Optimizing Distributed Reinforcement Learning with Reactor Model and Lingua Franca

Pranav Atreya   **Jacky Kwok**   **Amil Dravid**

1 Extended Abstract

Distributed Reinforcement Learning (RL) frameworks are essential for mapping RL workloads to multiple computational resources, allowing for faster generation of samples, estimation of values, and policy improvement. These computational paradigms require a seamless integration of training, serving, and simulation workloads. Existing frameworks, such as Ray [22], are not managing this orchestration efficiently, especially in RL tasks that demand intensive input/output and synchronization between actors.

As such, we have optimized *Lingua Franca*, a polyglot coordination language, for synchronized parallel RL workloads. The development of our proposed framework *Lingua Franca for Reinforcement Learning (LFRL)* represents several significant contributions to the field: LFRL is a distributed framework that integrates the critical components of RL tasks (training, simulation, and serving) into a single, cohesive system. LFRL is built on top of the reactor model [16] and supports true parallelism in Python, thereby significantly decreasing synchronization and I/O overhead. LFRL also supports seamless distribution of computation to GPUs when that computation can benefit from it, further accelerating deep RL workloads.

On average, LFRL outperformed Ray in generating samples from OpenAI Gym and Atari environments by 1.21x and 11.62x, reduced the average training time of synchronized parallel Q-learning by 31.2%, and accelerated Multi-Agent RL inference by 5.12x. LFRL represents a leap forward in the field of distributed RL, particularly in better satisfying the particular needs of deep reinforcement learning workloads. Its unique design principles and innovative features position it as a critical tool for researchers and practitioners in the rapidly evolving landscape of RL.

2 Introduction and Related Work

The field of machine learning (ML) has witnessed an exponential increase in computational requirements for training models, which tend to be increasingly larger deep neural networks. This complexity has necessitated the creation of new frameworks focused on training these networks and leveraging specialized hardware to reduce training times. Examples include TensorFlow [1], MXNet [7], PyTorch [26], and Deepspeed [28]. Beyond classical supervised learning, emerging AI applications are increasingly required to operate in dynamic environments and pursue long-term goals, problems that reinforcement learning (RL) are well suited for. RL involves learning to behave in uncertain environments based on feedback from interaction with the environment. Optimal behavior can be specified via appropriately chosen reward functions of the data. This approach has already led to significant achievements such AlphaGo [35], and more recently, the success of ChatGPT [23]. RL applications span various domains, including traffic systems [6], UAVs [13], large language models (LLMs) [23], and dexterous manipulation [2].

Deep RL presents new additional challenges beyond traditional RL for distributed frameworks. Deep RL, unlike its traditional counterparts, are typically applied in continuous state space environments, increasing the complexity of the task and thus the computational burden. In fact, with many system optimization problems the CPU is heavily utilized by deep RL training workloads [3]. However, the scalability of deep RL, particularly in learning complex state-action associations, hinges on efficiently leveraging both CPUs and GPUs [12, 8]. The combination of RL with deep neural networks necessitates a balanced approach in computational resource allocation. The processing speed, especially when updating policies involving millions of parameters, becomes a critical factor. The need to balance CPU and GPU resources, coupled with the limitations of existing frameworks, highlights the need for innovative approaches in developing efficient, scalable, and versatile systems capable of supporting the dynamic and complex nature of modern artificial intelligence and machine learning applications. This paper aims to delve into these challenges and explore potential solutions, paving the way for more efficient and effective reinforcement learning systems.

To this end, we introduce an optimized version of Lingua Franca (LF) [17]. Our optimized version of LF, which we dub *Lingua France for Reinforcement Learning (LFRL)* is tailored to address the unique challenges of RL applications.

LFRL stands out in its ability to effectively handle a diverse range of workloads, from lightweight, stateless computations needed for simulation to the more intensive, long-running computations required for training.

Contributions:

1. In this paper, we draw the connection between Lingua Franca and distributed RL, developing a runtime implementation (LFRL) based on LF for RL applications.

2. We demonstrate how the reactor-oriented paradigm implemented in LFRL optimizes the utilization of computational resources and efficiently parallelizes RL workloads.

3. We then describe the compilation process and the mechanisms of LFRL runtime, illustrating how it exploits parallelism and differs from the actor-based framework, Ray.

4. Our investigation also reveals that multithreading offers greater advantages than multiprocessing in distributed RL. To leverage these advantages, we modified the Lingua Franca compiler and introduced an optimized Reactor C runtime with No GIL Python, enabling true parallelism and abstracts away the burden of coordinating worker threads for RL tasks.

5. We further present an extensive evaluation on the generations of samples from Open AI Gym and Atari Environments, synchronized parallel Q-learning, and inference of Multi-Agent RL. Our results demonstrate that LFRL outperforms Ray in terms of training, inference, and simulation for RL.

3 Distributed Reinforcement Learning and Requirements

Reinforcement Learning (RL) algorithms aim to enhance an agent's policy performance within a given environment, often represented through a simulator. These algorithms alternate cycles of data collection with the latest policy, value estimation using the latest data (and possibly data collected from previous policies), and policy improvement. Depending on the signal provided by the reward function on the collected data and the efficiency of the algorithm itself, the reinforcement learning process can vary in the amount of data and computational resources needed to run until satisfactory performance is achieved. For most tasks of interest, these data and compute requirements are large, motivating distributed reinforcement learning frameworks that can more efficiently leverage parallel computational paradigms.

Many aspects of the reinforcement learning problem can be parallelized [30]. Data collection can be split across multiple worker threads, learning can be split across multiple worker threads with each thread maintaining its own value function and policy parameters, updates to the neural networks can be performed in parallelized fashions using popular deep learning frameworks [1, 4, 25], and simulators can themselves be parallelized, for example on accelerated computing infrastructure [27, 37, 31, 34].

3.1 Variants in RL Algorithms

Single-Agent Training: The most fundamental scenario in RL is training a single agent, which involves repeatedly applying the steps of rollout, replay, and optimization. Synchronous algorithms like A2C [19] and PPO [32] follow these steps in sequence, whereas asynchronous variants (e.g., A3C [19], Ape-X [11], APPO [38], IMPALA [9]) overlap rollout and optimization steps to enhance data throughput.

Multi-Agent Training: Multi-agent training involves multiple agents interacting within the environment, either cooperatively or competitively. While the dataflow structure resembles that of single-agent training, complexities arise when customizing training for individual agents. For instance, if agents require optimization at different frequencies or are trained with distinct algorithms, the training dataflow must accommodate multiple iterative loops with varying parameters.

Model-Based Algorithms: Model-based RL algorithms aim to learn the transition dynamics of the environment to increase training efficiency [21]. This adds a supervised training component to standard distributed RL, involving training one or more dynamics models with environment-generated data.

3.2 Challenges and Opportunities in Framework Optimization:

Deep RL algorithms like Q-learning and SARSA (State-Action-Reward-State-Action) traditionally relied on sequential learning of a value table. The advent of deep neural networks has enabled these algorithms to approximate complex functions without an explicit value table. The combination of RL with deep neural networks in deep RL models necessitates a balanced approach in computational resource allocation. The processing speed, especially when updating policies involving millions of parameters, becomes a critical factor. Current popular algorithms, such as Deep Q Networks (DQN) and Advantage Actor Critic (A2C), often employ a hybrid approach using

both CPUs and GPUs. This setup allows for the efficient execution of different phases of the RL process, with policy evaluation typically occurring on the CPU and policy updates on the GPU. The evolution of machine learning, especially in the realm of RL, has ushered in a new era of computational requirements and challenges.

Frameworks like Ray have made strides in CPU/GPU orchestration for RL, but they face efficiency challenges due to the actor model's inherent communication overhead and synchronization demands. Ideal frameworks for RL should support heterogeneous computations, flexibility in computational models, dynamic execution, and large data handling, while integrating seamlessly with deep learning libraries and simulation frameworks.

We argue that distributed RL frameworks can better handle millions of tasks per second by providing deterministic concurrency. Current frameworks, on the other hand, fall short in efficiently meeting the evolving demands of AI applications, indicating a significant gap in the field.

This research aims to propose a novel system architecture that addresses these comprehensive requirements, bridging the gap in the current landscape and pushing the boundaries of what's possible in distributed RL.

4 Introduction to Actor and Reactor Model

4.1 Actor Model

The Actor model [10], a foundational concept in concurrent computing, emerged in the 1970s as a response to the increasing complexity and interactivity in computer systems. It introduced a novel way to handle concurrent operations by conceptualizing "actors" as the primary units of computation. These actors are analogous to objects in object-oriented programming (OOP) [29], but they are designed specifically to address the challenges of concurrency. Each actor represents a self-contained unit with its own local state, and actors interact with each other exclusively through asynchronous message passing. This model contrasts with traditional approaches that often rely on shared state and synchronization mechanisms like locks, which can lead to issues like deadlocks and race conditions.

Actors in the model operate independently and concurrently, providing a natural framework for distributed and parallel systems. When an actor receives a message, it can perform several actions: it can create more actors, send messages to other actors, modify its own internal state, or decide how to respond to the next message it receives. This makes the model highly adaptable and scalable, suitable for applications ranging from simple concurrent programs to complex distributed systems. The asynchronous nature of message passing in the Actor model is key to its effectiveness in dealing with concurrency. It allows actors to send and receive messages without waiting for a response, thereby preventing bottlenecks and enabling continuous operation even when certain components are busy or delayed.

Furthermore, the Actor model introduces a flexible approach to message processing, without enforcing any strict order in which messages must be processed. This characteristic is particularly beneficial in distributed systems where message delivery times can vary unpredictably. Actors can process incoming messages in different sequences, and the model does not guarantee that messages will arrive in the order they were sent. This flexibility allows for more efficient utilization of resources and can lead to more robust system designs that are tolerant of delays and variable message delivery times. Overall, the Actor model's emphasis on independent, concurrent actors and asynchronous communication makes it a powerful paradigm for building scalable and resilient concurrent and distributed systems.

4.2 Reactor Model

The Reactor model [16] shows great potential as an alternative to the actor model, enabling efficient and deterministic concurrency. The newly proposed Reactor model represents an advancement in deterministic reactive systems, providing a structured framework for creating complex, reactive RL applications. Central to this model are the concepts of "reactors" and "reactions." Reactors can be interpreted as deterministic actors, but instead of responding to messages, they react to discrete events, each linked to a specific logical time, denoted by a "tag." These events can trigger reactions within a reactor, similar to message handlers in actor systems, but with a key difference: reactions in reactors are governed by a defined order, ensuring determinism. Reactions share some similarities with message handlers in traditional actor models but differ in their triggering mechanism. Instead of messages, they are activated by discrete events, which can also be generated by reactions. Each event associates a value with a tag, representing its logical release time within the system. Reactions can access and modify state shared with other reactions in the same reactor, but interaction between different reactors is solely through events. This design choice illustrates the

model's deterministic nature, as the order of reaction executions is predictable and subject to strict constraints, such as tag order and execution order for reactions within the same reactor.

The Reactor model can be thought of as a "sparse synchronous model." This means that synchronous-reactive interactions at a particular logical time may be confined to isolated parts of the system. When a reaction executes, it exclusively accesses the reactor's state. Moreover, for reactions triggered by the same event (or events with the same tag), their execution order is predefined, reinforcing deterministic behavior. Reactors are composed of ports (inputs and outputs), hierarchy, and actions. The term "reactors" not only relate to actors in actor systems but also aligns with the synchronous reactive programming paradigm, prominent in languages like Esterel, Signal, and Lustre. Unlike traditional actors, reactors don't directly reference their peers. Instead, they use named and typed ports for interconnection, enabling a hierarchical design. This hierarchy, besides facilitating deterministic behavior, also serves as a scoping mechanism for ports and imposes constraints on connection types.

Additionally, reactors feature actions, a variant of ports used for scheduling future events within the same reactor or as a synchronization mechanism between internal logic and asynchronous external events. This design choice provides a bridge between deterministic internal logic and the nondeterministic external world, like sensor data, environment states, or network messages. The Reactor model also incorporates state variables within reactors. The shared resources, like replay buffers, model parameters, environment states in RL, are key motivators for grouping reactions in a single reactor. However, reactors themselves do not share state, ensuring isolation between reactors and allowing parallel execution of reactions in different reactors, unless a connection necessitates sequential execution.

Connections between reactors establish explicit communication channels. These connections reveal dependencies that are crucial for scheduling decisions honoring data dependencies. The Reactor model simplifies the declaration of these dependencies by breaking down functionality into reactions with well-defined lexical scopes, thus eliminating dependencies out of scope. The Reactor model's execution is governed by a run time environment, which adheres to a discrete-event model to ensure determinacy. This environment is responsible for maintaining a global event queue and a reaction queue, managing logical time, and executing reactions in a predetermined order. De-

tails regarding the scheduler are described in the next section.

4.3 Ray

Ray [15, 22] is an open-source framework designed for distributed computing and provides a simple, universal API for building distributed applications. Ray utilizes the Actor model, which is central to its architecture and operation, making it highly effective for concurrent and distributed computing tasks. The Actor model in Ray is used to encapsulate state and behavior, with actors being distributed across a node or a cluster and communicating through asynchronous message passing. Actors can be used for performance reasons (like caching soft state or ML models), or they could be used for managing long living connections to databases or to web sockets. They can maintain state across multiple tasks, which is particularly useful for applications that require managing large, mutable states, such as machine learning models or large datasets.

The local scheduler in Ray is a component of a worker node called Raylet. Raylet manages the worker processes and consists of two components, a task scheduler and an object store. The task scheduler takes care of scheduling and executing work on a node. It addresses issues such as a worker being busy, not having the proper resources to run a task, or not having the values it needs to run a given task. Expensive serialization and deserialization as well as data copying are a common performance bottleneck in distributed computing. Shared memory, specifically that which is managed directly by the operating system kernel, emerges as a superior mechanism compared to conventional approaches like socket connections. This kind of memory management boasts efficiency that transcends even memory-mapped file-based sharing. The fundamental attribute of distributed object store is that all objects within the store are immutable and retained in shared memory. The object store has its eviction policy, removing objects from the store or transferring them to other nodes when the allocated size limit is exceeded. This design choice ensures optimal access speeds, particularly when multiple workers on a singular node need to engage with the data. It's noteworthy that every node possesses an independent object store. Each node of a Ray is equipped with an object store, within that node's Raylet, and all objects stored collectively form the distributed object store of a cluster. The object store manages a shared pool of memory across workers on the same node. The object store is implemented in Plasma, which belongs to the Apache Arrow project. Functionally, the

object store takes care of memory management and ultimately makes sure workers have access to the objects they need in Ray.

4.4 Lingua Franca

Lingua Franca (LF) [18, 33] is a polyglot coordination language designed to facilitate the development of concurrent systems by focusing on deterministic interactions with their environment. It operates on the reactor model, where reactors are the fundamental units of composition, each encapsulating reactions to external stimuli. LF's main feature is its deterministic nature, meaning that given a set of inputs, a LF program will always produce the same outputs, greatly enhancing testability and efficiency. This determinism is achieved by using a superdense model of time (where events are ordered by time and microstep), ensuring causality and the absence of non-deterministic feedback loops in the reaction network.

The architecture of LF includes a compiler (lfc). The compiler process involves parsing and validating the LF code, checking for syntax errors, instantiation cycles, and cyclic dependencies among reactions. Valid code can then be transpiled into target code in languages like C, C++, TypeScript, and Python, which is then combined with a runtime system to manage the execution of reactors. LF also allows for graphical program representation, enhancing program structure understanding and error identification.

In LF, reactors can be defined with parameters (immutable after initialization), ports (for data input/output), actions (for scheduling internal events), and timers (for periodic events). Reactors can also declare state variables to maintain state across logical time. Reactions in LF, which contain the core logic, are defined with triggers (conditions under which they execute), sources (additional data inputs), and effects (outputs or actions they can trigger). LF supports reaction deadlines, where an alternative code block executes if a reaction misses its specified deadline, thus enforcing timing constraints.

LF allows for flexibility in connecting reactors through multiports (ports handling multiple data channels) and banks (multiple reactor instances). Connections can be logical, implying synchronization between ports, or physical, introducing intentional non-determinism for scenarios where strict event ordering is not required. LF's syntax permits the creation of complex interaction patterns among reactors, offering a robust toolset for constructing scalable and maintainable concurrent systems.
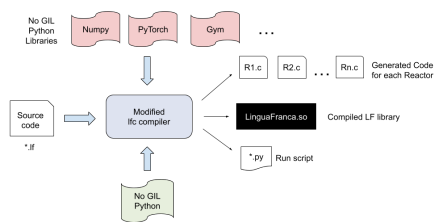

Figure 1: LFRL: compilation process

The formal semantics of LF are grounded in the theory of discrete-event systems, utilizing a generalized ultrametric space for modeling the behavior of LF programs. This approach guarantees that each LF program is deterministic and adheres to causality, essential for reliability in concurrent system design. LF's semantic model is fully abstract, providing both an operational perspective (how the program executes) and a denotational perspective (the meaning of program constructs), ensuring consistency.

We propose LFRL (Lingua Franca for Reinforcement Learning), a runtime implementation for Lingua Franca. Fig. 1 depicts the compilation process of the LFRL framework. This process begins with the source code written in Lingua Franca, with a .lf extension. The source code is then processed by a modified lfc compiler, tailored to work without Python's Global Interpreter Lock (GIL). Additionally, the compilation process involves various no-GIL versions of the Python libraries such as NumPy, PyTorch, Gym, etc.

This compiler generates C code for each reactor, with files named R1.c, R2.c, and so on as well as the compiled Lingua Franca library, LinguaFranca.so, which can be imported and used in Python scripts (*.py). The runtime implementation serves as a bridge between the high-level coordination language and the underlying Python environment without the GIL, enabling users to write truly concurrent Python programs and abstracts away the burden of coordinating worker threads.

4.5 Representing Distributed RL As Dataflow Graph

LFRL leverages the diagram synthesizer in LF, and provides a streamlined process to represent distributed RL as dataflow graphs [17]. By simply initializing reactors and setting their input and output ports, LF can automatically generate the corresponding dataflow graph. This diagrammatic feature is seamlessly integrated with Visual Studio extensions and Eclipse, offering a visual
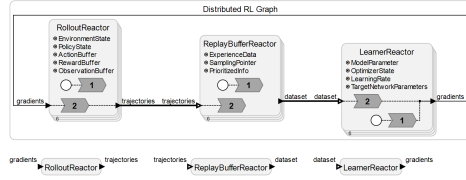
Figure 2: Generated Dataflow Graph for distributed RL

representation of the underlying RL processes. The resulting diagram (Fig. 2) clearly displays the reactors, ports, state variables, and their data dependencies, simplifying the understanding of the system's structure and aiding in the debugging of distributed RL systems. Users are therefore better positioned to manage the parallelization of RL processes. Detailed information about the implementation is provided in the appendix.

RolloutReactor: This reactor is pivotal in interacting with the environment. It gathers trajectories by executing a policy and recording the resulting states, rewards, and other pertinent outcomes. It receives gradients from the LearnerReactor for policy updates and sends experience (e.g. trajectories) to the ReplayBufferReactor.

- *EnvironmentState*: The present state of the environment.
- *PolicyState*: The internal state of the policy, significant for stateful policies such as those using recurrent neural networks.
- *ActionBuffer*: A temporary repository for actions decided by the policy.
- *RewardBuffer*: Temporary storage for rewards post-action.
- *ObservationBuffer*: Temporary storage for new environmental observations post-action.

ReplayBufferReactor: Acting as a centralized experience replay buffer, this reactor stores trajectories from the RolloutReactor for subsequent sampling. It provides sampled experience batches to the LearnerReactor for policy updates.

- *ExperienceData*: An accumulation of experiences (state, action, reward, subsequent state, and termination info).
- *SamplingPointer*: Indices or pointers facilitating efficient experience sampling.
- *PrioritizedInfo*: For prioritized replay buffers, additional data to manage experience priorities.

LearnerReactor: The LearnerReactor updates policies based on sampled experiences from the ReplayBufferReactor or directly from the RolloutReactor, and broadcast updated gradients to the RolloutReactor.

- *ModelParameter*: The neural network parameters, including weights and biases.
- *OptimizerState*: Elements related to the optimization process, such as momentum and learning rates.
- *LearningRate*: The current learning rate, either static or dynamically adjusted.
- *TargetNetworkParameters*: In algorithms like DQN [20], these are slowly updated parameters offering a stable learning target.

5 Optimizations for Distributed Reinforcement Learning

5.1 Scheduling Algorithm

The execution process of programs in the original LF involve a scheduler responsible for overseeing all scheduled future events, managing the logical time progression, and executing triggered reactions in the order dictated by the dependency graph. The scheduling mechanism in the original LF model, depicted in Fig. 3, operates with an event queue that strictly follows a tag order for processing upcoming events. Upon adding an event to the queue and processing it, the scheduler identifies triggered reactions, placing them in a reaction queue. These reactions are subsequently moved to a ready queue and executed by worker threads once all dependencies, as outlined by the Action-Port Graph (APG), are satisfied. Within this scheduling framework, the scheduler ensures a sequential execution of reactions, considering dependencies and aiming to maximize parallelism.

The scheduling mechanism is similar to Directed Acyclic Graph (DAG)-based strategies but differs by accommodating reactions within the Action-Port Graph (APG) that may not always need execution. Since the scheduler cannot predict in advance which reactions will be triggered at a given tag, it cannot precompute an optimal schedule. To address this, the scheduler assigns a level to each reaction, allowing reactions at the same level to execute concurrently. This heuristic eliminates the need for runtime APG analysis, streamlining the process. The scheduler processes reactions in a stepwise way, advancing to the next level only after completing all reactions at the current level. This approach substantially reduces synchronization overhead and contention on shared resources, contributing to its efficiency.

5.2 Scheduling Optimizations

The scheduling algorithm described in the above section is relatively straightforward to implement,
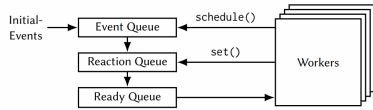
6

Figure 3: Scheduling mechanism in the LF runtime [18]

but achieving competitive performance requires additional optimizations. These optimizations, implemented in the C runtime in our LFRL, are detailed below.

Coordination of worker threads: Conceptually, the scheduler and workers are distinct; however, in practice, having a central scheduler with separate worker threads can cause significant synchronization overhead. To mitigate this, our implementation allows any worker thread to become the scheduler, allowing it to move ready reactions to the queue or advance to the logical time once all reactions are processed. Additionally, LFRL leverages the fact that we know the number of parallel reactions to execute from the APG, and thus can use a counting semaphore to regulate the number of active workers.

Lock-free data structure: The three queues (event, reaction, and ready queue) and other data structures are shared across all workers. Mutex-based synchronization would be inefficient due to high contention over these shared resources. Accordingly, LFRL adopts lock-free data structures wherever feasible. For example, the ready queue is a fixed-size buffer with an atomic counter, which corresponds to the maximum number of parallel reactions defined by the APG level. Each time a worker tries to execute a reaction it atomically decrements the counter. If the counter is negative, it indicates an empty queue, and the worker should proceed accordingly. Otherwise, the counter provides the index within the buffer from which to read. This operation is safe without additional synchronization, as all workers are waiting for upcoming new reactions.

5.3 Enabling Multi-threading for Reinforcement Learning

Parallel processing is a cornerstone in the field of machine learning, enabling the handling of computationally intensive tasks and large-scale data. The parallel execution models, namely multithreading and multiprocessing, provide different advantages. This section aims to dissect these models to guide the selection of an appropriate parallel execution strategy for RL.

Multithreading in RL:

1. Shared Memory Space: Multithreading involves threads operating within the same memory space, which facilitates faster and more efficient data sharing among threads compared to multiprocessing.
2. Resource Efficiency: The creation and management of threads consume fewer resources than processes. This efficiency stems from the shared memory space and the absence of a need for complex inter-process communication mechanisms.
3. I/O Bound Task Optimization: Multithreading proves advantageous in I/O-bound operations, where the ability to perform other tasks while waiting for I/O operations enhances efficiency.
4. Context Switching: The shared process and memory space of threads enable faster context switching than multiprocessing, as less information needs to be saved and restored.

Multiprocessing in RL

1. CPU Bound Task Optimization: Multiprocessing is typically more suitable for CPU-bound tasks, allowing for the distribution of tasks across multiple CPU cores. However, most of the CPU-bound tasks (e.g. gradient updates) in RL should be offloaded to GPU instead.
2. Fault Tolerance and Stability: The isolated nature of processes in multiprocessing ensures that a crash in one process does not impact others, thus enhancing application stability. The scheduler in LFRL guarantees the safety of threads and permits users to specify actions in the event of a failure.
3. Bypassing the GIL in Python: In Python, the Global Interpreter Lock (GIL) limits thread execution, multiprocessing provides a viable alternative for parallel CPU computations. This has been resolved by leveraging the No GIL version of Python.
4. Multi-Core Utilization: Multiprocessing enables the full utilization of multi-core processors by running separate processes on each core. However, it's important to consider that when sending excessively large objects, the processor may spend a significant amount of time on I/O overhead. LFRL, on the other hand, is able to bring threads to its full utilization

5.4 Optimizing Thread Allocations

The principle that running a task with multiple threads can speed up the process is generally true, as each thread can handle a portion of the work simultaneously. However, the scaling is not always linear due to various factors like thread allocation and CPU architecture. In our benchmarks,

the Intel CPU has a hybrid architecture, featuring a combination of performance and efficiency cores. Performance cores are designed for high-speed and intensive tasks, while efficiency cores are optimized for lower power consumption and handling background tasks. The original LF runtime randomly allocates threads for workers. As a result, when synchronous tasks are distributed across these cores, the overall speed is bounded by the slower efficiency cores. To achieve linear scaling, we have optimized the Reactor C runtime to prioritize using the performance cores before spawning threads.

6 Performance Comparison Between LFRL and Ray

To demonstrate the improved performance of LFRL over Ray for distributed RL workloads, we run the following experiments. We believe these experiments are comprehensive and adequate to support our assertion that our optimized implementation of the reactor model for RL significantly advances the state of the art for distributed RL.

1. We demonstrate lower overhead of broadcast and gather operations with LFRL than with Ray, and show that the overhead difference scales favorably for LFRL as we increase both the number of actors/reactors and the communication object size. While not an RL benchmark, this toy workload resembles the computation done in RL, and is useful for highlighting why exactly it is that LFRL outperforms Ray (sections 6.2, 6.3).

2. In most distributed RL settings, the majority of the parallelism that can be obtained in the data collection phase of the algorithm. As such, we extensively test parallel data collection in popular RL simulated environments with both LFRL and Ray (sections 6.4, 6.5).

3. To demonstrate that LFRL integrates seamlessly with other forms of parallel compute paradigms, namely GPU acceleration, and to demonstrate the versatility LFRL provides to implement various reinforcement learning algorithms, we implement and evaluate synchronized parallel Q-learning with deep neural networks (section 6.6).

4. Finally, we observe that multi-agent RL lends itself very well to being parallelized, as each agent can maintain its own learning and data collection actors/reactors. We evaluate in such a multi-agent RL (MARL) setting and observe favorable results for LFRL (section 6.7).

6.1 Experimental Setup

We perform all our experiments on an AWS EC2 m5.8xlarge instance, equipped with an Intel® Xeon® Platinum 8175M CPU @ 2.50GHz featuring 32 vCPUs. This setup includes 128 GiB of RAM and offers a 10 Gbps network bandwidth. The system runs on Ubuntu 20.04 and use Python 3.9.10, with NumPy version 1.22.3 and gym version 0.19.0. Benchmarks are available at this GitHub repository.
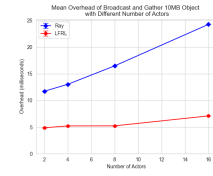
6.2 Number of Actors

Figure 4: Mean Overhead of Broadcast and Gather 10MB Object with Different Number of Actors using Ray and LFRL.
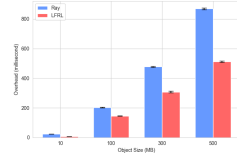

Figure 5: Mean Overhead of Broadcast and Gather on 16 actors with Different Object Sizes using Ray and LFRL.

Fig. 4 illustrates the mean overhead of broadcasting and gathering a 10MB object across different numbers of actors in a distributed RL setup. Two frameworks are compared: Ray and LFRL. The x-axis represents the number of actors, which are 2, 4, 8, and 16, while the y-axis shows the overhead in milliseconds. Both frameworks exhibit an increase in overhead as the number of actors grows, but Ray consistently has a higher overhead than LFRL. For instance, with 16 actors, Ray's overhead is close to 20 milliseconds, whereas LFRL's is just above 5 milliseconds.

6.3 Object Size

Fig. 5 presents a comparison of the mean overhead for broadcasting and gathering operations on objects of varying sizes using 16 actors, between Ray and LFRL frameworks, including a 99% confidence interval (CI). The x-axis displays the object size in megabytes (MB), ranging from 0 to 500 MB, while the y-axis indicates the overhead in milliseconds. From the graph, we can observe that as the object size increases, the overhead for both Ray and LFRL also increases. However, Ray's overhead grows at a higher rate than LFRL's. For instance, with the largest object size of 500 MB, Ray's overhead approaches 800 milliseconds, whereas LFRL's overhead is about half of that, around 400 milliseconds.

Both Fig. 4 and Fig.5 support the hypothesis that the reactor model employed by LFRL can outperform traditional actor models like Ray, particularly in scenarios with a greater number of actors and larger object size, due to optimizations such as reduced synchronization work and efficient multithreading with No GIL Python.
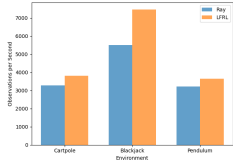

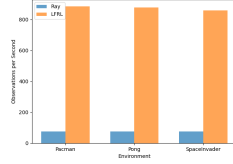Figure 6: Simulation Throughput of Ray and LFRL in Various Gym Environments.

Figure 7: Simulation Throughput of Ray and LFRL in Various Atari Environments.

6.4 Open AI Gym Environments

In Fig. 6, we see that our proposed LFRL outperforms Ray by 1.21x on average, with a particularly significant lead in the Blackjack environment. This suggests that LFRL is more efficient, especially in situations where there is a lower CPU demand for action updates within the environment. Considering the CPU architecture in m5x8.xlarge, which has a mix of 8 performance cores and 16 efficiency cores, the observed performance is not ideal due to the differing speeds of the cores and potential thermal throttling. The efficiency cores are slower and can bottleneck the system as the benchmark is limited by the slowest thread. Moreover, when multiple cores are active, the CPU may not run at its highest frequency due to thermal constraints. To optimize performance and achieve linear scaling, it is recommended to utilize only the 8 performance cores and disabled. This can be done by setting CPU affinity for performance cores, ensuring that CPU-bound tasks, which require synchronization, do not get restricted by the slower efficiency cores. By running tasks only on the performance cores, one can avoid the inconsistencies that come with hybrid architectures and get a more linear scaling with the number of actors or threads in use. It's important to note that vectorized environments are asynchronous and do not parallelize inference of policy. Therefore, it is not included in our benchmarks.

6.5 Atari Environments

In Fig. 7, we compare the performance of Ray and LFRL on Atari environments, specifically on Pacman, Pong, and SpaceInvader. Here, LFRL again significantly outperforms Ray in terms of

observations per second. On average, LFRL is roughly 11.62x faster than Ray across these environments. The substantial performance difference can be attributed to LFRL's efficiency in handling high I/O (input/output) bound tasks. Atari environments are more complex than the previously mentioned OpenAI gym environments. They represent each state as an 80x80 numpy array, which requires more computational resources to serialize and deserialize, especially when data needs to be sent over a network. Ray's use of pickle5 for serialization does help to increase throughput by efficiently serializing NumPy arrays, but it still introduces overhead during network transmission and the serialization/deserialization process. This overhead is particularly significant in environments where state updates are frequent and must be communicated quickly. LFRL's ability to handle the demands of complex simulation environments is thus a key advantage.
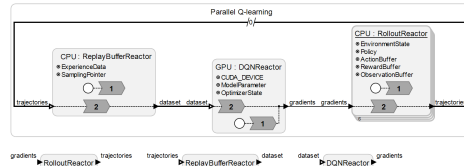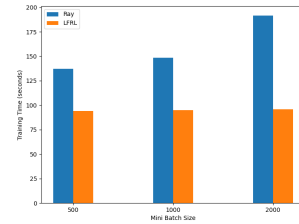

Figure 8: Dataflow Graph of Parallel Q-learning

Figure 9: Synchronized Parallel Q-learning with Different Batch Sizes from a Replay Buffer

6.6 Synchronized Parallel Q-learning

Deep Q-Networks (DQNs) [20] are an advancement in reinforcement learning that utilize deep neural networks to estimate Q-values. The Q-values are predictions of the expected discounted returns after taking certain actions given particular states in an environment. DQNs extend the capabilities of traditional Q-learning by handling larger state and action spaces, which are common in complex problems. These networks can scale up effectively with more data or increased model complexity; therefore, DQN usually utilizes GPU for gradient updates. However, since AWS EC2 m5.8xlarge doesn't include a GPU, benchmarking was conducted on a workstation with an Intel

9

i9-13950HX CPU @ 2.20GHz featuring 32 vCPUs, NVIDIA RTX4090, and 32 GiB of RAM. The system also runs on Ubuntu 20.04 and uses Python 3.9.10, with numpy 1.22.3, torch 1.9.0, and gym 0.20.0.

DQN has been implemented in a synchronized parallel Q-learning setup within a BlackJack environment [36]. The dataflow graph is shown in Fig. 8. The network is trained to take a blackjack hand as input and output scores for each of the possible actions in the game, which represent the expected rewards of taking those actions. The computational tasks are distributed with the rollout and replay buffer being executed on a CPU, while the DQN Reactor uses a GPU. This setup takes advantage of GPUs for complex matrix operations and multicore CPUs for sequential decision-making simulations in Open AI Gym environments.

The benchmark results, as seen in Fig. 9, demonstrate performance improvements using LFRL over Ray. In tests with a 500 sample batch size from the replay buffer, the average training time decreased by 31.2%. The data also shows that while Ray's training time increases with larger mini-batch sizes, LFRL's performance remains stable, indicating its ability to handle larger batches efficiently without significantly increasing resource utilization.
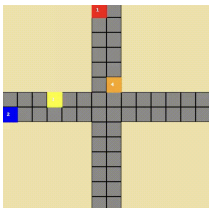

Figure 10: TrafficJunction4-v0 environment. The goal is to pass a four-way junction without any collisions.
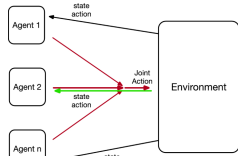

Figure 11: Multi-Agent Reinforcement Learning

6.7 Multi-Agent RL Inference Comparison

In Multi-Agent RL (MARL), multiple agents operate in a common environment, where each of them tries to optimize its own return by interacting with the environment and other agents [39] (See Fig. 11). In centralized MARL, a central controller aggregates information across the agents, including joint actions, rewards, and observations, and policies across different agents. In decentralized MARL, which is more common in coopera-

tive situations, each agent makes decisions based on its local observations [24].

We validate LFRL in a MARL setting with *magym* [14], a MARL library based on OpenAI Gym [5]. We use the TrafficJunction4-v0 environment, in which four agents are trying to pass a crossroad without crashing (see Fig. 10). This is a decentralized setting where each agent gets it's own local observation. See in Fig. 12, LFRL requires significantly less inference time than Ray (less than 1/2 the fitted line's slope). It is also noteworthy that as the number of agents increases as shown in Fig. 13, LFRL's inference time scales better (again, less than 1/2 the rate of increase than that of Ray). In TrafficJunction environments with 10 Agents, LFRL achieves a 5.12x speed up compared to Ray.
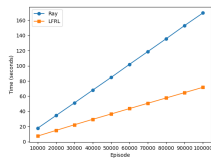

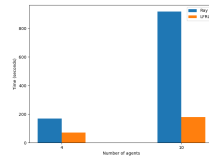Figure 12: Inference Time Between Ray and LFRL over Episode Counts

Figure 13: Inference Time Comparison between Ray and LFRL across Various Numbers of Agents

7 Discussion

In this study, we optimized a coordination language LF for RL applications. Our modifications, resulting in the so-called Lingua Franca for Reinforcement Learning (LFRL), allow it to outperform widely used frameworks like Ray in handling training, serving, and simulation tasks in RL. We achieve this by reducing the work needed for synchronization using Reactor model and decreasing the I/O overhead through optimzing the coordination of Python worker threads. Our empirical evaluations demonstrate LFRL's superior performance: a 1.21x and 11.62x faster sample generation in OpenAI Gym and Atari environments, a 31.2% reduction in average training time for synchronized parallel Q-learning, and a 5.12x acceleration in multi-agent RL inference. We aim to incorporate the optimizations into LFRL's federated execution, enabling efficient distributed training and serving across nodes and clusters. We also plan to delve deeper into the potential applications of our optimizations in deploying deep RL on embedded systems, and will compare it with frameworks such as the Robot Operations System, a middleware commonly used in robotics. With this work, we hope our important contributions allow for more efficient and scalable RL.

References

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)* (2016), pp. 265–283.

[2] AKKAYA, I., ANDRYCHOWICZ, M., CHOCIEJ, M., LITWIN, M., MCGREW, B., PETRON, A., PAINO, A., PLAPPERT, M., POWELL, G., RIBAS, R., ET AL. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113* (2019).

[3] ALI, H., ET AL. A view on deep reinforcement learning in system optimization. *arXiv preprint arXiv:1908.01275* (2019).

[4] BRADBURY, J., FROSTIG, R., HAWKINS, P., JOHNSON, M. J., LEARY, C., MACLAURIN, D., NECULA, G., PASZKE, A., VANDERPLAS, J., WANDERMAN-MILNE, S., ET AL. Jax: composable transformations of python+ numpy programs, 2018.

[5] BROCKMAN, G., CHEUNG, V., PETTERSSON, L., SCHNEIDER, J., SCHULMAN, J., TANG, J., AND ZAREMBA, W. Openai gym. *arXiv preprint arXiv:1606.01540* (2016).

[6] CHEN, L., LINGYS, J., CHEN, K., AND LIU, F. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 conference of the ACM special interest group on data communication* (2018), pp. 191–205.

[7] CHEN, T., LI, M., LI, Y., LIN, M., WANG, N., WANG, M., XIAO, T., XU, B., ZHANG, C., AND ZHANG, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[8] ESPEHOLT, L., MARINIER, R., STANCZYK, P., WANG, K., AND MICHALSKI, M. Seed rl: Scalable and efficient deep-rl with accelerated central inference. *arXiv preprint arXiv:1910.06591* (2019).

[9] 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* (2018), PMLR, pp. 1407–1416.

[10] HEWITT, C. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459* (2010).

[11] HORGAN, D., QUAN, J., BUDDEN, D., BARTH-MARON, G., HESSEL, M., VAN HASSELT, H., AND SILVER, D. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933* (2018).

[12] INCI, A., BOLOTIN, E., FU, Y., DALAL, G., MANNOR, S., NELLANS, D., AND MARCULESCU, D. The architectural implications of distributed reinforcement learning on cpu-gpu systems. *arXiv preprint arXiv:2012.04210* (2020).

[13] KAUFMANN, E., BAUERSFELD, L., LOQUERCIO, A., MÜLLER, M., KOLTUN, V., AND SCARAMUZZA, D. Championlevel drone racing using deep reinforcement learning. *Nature 620*, 7976 (2023), 982–987.

[14] KOUL, A. ma-gym: Collection of multi-agent environments based on openai gym. https://github.com/koulanurag/ma-gym, 2019.

[15] LIANG, E., WU, Z., LUO, M., MIKA, S., GONZALEZ, J. E., AND STOICA, I. Rllib flow: Distributed reinforcement learning is a dataflow problem. *Advances in Neural Information Processing Systems 34* (2021), 5506–5517.

[16] LOHSTROH, H. M. F. *Reactors: A deterministic model of concurrent computation for reactive systems*. University of California, Berkeley, 2020.

[17] LOHSTROH, M., MENARD, C., BATENI, S., AND LEE, E. A. Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS) 20*, 4 (2021), 1–27.

[18] MENARD, C., LOHSTROH, M., BATENI, S., CHORLIAN, M., DENG, A., DONOVAN, P., FOURNIER, C., LIN, S., SUCHERT, F., TANNEBERGER, T., ET AL. High-performance deterministic concurrency using lingua franca. *ACM Transactions on Architecture and Code Optimization 20*, 4 (2023), 1–29.

[19] MNIH, V., BADIA, A. P., MIRZA, M., GRAVES, A., LILLICRAP, T., HARLEY,

T., SILVER, D., AND KAVUKCUOGLU, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (2016), PMLR, pp. 1928–1937.

[20] MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., AND RIEDMILLER, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[21] MOERLAND, T. M., BROEKENS, J., PLAAT, A., JONKER, C. M., ET AL. Model-based reinforcement learning: A survey. *Foundations and Trends® in Machine Learning 16*, 1 (2023), 1–118.

[22] 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)* (2018), pp. 561–577.

[23] OPENAI. Gpt-4 technical report. *ArXiv abs/2303.08774* (2023).

[24] OROOJLOOY, A., AND HAJINEZHAD, D. A review of cooperative multi-agent deep reinforcement learning. *Applied Intelligence 53*, 11 (2023), 13677–13722.

[25] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in pytorch, 2017.

[26] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., ET AL. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[27] PUIG, X., UNDERSANDER, E., SZOT, A., COTE, M. D., PARTSEY, R., YANG, J., DESAI, R., CLEGG, A. W., HLAVAC, M., MIN, T., GERVET, T., VONDRUŠ, V., BERGES, V.-P., TURNER, J., MAKSYMETS, O., KIRA, Z., KALAKRISHNAN, M., MALIK, J., CHAPLOT, D. S., JAIN, U., BATRA, D., RAI, A., AND MOTTAGHI, R. Habitat 3.0: A co-habitat for humans, avatars and robots, 2023.

[28] RASLEY, J., RAJBHANDARI, S., RUWASE, O., AND HE, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (2020), pp. 3505–3506.

[29] RENTSCH, T. Object oriented programming. *ACM Sigplan Notices 17*, 9 (1982), 51–57.

[30] SAMSAMI, M. R., AND ALIMADAD, H. Distributed deep reinforcement learning: An overview. *arXiv preprint arXiv:2011.11012* (2020).

[31] SAVVA, M., KADIAN, A., MAKSYMETS, O., ZHAO, Y., WIJMANS, E., JAIN, B., STRAUB, J., LIU, J., KOLTUN, V., MALIK, J., PARIKH, D., AND BATRA, D. Habitat: A Platform for Embodied AI Research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (2019).

[32] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., RADFORD, A., AND KLIMOV, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[33] SCHULZ-ROSENGARTEN, A., VON HANXLEDEN, R., LOHSTROH, M., LEE, E. A., AND BATENI, S. Polyglot modal models through lingua franca. In *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*. IEEE, 2023, pp. 337–342.

[34] SHACKLETT, B., ROSENZWEIG, L. G., XIE, Z., SARKAR, B., SZOT, A., WIJMANS, E., KOLTUN, V., BATRA, D., AND FATAHALIAN, K. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Transactions on Graphics (TOG) 42*, 4 (2023), 1–13.

[35] SILVER, D., HUANG, A., MADDISON, C. J., GUEZ, A., SIFRE, L., VAN DEN DRIESSCHE, G., SCHRITTWIESER, J., ANTONOGLOU, I., PANNEERSHELVAM, V., LANCTOT, M., ET AL. Mastering the game of go with deep neural networks and tree search. *nature 529*, 7587 (2016), 484–489.

[36] SILVER, D., WIERSTRA, A. G. I. A. D., AND RIEDMILLER, M. Playing atari with deep reinforcement learning. *DeepMind Lab. arXiv 1312* (2013).

[37] Szot, A., Clegg, A., Undersander, E., Wijmans, E., Zhao, Y., Turner, J., Maestre, N., Mukadam, M., Chaplot, D., Maksymets, O., Gokaslan, A., Vondrus, V., Dharur, S., Meier, F., Galuba, W., Chang, A., Kira, Z., Koltun, V., Malik, J., Savva, M., and Batra, D. Habitat 2.0: Training home assistants to rearrange their habitat. In *Advances in Neural Information Processing Systems (NeurIPS)* (2021).

[38] Zeng, F., and Wang, C. Visual navigation with asynchronous proximal policy optimization in artificial agents. *Journal of Robotics 2020* (2020), 1–7.

[39] Zhang, K., Yang, Z., and Başar, T. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of reinforcement learning and control* (2021), 321–384.

8 Reactor Code Appendix

```
1  target Python {
2    threading: True
3  }
4
5  preamble {=
6      #Import packages
7      #e.g Torch, Gym, Numpy
8  =}
9
10 reactor RolloutReactor {
11   input[6] gradients
12   output[6] trajectories
13
14   state EnvironmentState
15   state PolicyState
16   state ActionBuffer
17   state RewardBuffer
18   state ObservationBuffer
19
20   reaction(startup) {=
21     #Initialize Environment
22   =}
23
24   reaction(gradients) ->
       trajectories {=
25     #Perform rollouts for the
       Environment
26   =}
27 }
28
29 reactor ReplayBufferReactor {
30   input[6] trajectories
31   output[6] dataset
32
33   state ExperienceData
34   state SamplingPointer
35   state PrioritizedInfo
36
37   reaction(startup) {=
38     #Initialize ReplayBuffer
39   =}
40
41   reaction(trajectories) ->
       gradients {=
42     #Append Trajectories into
       ReplayBuffer
43   =}
44 }
45
46 reactor LearnerReactor {
47   output[6] gradients
48   input[6] dataset
49
50   state ModelParameter
51   state OptimizerState
52   state LearningRate
53   state TargetNetworkParameters
54
55   reaction(startup) -> gradients
       {=
56     # Initialize the policy
57   =}
58
59   reaction(dataset) -> gradients
       {=
60     # Update the policy
61   =}
62 }
63
64 main reactor {
65   rollout = new[6] RolloutReactor
       ()
66   replay = new[6]
       ReplayBufferReactor()
67   learner = new[6] LearnerReactor
       ()
68
69   # Specifiy
70   (learner.gradients)+ -> rollout.
       gradients
71   (rollout.trajectories)+ ->
       replay.trajectories
72   (replay.dataset)+ -> learner.
       dataset
73 }
```