed at all. Therefore, it is expected for the environment to produce observation updates in an independent manner for different agents. What is more, one player's action may produce varying observations for different agents. In some game models, after agent taking an action, he might see several observations before he is able to act again - these observations are produced by the environment while proceeding other players' actions. Similarly, agent might witness several rewards between his subsequent actions, as they are issued be the environment as payoff estimation or calculation changes. From the agent's point of view, the game step is measured since his committing action until the time point just before his next action, therefore the reward for step is sum of partial rewards issued by the environment in this time interval.

Game Domain

The key difference between *PettingZoo* and *Amfiteatr* libraries is communication between agents and environment. *PettingZoo* features two main API templates - AEC (Agent Environment Cycle) and Parallel API; both *PettingZoo's* APIs define interface for environment. There is no standard API for agents, and no automatic orchestration of entities is proposed. *Amfiteatr* resolves entity interaction via data protocol defined by *game domain. Game domain* collects types commonly agreed by every agent and environment on. With defined *domain*, abstract messages types are automatically derived. Currently, the following parameters form domain:

- **AgentId** - Type used to identify agents, usually it will be number, enumeration variant or string. Every agent must be provided with unique id, as it is used by the environment to select communication channels.
- **ActionType** - Represents player's action in game, every acceptable action must be representable as this type. Not every possible action of ActionType must be available (legal) for a particular agent at a particular step - in asymmetric games some actions may be allowed for some players and never allowed for others.
- **UpdateType** - Represents data that are produced by the environment as observation. Every possible observation

must be representable as this type. Similarly, not every observation in this format may be always accepted by a particular agent, issuing improper observation should cause error on the side of the agent.

- **UniversalReward** - Universally agreed payoff type. For evaluation purpose, it has several arithmetical requirements such as property of adding, subtracting and comparing. Usually, it is defined as a numeric value, yet it can be a different structure, provided that the said arithmetical requirements are met.
- **GameErrorType -** Type for descripting internal game errors. These errors may be produced by the environment and agents validating actions or observations during state transitions. Their main purpose is for debugging models and controlling flow when game rules have been violated.

Game state, information sets are not defined in *game domain*. Although those data structures must implement certain *traits* constrained by *game domain* to be compatible with game data protocol.
Naturally, it is possible to create separate threaded agents communicating with the environment using *PettingZoo*, yet thread synchronization and data transfer are currently out of the scope of library. Symmetrically, in *Amfiteatr*, traits implemented to build automatic game model can be used execute model step-by-step with lower-level interface, but automatic run with protocol is more native approach here.

Communication between Entities

With *game domain* defining game types as in abstract, data protocol is defined. Agents send messages of type *AgentMessage* informing about their actions, observed errors, or inform about disconnecting. The environment sends domain dependent messages *EnvironmentMessage* including rewards, observations, and error information; it also controls game flow with selecting agents to take actions and ending game. Each agent may be running on the same machine as the environment or may be delegated to other instance. Therefore, the medium for communication may differ regarding the model implemented; what is more, it may be expected to run some agents locally and some remotely. This leads to the introduction of trait interface for communication endpoint. The implementation of endpoint is required to send and receive adequate protocol messages via attached medium. Error in mediums must be captured and handled by endpoint implementation. Endpoint interface is type safe, as the agent's endpoint can only send *AgentMessage* defined by domain, and, when receiving, can only produce *EnvironmentMessage* or produce appropriate error.

Primary proposed endpoint uses standard inter-thread communication channel mpsc::channel. The advantages of this channel are its safety and speed, as it does not convert data during transit. The disadvantage of this implementation is its forced locality, as it is a thread communication mechanism and not a network protocol.

Network communication endpoint requires data serialization and deserialization when in move. Experimentally implemented TCP endpoint serializes in-memory message to binary data chunk, The chunk is being sent over TCP protocol to paired endpoint which then deserializes data reconstructing original in-memory message. to. Serialization and deserialization of messages is done in endpoint implementation. Conveniently, serialization methods can be automatically derived for most data types using existing serialization libraries e.g. serde (2024) and speedy (2024). Example of serialization of military Protocol Data Units (Committee D.I.S.S and others, 1998) has already been done by Scott et al. (2020). The construction of game domain over Protocol Data Units might be subject of a future implementation, as a step towards building military purpose models , and in future domain parameters compatible with standard might be released.

Flow of Multiplayer Game

Game session or simulation involving multiple actors needs to define how players interact and influence game state. In this section, the concept used in *Amfiteatr* is described and compared to design behind in *PettingZoo* (Terry et al., 2021). In both libraries, the central point of the game is environment, which performs state transition producing observation updates and rewards for agents. The core difference lays in the way the agents communicate with the environment. *PettingZoo* standardizes programmer API for the environment to invoke game step and retrieve observations. Typical game session is organized in loop over agents collecting observations, selecting action and stepping forward the environment. The agent logic is not specified. The environment does not run automatically, each game must be “stepped through”. This approach has the advantage of being straight-forward and simple. In contrast, developed *Amfiteatr* library takes the approach of the environment and agents running in separated threads and communicating with defined protocol. Every step, the environment chooses one agent and communicates *YourMove* signal to him; the agent then chooses action and communicates it to the environment. Then the environment makes a step which sets game in the new state and decides how the change is observed by any agent and constructs observation data for agents. One of the principles of *Amfiteatr* library is to provide a generic implementation of the environment and agents with pluggable logic for internal states and decision making while preserving type safety guarantees of the model. With environment's state, agents' information sets and policies implementing certain interfaces (called *traits* in Rust), it is possible to orchestrate them in separate threads and ensure they follow type safe protocol derived from game domain.

Partial Observations and Rewards

Each environment step triggered by one player's action results in the execution of game state transition. During state transition, observations and rewards are generated for agents (not limited to the next playing agent). It might be convenient to issue observations for certain agents after any player's action instead of in the moment just before his act time. In such case during interval appearing to a player as his single step – between his one action and his next action, he could observe facts several times and receive several rewards. An agent is then expected to apply observations (updates) chronologically and calculate reward as a sum of partial rewards received during the step. This approach enforces on agent supporting partial observations and rewards. Each game step, the environment may but does not have to issue an observation for certain player. Construction of environment issuing single observation and reward to the next playing agent is still possible. This versatility helps implementing game logic for games with large and complicated states. When observation is defined as current visible world (like state of chess board), partial rewards have no practical use; however, observation may be defined as some description of world changes (e.g. "unit A is observed to move north"); what is more, different agents can observe different things. In models with one-to-one mapping actions to observations, the environment would have to track individual observation stack since their last observation for every agent. Complicated games with asymmetric observation could benefit from concepts of partial observations. As stated before, partial observations and rewards are optional features and models with one observation per agent action are possible; thus, the Amfiteatr approach is compatible with AEC template environment of *PettingZoo* where agents observe their position in game just before their next action.

The example game flow clip is presented on sequence diagram on Figure 2; the diagram features environment providing *Agent 1* with two observations and two reward updates. In this example, *Agent 1* step is measured since his first taking action to the next one. His reward is the sum of partial rewards *Reward 1A* and *Reward 1B*. His information set is updated with stacking observations: $s' = \text{update}(\text{update}(s, \text{observation 1a}), \text{observation 1b})$. The approach used in *PettingZoo* AEC environment template would produce just one reward and observation for *Agent 1* – just before he takes second action (*Reward 1B*, *Observation 1B*). The approach of AEC can be translated into the one proposed in the *Amfiteatr* library ensuring the game step produces feedback only for the next player.

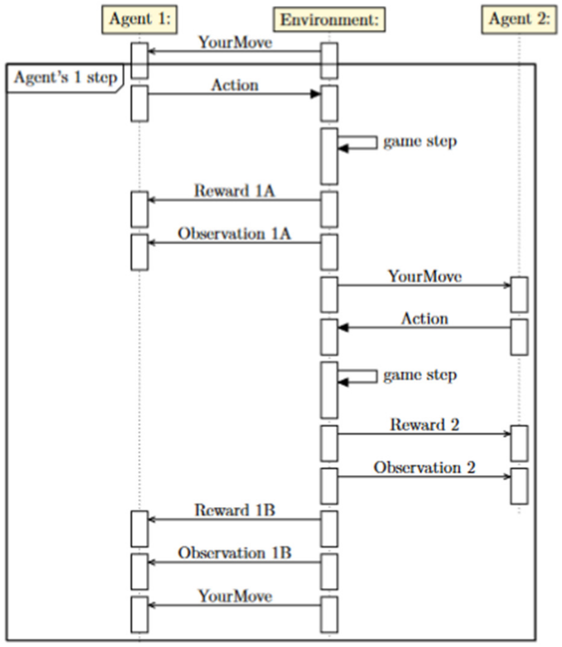

Fig 2: Sequence diagram for two player game with inter step feedback.

Generic Agent Construction

Model simulation in *Amfiteatr* requires agents to run automatically from the start of the game to the end. The automation of agent requires the ability to select action and update his information set. The current library version includes two generic implementations with pluggable information set and policy logic. One keeping track of game trajectory for further analysis (i.e. applying experience to learning policy algorithm), and the other for use in situation where experience collection is not needed (pre-trained agents or using explicitly defined policy). The generic implementation of Agent utilizes structures of *information set, policy* and *communication endpoint.* With these elements defined, the autonomous agent can be compiled.

Information Sets

As stated in game theory introduction, *information set* represents knowledge of agent regarding game state. Information set is not locked by the game's *domain parameters.* This allows building models with asymmetric agents with different view on the game. Some agents might be constructed to use raw observation data, and some might process observed data to calculate or approximate interesting unknown data. Compatibility with game protocol requires all *information sets* to meet certain programmer interface. Compatibility is enforced by the implementation of InformationSet trait, with the most crucial function of updating state. Update function must accept *UpdateType* defined in *domain parameters* and perform change on information set, producing error if the update cannot be applied. Error would then be automatically communicated by the agent to the environment. Error situations, in information set transition, should generally not occur in finished models, as update was previously issued by the environment, what suggests that either updated of information set or environment state processing has problems in implementation.

Policies

Policy represents function outputting action given the players' *information set*. Policy is implemented as a structure implementing Policy trait with associated type representing information set. Trait requires the implementation of select_action() function and producing action of type defined in *domain parameters.* As stated in the theoretical introduction, in some cases, agent may not have any possible action to choose, therefore policy output is wrapped in Option type, enforcing output consumer to check if action is not None.

Library is supposed to support the construction of reinforcement learning models. Typical implementation of learning agent involves using artificial neural networks to construct policy function. Commonly used policies can be classified as value-based type, policy gradient type, or as a hybrid of those two types. Critic type policies evaluate expected quality of actions and selects one with the best expected outcome. Example of such algorithm is Q-learning (Watkins and Dayan, 1992), and its neural network implementation DQN (Deep Q-learning Network) used in Mnih et al. (2013). Q-learning technique tries to learn parameters to estimate Q-function, given by equation:

$$Q_{\pi}^{*}(s,a) = r + \gamma Q_{\pi}^{*}(s',a')$$

The value of Q-function for policy π, information set s, and action a is immediate reward r summed with $Q - value$ in the next observed information set following the same policy π for the rest of the game. Future rewards are multiplied by discount factor γ, typically $\gamma < 1$ and it is responsible for inflation of rewards and making policy convergent. DQN uses neural network to estimate Q-function, therefore the application of that policy depends on the possibility of representing information set and action as tensor input to DQN, as presented on Figure 3.

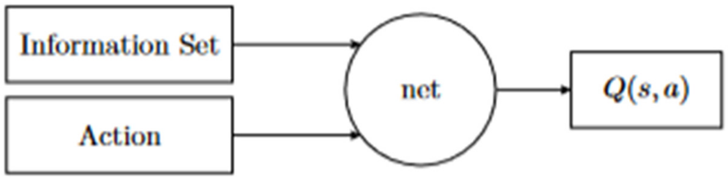

Fig 3: Scheme of DQN

Alternative family is made of policy gradient methods. These policies rather than evaluate the quality of particular actions, they analyse state (information set) and output distribution of actions to sample. If action space is a continuous policy, then output of network can be used as action realisation. For discrete action space, problem can be viewed as a categorisation problem, where information set is "labelled" with adequate action. The learning process involves updating policy parameters, so it produces distributions favouring better actions. Example of critic policy is algorithm REINFORCE introduced by Williams (1992) and further discussed by Lehmann (2024). Algorithm works by updating network parameters according to policy gradient theorem. The implementation of neural network-based policy gradient methods requires that information set is representable as tensor just like in value-based policies. Action does not need to be convertible to tensor, however it must be able to construct from the sample taken from the output of the network. The scheme of neural network-backed policy gradient is presented on Figure 4.

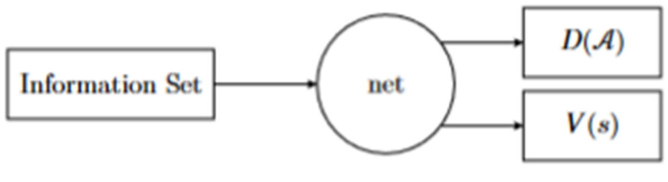

Fig 4: Scheme of policy gradient neural network on example of A2C

Currently, library provides generic implementations for DQN and A2C (Advantage Actor Critic) policies. These policies can be parametrized with needed neural network shape, initial values and selected optimizer. They are implemented automatically for information sets and actions, provided that they implement needed interfaces discussed above. Both types of learning policies require experience collected in past games. Library provides generic tracing agent implementation that builds and stores the history of visited information sets, taken actions and collected rewards. At given time, agent can update one trajectory related to the current game; finished trajectories are stored in the history vector that can be used later to construct the batch of training data. Learning interface supports fitting policies to optimize payoffs distributed by the environment or custom values calculated from information set. This feature is meant to help modelling agents with their own agenda, without changing their payoff function in the environment.

Performance

Performance comparison with PettingZoo multi-agent reinforcement learning library

In order to measure potential performance gains of using Rust powered *Amfiteatr* library, the experiment using *PettingZoo's* game Connect Four environment has been performed. For the experiment, the following models were built:

- Python - PettingZoo *Connect Four* game environment with agents implemented in Python.
- Wrapped - Amfiteatr model with environment state being a wrapper around original Python *PettingZoo's Connect Four* Environment.
- Rust – Pure Amfiteatr model with environment state being rewritten in Rust.
- RustLight – Pure Amfiteatr model with environment state being rewritten in Rust and compiled without optional extensive logging in core library.

Rust environment state implementation has been done preserving original operations without attempts to optimise code in game logic area. In every variant neural network backend used was *libtorch.*

For every model template, there were constructed agents using Advantage Actor Critic (Lehmann, 2024) policy. Every launch consisted of 100 learning epochs, each consisting of 128 episodes. Between each learning epoch, 100 test episodes were performed. Tests were performed for different sizes of hidden layers of neural network. For most of the cases, the experiment was launched 50 times. Due to long time of evaluation, Python model with layer of size 10000 was launched 10 times. Tables 1 and 2 present average execution time for models with different sizes of hidden layers of neural network; standard deviation is given in parenthesis. Table 1 is dedicated to models with single hidden layer and Table 2 gathers results for models with two hidden linear layers with *tanh()* layer between.

Table 1: Execution times of models with single hidden linear layer [s]

(standard deviation in parenthesis)

	Layer size				
Variant	**1**	**10**	**100**	**1000**	**10000**
Python	70.62 (2.39)	78.99 (0.79)	87.84 (0.60)	101.50 (0.59)	3473.97 (19.30)
Wrapped	72.92 (3.17)	76.01 (1.52)	77.59 (1.27)	85.13 (1.23)	108.25 (5.51)
Rust	22.44 (1.25)	24.68 (3.04)	24.52 (0.60)	28.22 (0.81)	50.13 (2.69)
Rust Light	22.58 (2.71)	24.00 (0.53)	24.61 (0.59)	28.45 (0.91)	47.99 (2.82)

Table 2: Execution times of models with two hidden linear layers [s] (standard deviation in parenthesis)

	Layers size			
Variant	**1,1**	**10,10**	**100,100**	**1000,1000**
Python	73.01 (3.37)	83.57 (0.72)	98.35 (0.64)	128.28 (3.89)
Wrapped	70.92 (4.73)	78.16 (1.75)	80.99 (1.37)	140.81 (4.14)
Rust	24.27 (4.61)	26.76 (2.39)	27.66 (0.71)	50.18 (3.11)
Rust Light	23.34 (1.79)	26.26 (0.86)	27.58 (0.69)	49.82 (3.57)

The presented results show that generally models built in Rust execute faster than those written in Python. In tested cases, Rust powered model with both agent and environment written in pure Rust was usually between 2 and 3 times faster than Python implementation. *Amfiteatr* wrapping of *PettingZoo*'s Python environment performs slightly better than a model in Python. Better performance of Rust models is coherent with expectation as it is a language producing faster programs in general; however, despite being a slower language, Python models benefit from fast tensor operations provided by low-level *libtorch* implementation. An interesting observation was made for a single layer size of 10000 in Python: execution time has drastically grown. In additional tests, such observations have been observed in Rust models with a greater number of nodes in network. In this case, Python model was executed with average 5,6 million involuntary context switches by work scheduler caused by time expiration of time slice, as contrast Rust model executed with average 6571 involuntary context switches. All experiments were performed on personal computer without CUDA support. It is possible that dedicated GPU executed kernels prevent the occurrence of such cases. It is expected that differences between the execution time of *Amfiteatr* and *PettingZoo* models will be more stable in setups dedicated to machine learning.

Performance evaluation of communication via mpsc channel and TCP socket

To compare standard inter-thread channel and experimental communication channel via TCP protocol, the model for multi-armed bandit problem with multiple players has been built. The model was executed with 1, 10, 100, 200, 300, 400 and 500 active players, performing action sampled from uniform distribution. Every TCP agent was executed on localhost, reducing the influence of network latency. Due to the experimental state of TCP based communication medium, executions for 1000 and 10000 agents failed. The experiment was designed with possibly simple game with minimal game logic computation and used no tracing agents to maximalize computational ratio for communication operations and other. For every communication variant and number of active agents, the experiment was repeated 50 times. Means are presented in Table 3; standard deviations are placed in parenthesis.

Table 3: Model evaluation times for different communication mediums [s]

(standard deviation in parenthesis)

	Active Agents						
Variant	1	10	100	200	300	400	500
std::mpsc	0.12 (<0.01)	0.14 (<0.01)	2.48 (0.01)	9.38 (0.04)	20.92 (0.13)	36.80 (0.27)	56.70 (0.13)
TCP based	4.10 (0.01)	4.21 (0.01)	15.39 (0.07)	48.21 (0.21)	142.19 (3.70)	255.24 (4.93)	393.70 (1.42)

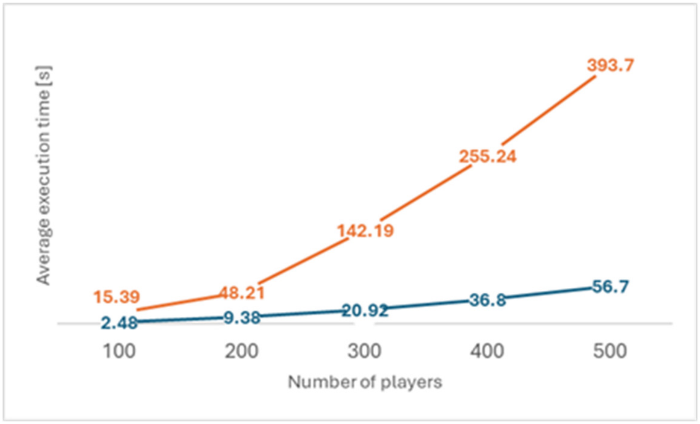

Fig 5: Chart of execution times comparing mpsc: channel with TCP stream

Performance experiment has shown that single host model executions perform better, significantly better than distributed. Local inter-thread communication channel will always be faster. Here model mpsc::channel has performed about 6 times better. In some cases of models with great amount of game logic computation, the relative time cost of network communication may be small. However, for the majority of models, it is more suitable to add computational power to local machine that grows network distributed environment. Further performance tests on stabilized network communication medium might help decide whether or when it is beneficial to build distributed models.

Future work

Library is in early stage of development and serious amount of work is to be done yet. Currently, locally launched as well as network distributed games can be launched using game protocol. Repeated model execution can be orchestrated with local control – in single program controlling all actors. A protocol allowing synchronizing agents and environment in repeated execution is yet to be implemented. The repetition of experiments is essential in reinforcement learning process, therefore construction of distributed multi-agent reinforcement learning models is not yet supported by the library. . What is more, current network communication medium is not stable and needs more development work. Beside solving problems with distributed simulation, more generic implementations of learning policies are planned. With the development of the library, more performance benchmarks are to be made, especially with a wider selection of hardware. Last but not least, support for parallel environments with similar purpose like Parallel API in *PettingZoo* is to be provided.

Conclusion

This paper presented the concept of Rust powered multi-agent reinforcement learning library. The presented work includes architecture description and benchmarks of game model execution related to community standard library. Library in current stage is fit to launch local reinforcement learning sessions of sequentially ordered agents. Performance benchmarks show slight performance gain in models compiled in Rust with the usage of *Amfiteatr* library. Comparing to *PettingZoo,* the library is in its early stage – API is not yet stable; fully parallel environment is yet to be built and the number of implemented game problems is much less. Nevertheless, library might find its niche for specific deployments. Thread and type safety provided by Rust language can be leveraged to build learning algorithms to solve decision making problems with hight reliability. Thus, future and more stable versions of library might be used in critical infrastructure and industry projects or by researchers willing to use Rust in their work.

References

- Ameljańczyk, A. (1980) 'Teoria gier i optymalizacja wektorowa', WAT, Warszawa [Preprint].
- Binmore, K. (2007) Game theory: a very short introduction. OUP Oxford.
- Committee, D.I.S.S. and others (1998) 'IEEE standard for distributed interactive simulation-application protocols', IEEE Standard, 1278, pp. 1–52.
- Klabnik, S., Nichols, C. and Community, R. (2024) The Rust Programming Language. Available at: https://doc.rust-lang.org/stable/book/.
- Lehmann, M. (2024) 'The Definitive Guide to Policy Gradients in Deep Reinforcement Learning: Theory, Algorithms and Implementations', arXiv preprint arXiv:2401.13662 [Preprint].
- Mnih, V. et al. (2013) 'Playing atari with deep reinforcement learning', arXiv preprint arXiv:1312.5602 [Preprint].
- Mnih, V. et al. (2015) 'Human-level control through deep reinforcement learning', nature, 518(7540), pp. 529–533.
- Scott, N.W. et al. (2020) 'Using Serde to Serialize and Deserialize DIS PDUs', in 2020 International Conference on Computational Science and Computational Intelligence (CSCI), pp. 1425–1428.
- 'serde' (serialization/deserialization library). Available at: https://github.com/serde-rs/serde (Accessed: 23 March 2024).
- Silver, D. et al. (2016) 'Mastering the game of Go with deep neural networks and tree search', nature, 529(7587), pp. 484–489.
- Silver, D. et al. (2017) 'Mastering chess and shogi by self-play with a general reinforcement learning algorithm', arXiv preprint arXiv:1712.01815 [Preprint].
- speedy (no date) https://github.com/koute/speedy. Available at: https://github.com/koute/speedy (Accessed: 23 March 2024).
- Sutton, R.S. and Barto, A.G. (2018) Reinforcement learning: An introduction. MIT press.
- Terry, J. et al. (2021) 'Pettingzoo: Gym for multi-agent reinforcement learning', Advances in Neural Information Processing Systems, 34, pp. 15032–15043.
- Towers, M. et al. (2023) 'Gymnasium'. Zenodo. Available at: https://doi.org/10.5281/zenodo.8127026.
- Watkins, C.J.C.H. and Dayan, P. (1992) 'Q-learning', Machine learning, 8, pp. 279–292.
- Williams, R.J. (1992) 'Simple statistical gradient-following algorithms for connectionist reinforcement learning', Machine learning, 8(3), pp. 229–256.