CSC 36000:
Modern Distributed Computing *with AI Agents*

By Saptarashmi Bandyopadhyay
Email: sbandyopadhyay@ccny.cuny.edu
Assistant Professor of Computer Science
City College of New York and Graduate Center at City University of New York

October 22, 2025 CSC 36000

Today's Lecture

Recap: Multi-Agent Reinforcement Learning

Distributed Machine Learning

- **Centralized Parameter Server**
- **Decentralized Ring All-Reduce**

Ray RLLib Code Example (compared to JAXMARL)

Future of Distributed Multi-Agent AI (Oct 24)

Multi-Agent Reinforcement Learning

Recall: Single-Agent to Cooperative Multi-Agent RL

- In Single-Agent RL, we want our agent to learn the optimal policy that takes a state *s* and tells us an action to take *a*

$$\pi^*(s) \rightarrow a$$

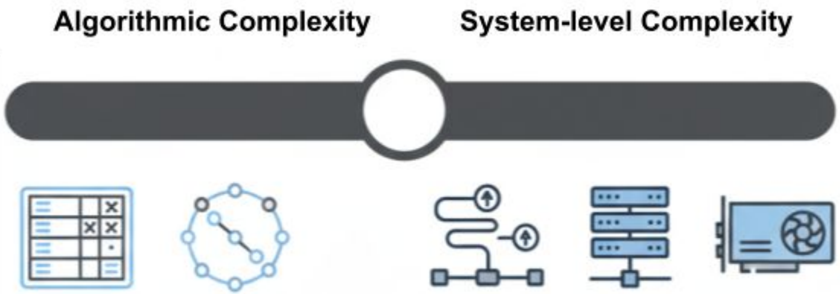
- In Multi-Agent RL, we extend this idea to multiple agents, often resulting in learning a joint action-value function

$$Q_{tot}(s, \mathbf{a})$$

  that gives us multiple actions, one for each agent
- Algorithms like VDN and QMIX learn to optimize this function, teaching the agents how to coordinate in the process. What happens when we need more than one machine to do this?

From Algorithmic Complexity to System Complexity

- In MARL, the problems we usually deal with are *algorithmic*
  - Coordination, Credit Assignment, etc.
- At scale, we have *system-level* challenges
  - Computation, Memory, Bandwidth, etc.
- We need to consider not only *what* we compute, but *how*
- This is how we transition from a pure AI problem to AI + distributed systems problem



Three Problems in Distributed MARL

1. **Non-stationarity:** The environment is constantly changing as all other agents update their policy, resulting in a moving-target problem
2. **Credit Assignment:** With so much happening, how do you assign credit or blame to a single agent's action?
3. **Curse of Dimensionality:** The joint state-action space grows exponentially with the number of agents:

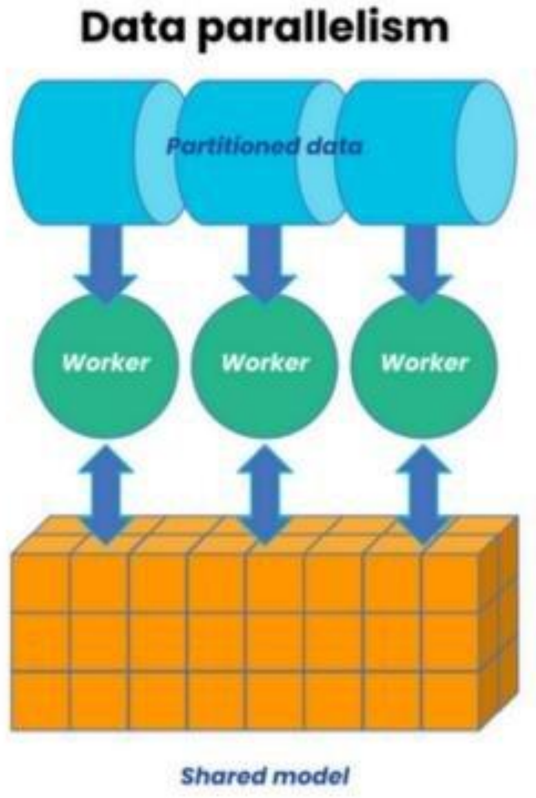
$$(|S \times A|^N)$$

   this is computationally intractable!

Architectural Patterns for Distributed Learning

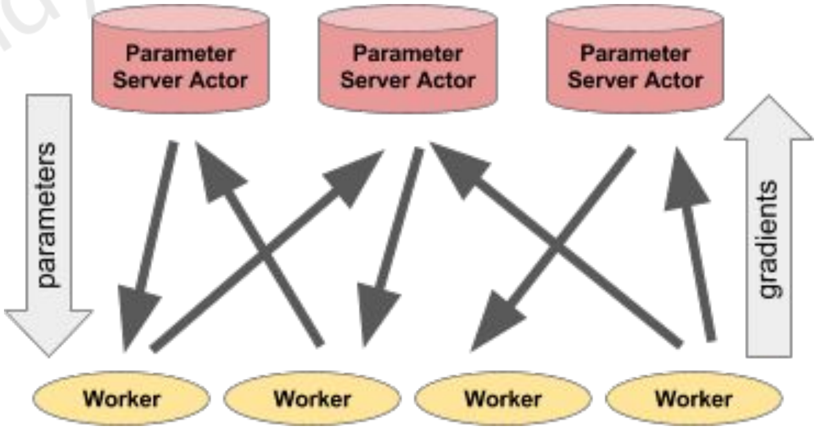
Gradient Aggregation

- In data-parallel training, we have a copy of the model on each worker node (e.g. GPU)
- Each worker computers gradients on its local batch of data (e.g. agent experiences)
- These gradients need to be aggregated across all the workers before the next update
- This step is the primary bottleneck in distributed computing



Architecture 1: Parameter Server

- **Characteristics:** Centralized, Client-Server
- **Server Nodes:** Maintain authoritative global copy of the parameters
- **Worker Nodes:** Pull the latest parameters from the server, compute gradients on local data, and push gradients back to server
- The server must then aggregate the gradients and update the global model



Parameter Server: Consistency Models

- **Synchronous Training:** The server wait for *all* the gradients before updating the model
  - **Pro:** Guarantees consistency, statistically equivalent to single-machine training
  - **Con:** Performance is constrained by :"stragglers")
- **Asynchronous Training:** Update occurs when *any* gradient is received
  - **Pro:** High throughput, robust to stragglers and failures
  - **Con:** Workers compute gradients using outdated parameters, which harms model convergence

The Achilles Heel

- The centralized server needs to handle the communication with all N workers
- *Fan-out* of weights and *Fan-in* of gradients centers *all* network traffic on the server node
- As N grows the network bandwidth becomes the bottleneck
- This fundamentally limits how large a parameter server can scale

Architecture 2: Ring All-Reduce

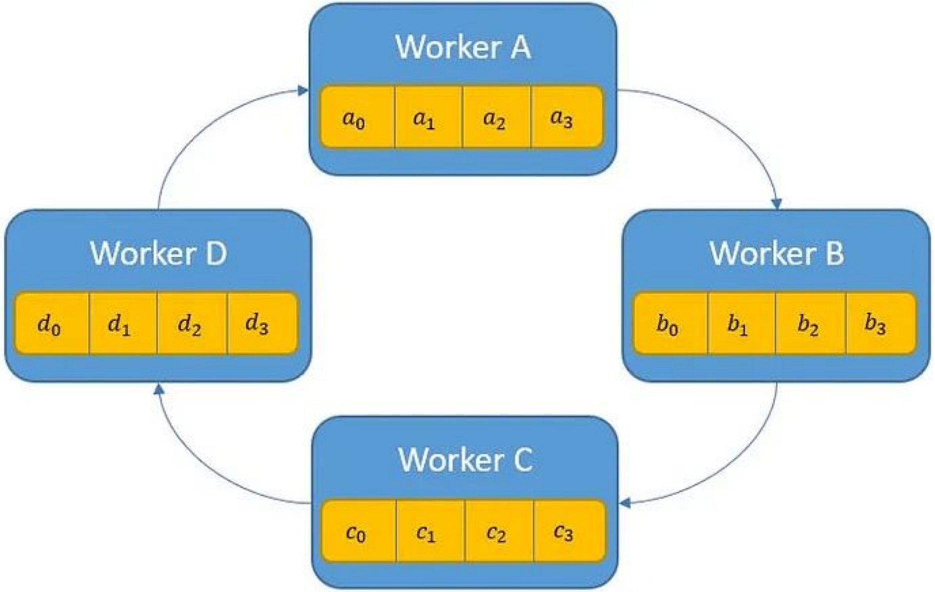
- **Characteristics:** Decentralized, Peer-to-peer
- ***All-Reduce*:** A collective communication primitive - all nodes participate, all nodes receive the final aggregated result
- In the **Ring All-Reduce** algorithm, each node only communicates with its two immediate neighbors, forming a logical ring
- Two Phases: ***Reduce-Scatter*** and ***All-Gather***

Ring All-Reduce

- **Phase 1: Reduce-Scatter** (or Share-Reduce)
  - In $N-1$ steps each node sends a chunk to its right neighbor and receives a chunk from the left neighbor
  - It adds the incoming chunk to its corresponding chunk
  - Result: Each node holds one chunk of the final, fully-summed vector
- **Phase 2: All-Gather** (or Share-Only)
  - In $N-1$ nodes circulate their summed chunks
  - Each node now has a copy of the final result

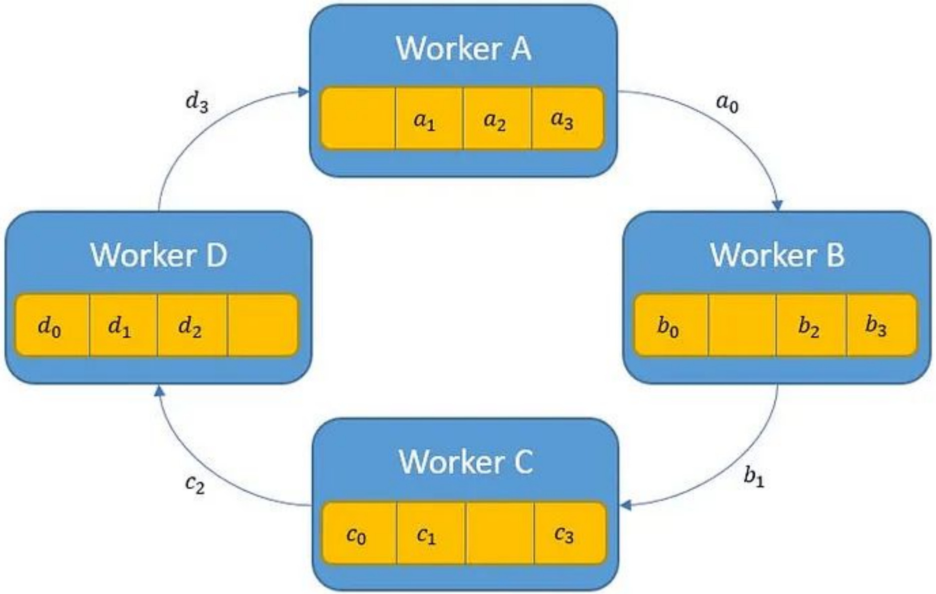
Example: Ring All-Reduce

- Start with a ring of nodes. Each node has a "piece" of the final result
- Our goal is to have each node hold the sum of the pieces



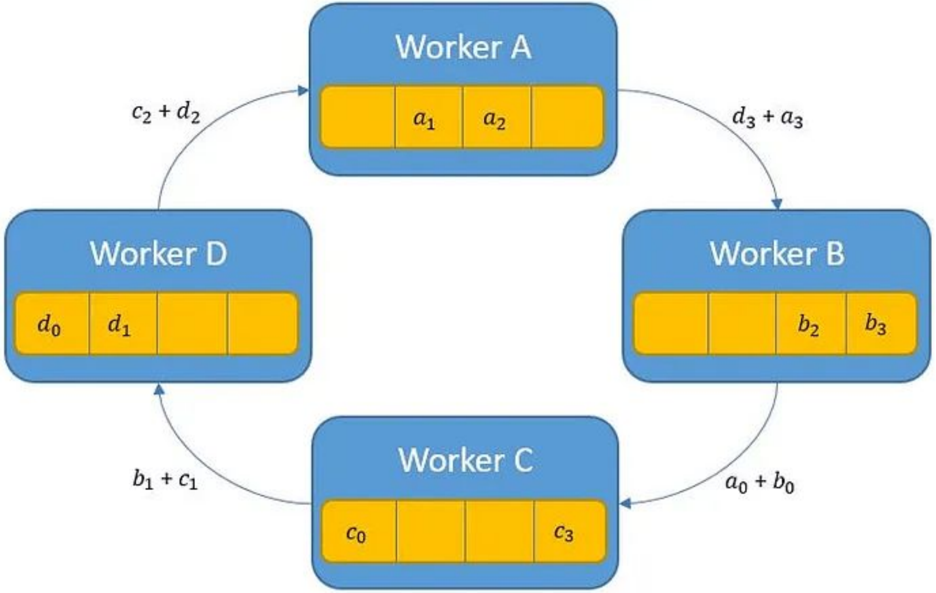
Example: Ring All-Reduce

- **Reduce-Scatter:** Each node sends a chunk to its next neighbor and receives a chunk from its previous neighbor



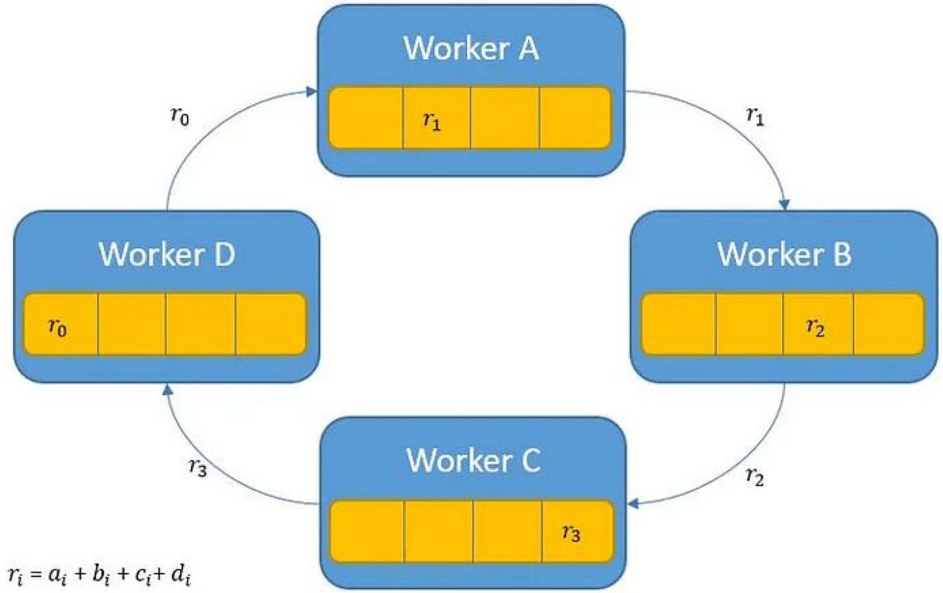
Example: Ring All-Reduce

- **Reduce-Scatter:** As the reduce continues, each node builds up a single piece of the final, aggregated result



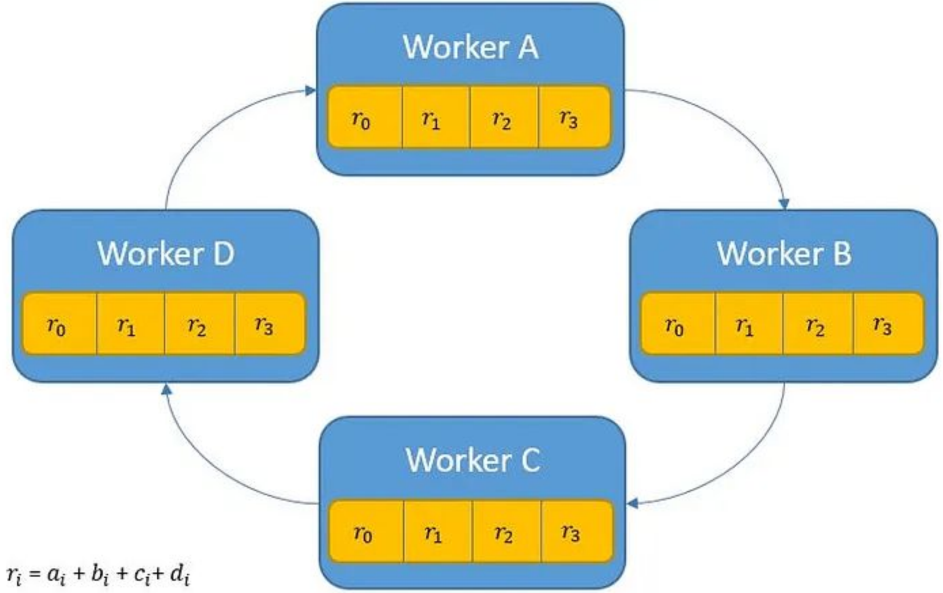
Example: Ring All-Reduce

- **Reduce-Scatter:** At the end of the first phase, each node holds a chunk of the final result



Example: Ring All-Gather

- **All-Gather:** In Phase 2, we simply pass along the chunks of the final result in a similar way to how we passed chunks in All-Reduce
- This time, we only store (*share*), we do not add (*reduce*) the data



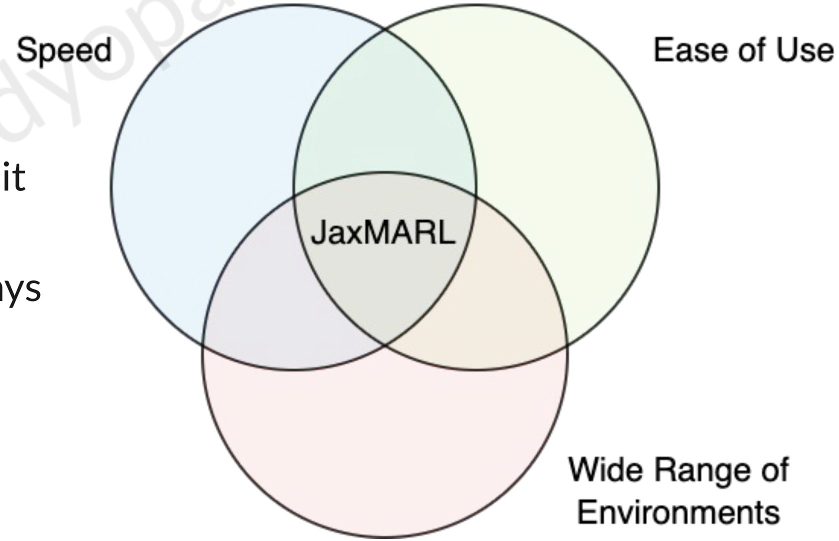
Ring All-Reduce Bandwidth Optimality

- In the Parameter Server, our communication load was $O(N \cdot M)$, where N is the number of worker nodes and M is the model size
- In Ring All-Reduce, each node sends and receives a total of $2 \times \frac{N-1}{N} \times M$ bytes of data
- As N grows, this approaches $2M$, so the communication cost per node becomes *independent* of N
- This is why we call All-Reduce *bandwidth-optimal*

MARL in Practice: JaxMARL

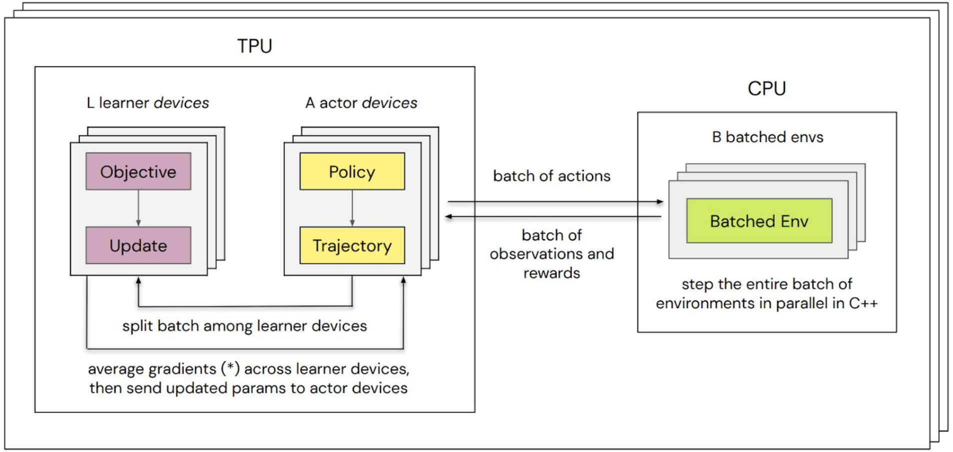
Introduction

- Multi-Agent RL can be pretty slow
- JAX-enabled hardware acceleration can make it 12500x faster
- This means that experiments that once took days now take hours or minutes



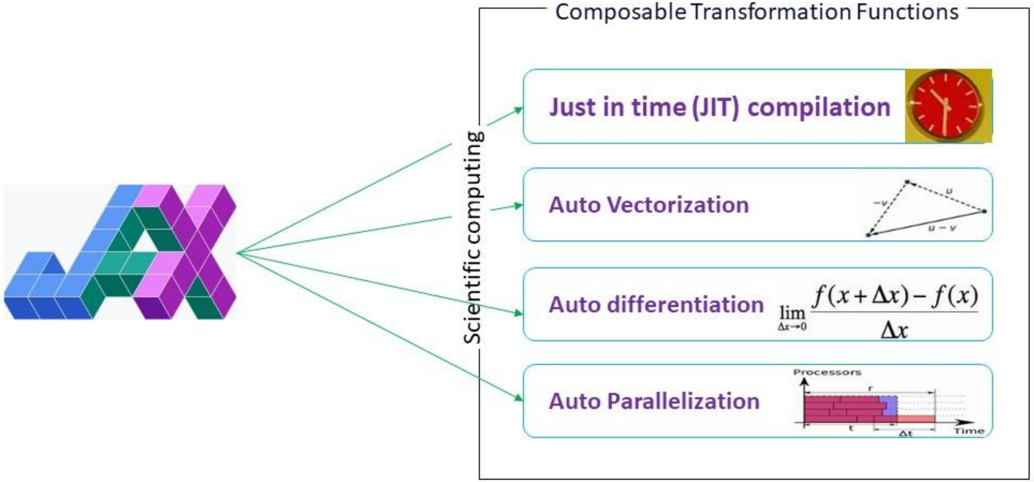
Recap: Non-JAXMARL Training Approaches

- Current RL environments before JAXMARL are normally run on CPU
- RL training and inference on the environments are done on the hardware accelerator
    - GPU or TPU
- Research Question: Why not run many environment threads in parallel?
    - Can help to increase training speed?!

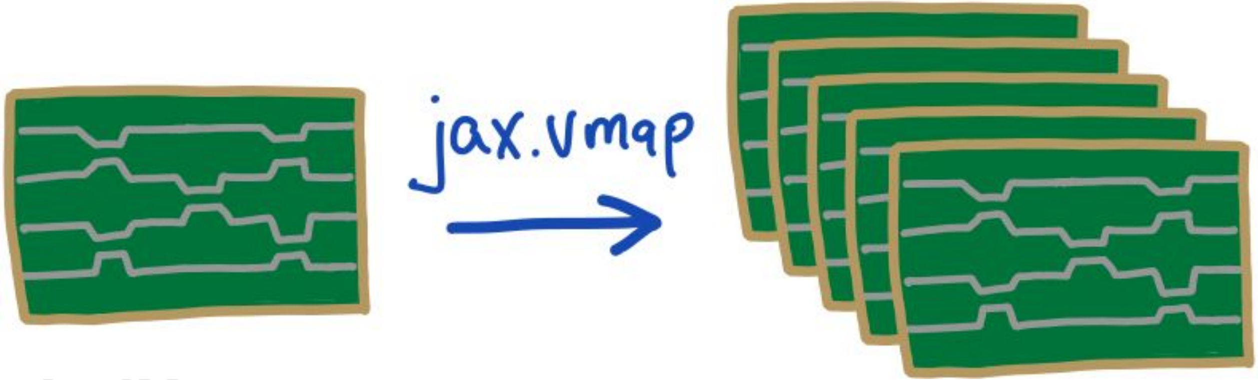


This entire computation is replicated across S slices of a TPU Pod, in which case gradients in (*) are averaged across all learner devices of all slices

Use JAX for Multi-Agent Training



Vectorizing Environments with JAX vmap



Using JAX with JIT

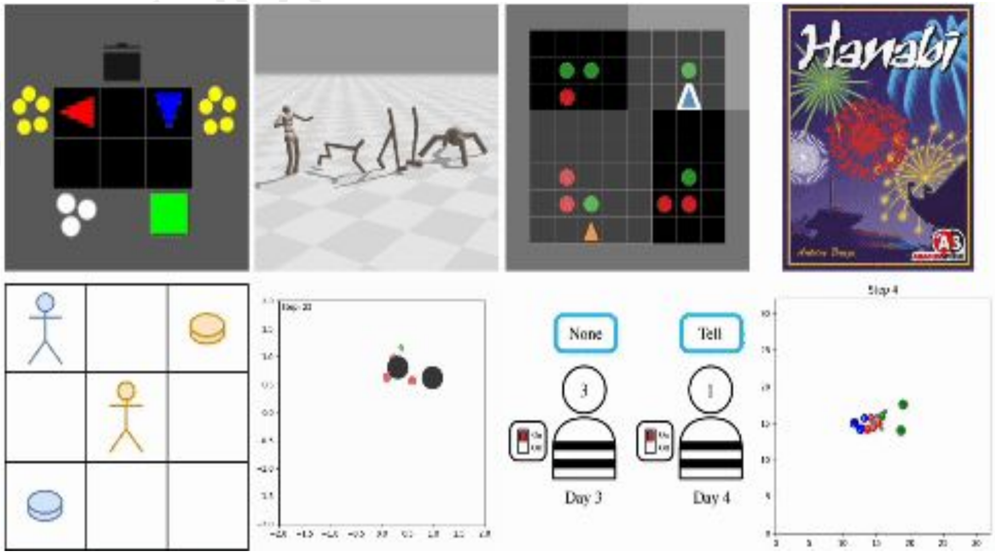
- Vmap many environments in parallel for faster runtime!
- Avoid slow CPU-GPU data transfer
- Avoid Python command overhead
- Allow for Jax JIT optimisation (operator fusion)
- Avoid messy and complicated multi-processing setups!

Multi-Agent Environments

- We implement 8 popular MARL environments
- We provide Q-learning and PPO

```python
import jax
from jaxmarl import make

key = jax.random.PRNGKey(0)
key, key_reset, key_act, key_step = jax.random.split(key, 4)

# Initialise and reset the environment.
env = make('MPE_simple_world_comm_v3')
obs, state = env.reset(key_reset)

# Sample random actions.
key_act = jax.random.split(key_act, env.num_agents)
actions = {agent: env.action_space(agent).sample(key_act[i]) \
            for i, agent in enumerate(env.agents)}

# Perform the step transition.
obs, state, reward, done, infos = env.step(key_step, state, actions)
```

Overcooked MABrax STORM Hanabi



Coin Game Multi-Particle Tag Switch riddle Starcraft (SMAX)

Questions?