Stellaris: Staleness-Aware Distributed Reinforcement Learning with Serverless Computing

Hanfei Yu
Stevens Institute of Technology
Hoboken, NJ, USA
hyu42@stevens.edu

Hao Wang
Stevens Institute of Technology
Hoboken, NJ, USA
hwang9@stevens.edu

Devesh Tiwari
Northeastern University
Boston, MA, USA
d.tiwari@northeastern.edu

Jian Li
Stony Brook University
Stony Brook, NY, USA
jian.li.3@stonybrook.edu

Seung-Jong Park
Missouri University of Science and Technology
Rolla, MO, USA
seung-jong.park@mst.edu

***Abstract*—Deep reinforcement learning (DRL) has achieved remarkable success in diverse areas, including gaming AI, scientific simulations, and large-scale (HPC) system scheduling. DRL training, which involves a trial-and-error process, demands considerable time and computational resources. To overcome this challenge, distributed DRL algorithms and frameworks have been developed to expedite training by leveraging large-scale resources. However, existing distributed DRL solutions rely on synchronous learning with serverful infrastructures, suffering from low training efficiency and overwhelming training costs.**

This paper proposes *Stellaris*, the first to introduce a generic asynchronous learning paradigm for distributed DRL training with serverless computing. We devise an importance sampling truncation technique to stabilize DRL training and develop a staleness-aware gradient aggregation method tailored to the dynamic staleness in asynchronous serverless DRL training. Experiments on AWS EC2 regular testbeds and HPC clusters show that *Stellaris* outperforms existing state-of-the-art DRL baselines by achieving 2.2× higher rewards (*i.e.*, training quality) and reducing 41% training costs.

I. Introduction

The deep reinforcement learning (DRL) demonstrates notable success across diverse domains such as gaming AI [9, 39, 69, 74], robotics [12, 86], scientific simulations [33, 68, 76], and large-scale (HPC) system scheduling [49, 56, 80, 81]. The training process of a DRL agent entails iterative experimentation, demanding significant time and computational resources. Consequently, distributed DRL algorithms and infrastructures have been proposed extensively to expedite training through the utilization of large-scale resources. Common routines for reducing the operational cost of distributed training infrastructure include offloading training tasks to HPC clusters [3, 40] and renting virtual machine (VM) servers from Infrastructure-as-a-Service (IaaS) cloud providers [16, 45].

Despite the immense potential of DRL, existing large-scale clusters often suffer from low efficiency and sub-optimal resource utilization when running distributed DRL algorithms. We identify three main pain points hindering the effectiveness of DRL in current large-scale distributed training. *First*, the timing of policy updates poses a significant challenge due to the dynamic staleness inherent in the distributed setting, leading to slowed training and decelerated convergence [15, 82, 84]. *Second*, distributed DRL tasks exhibit dynamic demands for volumes of computing resources. Determining the appropriate system resources to allocate for distributed reinforcement learning (RL) tasks remains a complex issue, thus often resulting in low cost-efficiency and sub-optimal resource utilization [46]. *Third*, distributed DRL demands dynamic volumes of trajectory data during training [79] while existing fixed trajectory sampling strategies cannot meet the requirements, further impeding the training process.

Framework	Async. Learners	Scalable Actors	On-&Off-policy	Server-less
Ray RLlib [43]	×	×	✓	×
MSRL [85]	×	×	✓	×
SEED RL [17]	×	×	✓	×
SRL [46]	×	×	✓	×
PQL [41]	×	×	×	×
MinionsRL [79]	×	✓	×	✓
Stellaris+∗	✓	✓	✓	✓

TABLE I: Summary of DRL training frameworks. ∗ indicates integration with *Stellaris*.

Existing solutions for distributed DRL training largely rely on the Actor-learner architecture [16, 17, 45, 46, 79], where actors interact with the environment to collect trajectories (training data) and learners train an optimal policy based on the data to maximize rewards. However, they fail to fill the above gaps due to missing necessary features. We are the first to design an efficient and robust asynchronous learning paradigm with serverless computing to address the ignored gaps in distributed DRL training.

In this paper, we propose *Stellaris*, a generic asynchronous learning paradigm to accelerate DRL training with serverless computing. Table I compares the features of *Stellaris* with state-of-the-art solutions. *Stellaris* supports both serverful and serverless training infrastructures as well as on- and off-

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

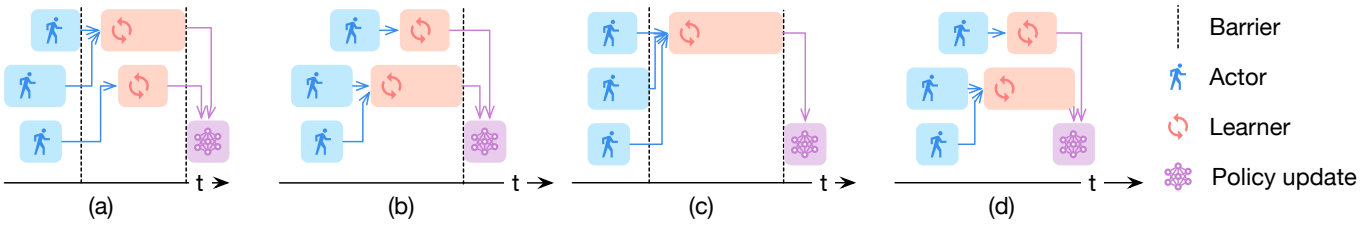

Fig. 1: Actor-learner architectures for distributed DRL training solutions, including (a) synchronous actors + synchronous learners, (b) asynchronous actors + synchronous learners, (c) synchronous actors + single learner, and (d) our *Stellaris*: synchronous/asynchronous actors + asynchronous learners.

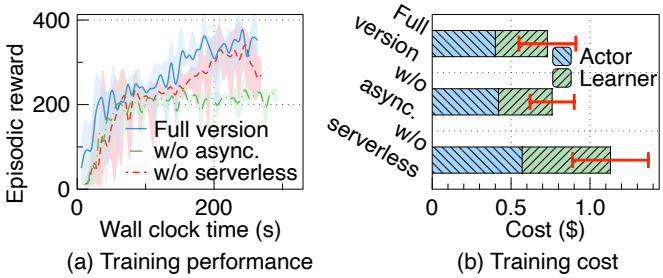

Fig. 2: Training performance (a) and cost (b) of running asynchronous serverless DRL training.

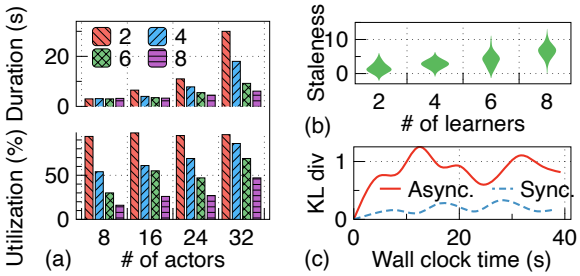

Fig. 3: Characterizations of asynchronous learners in serverless DRL training, including (a) dynamic learner orchestration, (b) dynamic staleness, and (c) unstable policy updates.

policy DRL algorithms. We leverage the agile scalability and fine-grained allocation of serverless computing to meet the dynamic and transient resource demands in distributed DRL. To accommodate the dynamic volumes of trajectory data, we devise a novel importance sampling truncation technique to stabilize the training process. We also develop a staleness-aware gradient aggregation method tailored to addressing the dynamic staleness in distributed DRL training. The main contributions of this paper are summarized as follows:

- We design *Stellaris*, a generic asynchronous learning paradigm in serverless computing architecture for accelerating distributed DRL training.
- We devise an importance sampling truncation technique to stabilize DRL training in the asynchronous learning setting.
- We develop a staleness-aware gradient aggregation method to trade-off between policy update speed and convergence, hence accelerating the training process.
- We evaluate the *Stellaris* prototype by integrating with state-of-the-art (SOTA) DRL algorithms and frameworks on AWS EC2 regular testbeds and HPC clusters with 16 GPUs and 960 CPU cores. Extensive experiments with Mujoco and Atari benchmarks show that *Stellaris* improves the final reward (*i.e.*, training quality) by up to 2.2× and reduces the training cost by up to 41%.

II. Background and Motivation

A. Distributed DRL

DRL aims to optimize a policy π parameterized with θ by maximizing the expected return. The DRL agent learns to maximize the cumulative reward $J(\pi) := \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T}\gamma^t r_t\right]$, where τ is a trajectory, r_t is the reward at timestep t, and γ is the discount factor. The policy π is updated via policy gradients [72], $\nabla_\theta J(\pi_\theta) := \mathbb{E}_t\left[\nabla_\theta \log \pi_\theta(a_t|s_t) A_t(s_t|a_t)\right]$, where A_t is the advantage function.

Actor-learner architectures are one of the most performant and efficient large-scale approaches to enable distributed DRL training [16, 17, 24, 27, 35, 45, 77]. Fig. 1 illustrates the actor-learner architecture for existing DRL training solutions. In actor-learner, an agent is divided into two sub-modules, *i.e.*, one *learner* and multiple *actors*. Each training *round* consists of two steps: 1) each actor interacts with a copy of the same environment under the guidance of a policy and submits the sampled data to the learner, and 2) the centralized learner computes gradients using the sampled data, updates its policy, and synchronizes the new policy to multiple actors. Though training with synchronous actors is more stable [20], recent solutions have shifted to asynchronous actors due to higher sampling efficiency [16, 17, 45]. However, serverful distributed DRL solutions—training with virtual or physical servers—are expensive due to unavoidable idle resources and low resource utilization [79].

B. Serverless DRL Training

Emerged as a new cloud computing diagram, serverless computing has been wildly adopted for designing cost-efficient distributed Machine Learning (ML) training systems [10, 21, 23]. A recent work, MinionsRL [79], first attempted to build a serverless DRL training framework by packaging the

learner and actors as serverless functions. Though MinionsRL provides cheaper and faster training with dynamic scaling using serverless actors, the centralized learner becomes a critical bottleneck of large-scale DRL training for two reasons: 1) MinionsRL only considers single-learner training, which unavoidably slows down the gradient computation and policy update for a large number of trajectories, and 2) MinionsRL is designed for training synchronous DRL algorithms only, which cannot be directly applied to asynchronous DRL training.

C. Benefits of Asynchronous Serverless Learners

Asynchronous learning in distributed ML training has been extensively studied [25, 83, 84]. Due to the instability of DRL training, existing distributed DRL solutions either employ synchronous learners [43, 46, 85] (Fig. 1(a) and (b)) or even use centralized learner [16, 17, 45, 79] (Fig. 1(c)) for stable training. Asynchrony in distributed DRL usually limits to actors, which has been focused by many works [16, 17, 45, 50, 64]. However, we argue that *asynchronous learners* can significantly benefit serverless DRL training with a realistic example. We run the popular Proximal Policy Optimization (PPO) algorithms [66] with Ray RLlib [43] in the MuJoCo [73] Hopper environment. To show the benefits of asynchronous learning and serverless computing, we disable each component and create two variants for comparison. Each baseline is configured with 128 actors and four learners (experimental setup in §VIII-A), where each actor samples 1,024 timesteps and the original synchronous learners are replaced with asynchronous ones for comparison. Fig. 2(a) and (b) show the episodic rewards and training costs of running the baselines. The results demonstrate that asynchronous learning and serverless computing jointly improve distributed DRL training to deliver superior training performance while reducing training costs.

D. Barriers for Asynchronous Serverless Learners

Despite the potential merits, directly shifting existing serverful *multi-learners* schemes or *asynchronous learning* to serverless DRL training raises three challenging problems:

Dynamic learner orchestration. Multi-learners are commonly supported and implemented by popular DRL algorithms [16, 45] and DRL training frameworks [26, 43, 85], due to the need to utilize multiple GPUs for data parallelism. However, existing multi-learner schemes all assume a fixed number of serverful learners pre-allocated before the training begins. Due to dynamic actor scaling, simply integrating serverful multi-learner schemes into serverless DRL training cannot achieve both fast learning time and high GPU utilization at the same time. Fig. 3(a) shows the total learning time and GPU utilization of different numbers of learners (2, 4, 6, and 8) and actors (8, 16, 24, and 32) for running PPO in the Hopper environment. Increasing learners reduces the total learning time at large volumes of actors while wasting GPU resources at small volumes. Therefore, multi-learner allocation should be scalable and dynamic to achieve efficient learning for serverless DRL training.

Dynamic staleness. When multiple learners compute gradients for policy updates asynchronously, some gradients can be stale due to missing policy aggregation, which are picked up in the later rounds. Stale gradients are proven to damage the training performance, prolong the convergence rate, and even diverge the learning process [15]. A line of research [25, 83, 84] proposes to bound the staleness in such asynchronous learning to trade-off between synchronization delay and learning efficiency. However, existing approaches assume that the number of workers (*i.e.*, learners) is fixed, leading to sub-optimal staleness bounds in the dynamic learner setting. Fig. 3(b) shows the probability density function (PDF) of staleness values that occurred when running PPO with different numbers of asynchronous learners in the Hopper environment. The staleness values gradually increase as the number of learners grows. Based on the observation, staleness bounds should be adaptive for an optimal dynamic learner design.

Unstable policy updates. Importance sampling is commonly employed to stabilize policy updates in DRL algorithms [16, 45, 66]. Existing synchronous learner schemes copy the same policy to each learner before computing gradients, where policy updates are clipped based on the importance sampling ratio between the policy and trajectories sampled by actors. However, each learner holds a unique policy and computes gradients individually in the asynchronous learner setting. Existing importance sampling methods fail to mitigate unstable policy updates in asynchronous learner policy aggregation. Fig. 3(c) shows the Kullback–Leibler (KL) divergence between policies searched iteratively when running PPO in the Hopper environment. Intuitively, a larger KL divergence between two policies indicates a wilder update. Asynchronous learners introduce significantly larger policy updates than synchronous learners, which are generally considered to be the reason for drastic performance drops in DRL training [34, 71]. To integrate asynchronous learners into serverless DRL training, we must restrict large policy updates brought by asynchrony to deliver stable training.

III. Objectives and Challenges

Based on the analysis and observations from §II, we carefully craft *Stellaris* to achieve three goals:

Accelerating distributed DRL. We aim to improve the efficiency and effectiveness of training algorithms for DRL tasks. We design novel asynchronous learning techniques in *Stellaris* to achieve faster convergence rates with minimal modifications on DRL algorithm deployment.

Stabilizing DRL training. Asynchronous distributed DRL training involves multiple learners, each with a unique policy. Sampling trajectories for asynchronous learners introduces the cross-learner policy drift (§V-A), leading to unstable training, performance variance, and undesired increases in training costs. Hence, we must stabilize training in *Stellaris* to deliver robust training performance and cost.

Optimizing cost-efficiency. Unlike ML, distributed DRL training requires recurrent interactions between learners, actors, and RL environments, often leading to numerous trial-and-error

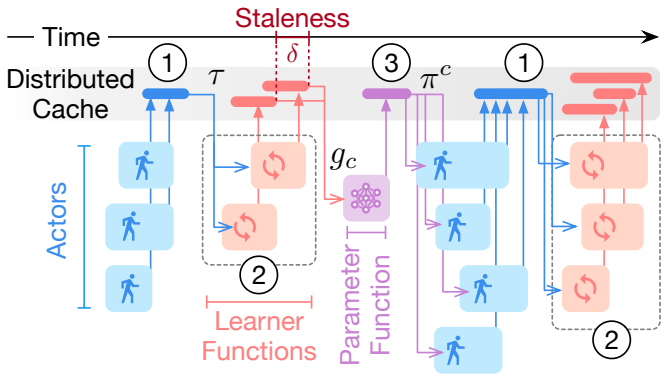

Fig. 4: *Stellaris*'s workflow.

and high training cost [21, 79]. Utilizing serverless computing immensely enlarges the design space of learner orchestration for optimizing cost-efficiency. *Stellaris* should provide cost-efficient learner function management while maintaining high-performant training.

We must address the three following challenges to achieve the above objectives:

How to deal with staleness in asynchronous distributed training with serverless learners? Staleness in traditional asynchronous parameter-server learning [25, 83, 84] assumes a fixed number of participants (*i.e.*, learners). However, *Stellaris*'s serverless learners are event-driven and on-demand, spawned and terminated by needs. Gradient aggregation and policy update in *Stellaris* can involve a varying number of learners, thus leaving a chaotic environment with dynamic staleness that extensively slows down the training process.

How to stabilize policy updates with multiple asynchronous learners? In *Stellaris*, each learner keeps a unique policy and asynchronously computes gradients. Though learner policies clip their importance sampling ratios and guarantee conservative updates locally, they lack a global view to prevent *cross-learner divergence* in the aggregation phase, leading to unstable training performance.

How to orchestrate serverless learners to achieve minimal training cost while preserving training performance? Introducing serverless computing to distributed DRL extensively complexifies the design space of asynchronous learning in *Stellaris*. We should carefully orchestrate serverless functions to achieve minimal training cost and high resource utilization while maintaining training performance.

IV. System Overview

Stellaris is a generic asynchronous learning paradigm for accelerating DRL training with serverless computing. Fig. 4 shows *Stellaris*'s workflow. *Stellaris* has four major components to enable efficient asynchronous DRL with serverless computing: 1) **Parameter Function** controls the staleness and aggregates the gradients with staleness-awareness, 2) **Learner Function** computes the gradients based on the sampled trajectory batch, 3) **Actors** interact with RL environments to collect trajectories for training, and 4) **Distributed Cache** (referred as "the Cache") is an in-memory key-value data buffer (*e.g.*, Redis) that stores the intermediate results from previous components, such as trajectories, gradients, and policy models.

We introduce *Stellaris*'s workflow in three steps:

Step (1): Importance sampling driven trajectory collection. In DRL, trajectories are the training data collected from the actor-environment interactions for training and updating the RL policy. Before sampling, each actor pulls the latest policy model from the Cache to update its own policy. After the actor policy is updated, the actor uses the new policy to interact with an RL environment and sample trajectories from the policy. Once complete, each actor submits trajectories as sample batches back to the Cache for input to learner functions. *Stellaris* supports both serverful and serverless actors for trajectory collection. However, actor policies may not align with learner policies in the asynchronous setting, multiple learners working asynchronously further exacerbate the policy drift issue [16, 45]. We leverage the importance sampling technique to prioritize trajectories based on actor-learner policy differences and truncate the importance sampling ratios to address the cross-learner policy drift (§V-A).

Step (2): On-demand gradient calculation. *Stellaris* invokes an appropriate number of serverless learner functions according to the demands to calculate gradients cost-efficiently. Whenever there are new sample batches available in the Cache, *Stellaris* invokes a set of learner functions concurrently to learn from the trajectories and produce gradients. The function inputs are the IDs of trajectory samples and policies, specifying the keys to query corresponding sample trajectory batches and policy models from the Cache. Each learner function pulls the latest policy model, fetches the sample batch for gradient calculation, and submits computed gradients back to the Cache for aggregation. Unlike synchronous learners in existing distributed DRL solutions [16, 26, 43, 79, 85], *Stellaris*'s event-driven serverless learners are naturally suitable for asynchronous learning.

Step (3): Staleness-aware gradient aggregation. We design a serverless Parameter Function to achieve staleness-aware gradient aggregation. *Stellaris* determines whether to invoke the Parameter Function based on the trade-off between policy update speed and convergence rate. We devise a gradient aggregation rule that makes adaptive aggregation decisions to balance between the two goals (§V-C). Upon invocation, we feed the gradient IDs as input to the Parameter Function so that it can retrieve the gradient values from the Cache. Then, the Parameter Function calls the backward propagation to update a new policy and sends the new policy model to the Cache for future learners and actors to use.

V. *Stellaris*'s Design

A. Importance Sampling Truncated Trajectory Processing

Suffering from high variance and large policy updates [19], DRL training often experiences sudden performance drops, leading to unstable training performance. Differences between two policies $\pi_{\theta'}$ and π_θ are commonly measured using

importance sampling ratio [32, 48, 66, 78]. Recall that $\pi(a|s)$ is the probability of taking action a in state s via policy π. The importance sampling ratio R is defined as the difference of action distributions generated by two policies given the same states:

$$R := \frac{\pi_{\theta'}(a|s)}{\pi_{\theta}(a|s)}, \tag{1}$$

which we abbreviate as $\frac{\pi_{\theta'}}{\pi_{\theta}}$ for simplicity. Intuitively, a larger importance sampling ratio indicates two policies conflict more with each other. The importance sampling ratio can potentially explode if a newly updated policy $\pi_{\theta'}$ is too far from the previous one π_{θ}, which is shown to result in drastic performance drop in DRL training both empirically [48, 63, 66] and theoretically [34, 71]. To alleviate the unstable training, existing DRL methods [16, 45, 48, 63, 66, 78] wildly applied importance sampling [30] to reduce variance and restrict policy updates.

Though existing importance sampling methods effectively restrict the policy updates for synchronous learners, they fail to stabilize the training in *Stellaris*'s asynchronous multi-learner setting. Let $\{\pi_{\theta_1}, \ldots, \pi_{\theta_n}\}$ denote the group of n learner policies that participate in the gradient aggregation phase and μ_{θ} denote the actor policy that samples trajectories for updating policies. Existing importance sampling methods [16, 45, 66] in DRL assume all learners synchronize their policies (*i.e.*, $\pi_{\theta} = \pi_{\theta_1} = \ldots = \pi_{\theta_n}$) before computing gradients. In this case, one only needs to bound the importance sampling ratio between learner and actor policies (*i.e.*, $\left|\frac{\pi_{\theta}}{\mu_{\theta}}\right|$), to prevent actors from sampling out of learners' policy distribution. However, in *Stellaris*, each learner may hold a unique policy (*i.e.*, $\pi_{\theta_1} \neq \ldots \neq \pi_{\theta_n}$) due to asynchronous learning. When the i^{th} learner only calculates its local importance sampling ratio $\frac{\pi_{\theta_i}}{\mu_{\theta}}$, any unbounded cross-learner importance sampling ratios within $\{\frac{\pi_{\theta_1}}{\mu_{\theta}}, \ldots, \frac{\pi_{\theta_{i-1}}}{\mu_{\theta}}, \frac{\pi_{\theta_{i+1}}}{\mu_{\theta}}, \ldots, \frac{\pi_{\theta_n}}{\mu_{\theta}}\}$ can still risk being explosive and thus incurring policy drift.

Fig. 5(a) shows an example of cross-learner policy drift. When Learner 2 computes its local importance sampling ratio $\frac{\pi_2}{\pi_1}$, it ignores the other learner (Learner 2) that also contributes to the policy update. As a result, $\frac{\pi_3}{\pi_1}$ may still explode and hinder the training if no cross-learner truncation is applied.

Therefore, we propose to truncate the importance sampling ratios of asynchronous learners with a *global view* during *Stellaris*'s policy aggregation. This global importance sampling truncation further reduces the variance and enforces reasonable policy updates. Specifically, when computing gradients using trajectories sampled by some actor policy μ_{θ}, the global importance sampling ratio R' should be truncated by the minimal learner-actor policy ratio observed during the aggregation phase:

$$R' := \min(\left|\min_i(\frac{\pi_{\theta_i}}{\mu_{\theta}})\right|, \rho), \ i \in \{1, \ldots, n\}, \tag{2}$$

where ρ is the clip threshold similar to existing importance sampling clipping methods [16, 45]. Intuitively, when any cross-learner ratio $\frac{\pi_{\theta_i}}{\mu_{\theta}}$ action distribution changes significantly, we can risk sampling trajectories outside that distribution. The truncation in Eq. (2) pulls the large importance sampling ratio back to ρ. Hence, we can easily integrate existing importance sampling-clipped policy gradient objectives [32] with our truncation, which is extended as

$$\nabla J(\pi_{\theta}) = \mathbb{E}_t\Big[\mathbb{E}_{\tau_t \sim \mu_{\theta}}\big[\min(\left|\min_i(\frac{\pi_{\theta_i}}{\mu_{\theta}})\right|, \rho) A_t\big]\Big].$$

We further prove that importance sampling truncation in Eq. (2) holds a lower bound on the monotonic reward improvement (§VI-B), which theoretically guarantees *Stellaris*'s training performance gains.

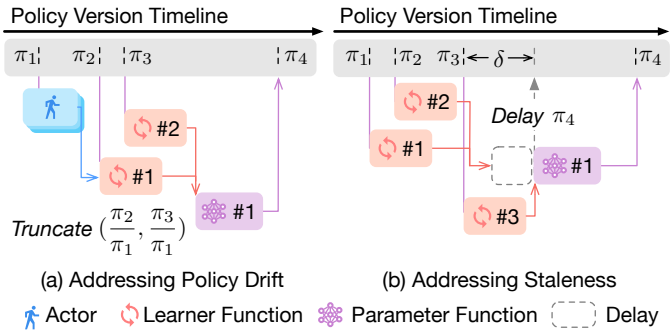

Fig. 5: (a) Importance sampling truncation. (b) Staleness-aware gradient aggregation.

B. On-Demand Gradient Calculation with Serverless Learners

Stellaris manages the learner functions in an extremely dynamic environment and must address three problems that hinder learning efficiency: 1) Data transmission overhead between CPUs and GPUs, 2) Launch and terminate learner functions on-demand, and 3) Data passing overhead between serverless learners. We carefully design the following components of *Stellaris* to achieve efficient learner orchestration by tackling the three problems.

GPU data loader is a lightweight daemon program that pre-loads trajectories from the Distributed Cache to GPUs. Similar to serverless function pre-warming [22, 61, 62, 67] that decouples the loading and execution of function codes, our data loader also decouples the trajectory loading and learning of learners. The data loader monitors trajectories in the Cache, batches accumulated trajectories, and pre-loads the batches to GPU memory. After the trajectories are loaded in GPUs, we export a pointer that points to the GPU memory slot for learners' fast access. We use Python multi-threading to parallelize the long-running data loader to improve loading throughput.

Learner schemes determine when to invoke a learner function and how they transmit data in *Stellaris*. When the data loader indicates available trajectories, we immediately initialize a learner function by passing two parameters: the latest policy weights of the parameter learner and trajectory pointers produced by the data loader. The learner function then initializes the policy model to compute gradients on the given trajectories in GPU memory. When a learner function completes the gradient computation, it submits the gradients

back to the Cache and then terminates if no more trajectory pointers are provided.

Hierarchical data passing employs three communication methods to achieve efficient messaging in *Stellaris*: 1) *shared memory* for learner functions that are located in the same physical server to exchange gradients and access trajectories, 2) *remote procedure call (RPC)* for learners' remote communication, and 3) *Distributed Cache* as external storage for persisting trajectories. We combine these three techniques to jointly improve *Stellaris*'s communication efficiency.

C. Staleness-Aware Gradient Aggregation

We propose a staleness-aware gradient aggregation method to control the dynamic staleness in the serverless learning environment. Unlike the immediate aggregation in traditional asynchronous learning, *Stellaris* delays the gradient aggregation to enforce a strictly controlled bound on staleness. Specifically, *Stellaris* keeps monitoring the Cache for any newly computed gradients. Whenever new gradients are available in the Cache, *Stellaris* fetches the gradients to its local gradient queue while delaying the aggregation. Every gradient enqueue triggers *Stellaris* to evaluate the average staleness $\bar{\delta}$ of the queue. We allow *Stellaris* to aggregate the gradients from its queue only if the average staleness is below a certain threshold β.

Fig. 5(b) shows the staleness-aware gradient aggregation in *Stellaris*. If the staleness of current gradients (submitted by Learners 1 and 2) is over a certain threshold, *Stellaris* delays the next gradient aggregation until new gradients (from Learner 3) arrive to decrease the average staleness below the threshold.

Instead of enforcing a static bound on staleness as existing methods [25, 82], we propose to dynamically configure *Stellaris*'s staleness threshold in real-time. Concretely, we temporarily disable the threshold at the first training round to obtain the maximum staleness $\delta_{\max}$ in a pure asynchronous environment. For the following training rounds, we set the staleness threshold β_k at round k given by

$$\beta_k := \delta_{\max} \times d^k, \ d \in (0, 1], \tag{3}$$

where d is the exponential factor that decays through training rounds. Intuitively, *Stellaris* relaxes the staleness threshold in the early rounds to obtain more gradient computations, because the gradients can be admitted to aggregate frequently, hence accelerating the training. In the later rounds, *Stellaris* gradually narrows the staleness threshold to slow down gradient aggregation and enforce less staled policy updates for stable convergence and better generalization. Note that the decay factor d is a knob for adjusting between synchronous and asynchronous learning. $d = 0$ removes the threshold and forces the parameter function to synchronize all learners, whereas $d = 1$ allows a pure asynchronous setting.

We also leverage the staleness $\delta_{n,c}$ to modulate the learning rate α per-gradient basis. For a gradient g_c with staleness δ_c, the learning rate α_c tailored to g_c is given by

$$\alpha_c := \frac{\alpha_0}{\sqrt[v]{\delta_c}}, \text{ if } \delta_c > 0, \tag{4}$$

where α_0 is the original learning rate obtained from the optimizer (*e.g.*, SGD, Adam, or RMSProp) and $v \in \mathbb{Z}_{\geq 0}$ is a non-negative root factor that avoids diminishing policy updates. Putting Eq. (3) and Eq. (4) together, we can summarize *Stellaris*'s gradient aggregation and policy update as

$$g_c := \frac{1}{H_c} \sum_{i=1}^{H_c} \frac{\alpha_0}{\sqrt[v]{\delta_j}} g_{i,j}, \ \theta_{c+1} := \theta_c - g_c,$$

where $j \in [0, c]$ is the clock of gradient $g_{i,j}$ within H_c. We further theoretically show that *Stellaris*'s gradient aggregation does not affect the convergence property of DRL algorithms with existing optimizers (§VI-A).

VI. Analysis

A. Convergence Analysis

We show that *Stellaris*'s gradient aggregation does not affect the convergence property of existing optimizers using the SGD optimizer as an example. We prove that *Stellaris* achieves a convergence rate of $\mathcal{O}(\frac{1}{\sqrt{T}})$, where T is the total number of policy update steps, that recovers the original convergence rate of SGD optimizer [42, 82].

Theorem 1. *Let C_1 and C_2 be certain positive constants depending on the objective $J(\theta)$. With common assumptions (unbiased gradients, bounded variance, and Lipschitz-smooth) [15, 37, 42, 84], we achieve a convergence rate of*

$$\frac{1}{T} \sum_{t=1}^{T} \mathbb{E}(||\nabla J(\theta_t)||^2) \leq 2\sqrt{\frac{2C_1C_2}{Tb}},$$

where b is the mini-batch size used by each learner function to produce gradients.

We import Theorem 1 from [82] with some simplifications, which gives the original convergence rate with arbitrary staleness. *Stellaris* enforces a strict threshold on the mean of staleness β_k at any round k. Thus, we can take the bounded average staleness as a constant and reduce the convergence rate to an order of $\mathcal{O}(\frac{1}{\sqrt{Tb}})$ to achieve a near-linear speedup.

B. Reward Improvement Lower Bound

We prove that *Stellaris*'s importance sampling truncation has a lower bound on monotonic reward improvement. Recall that $J(\pi)$ denotes the cumulative rewards achieved by rolling out policy π.

Theorem 2. *For any learner policies π_i within learner group $\{\pi_1, \ldots, \pi_n\}$ and an actor policy μ, with importance sampling truncation in Eq. (2), the following reward improvement lower bound holds:*

$$J(\pi_i) - J(\mu) \geq -\frac{\gamma \epsilon^{\pi_i} \sqrt{2 \log \rho}}{(1-\gamma)^2},$$

where the constant $\epsilon^{\pi} \doteq \max_s |\mathbb{E}_{a \sim \pi}[A^{\mu}]|$, γ is the discount factor, and ρ is the importance sampling truncation threshold in Eq. (2), respectively.

Proof. We refer to the Corollary 1 from Achiam et al. [2]. Given a learner policy π_i and actor policy μ, Theorem 2 can be derived as

$$
\begin{aligned}
&J(\pi_i) - J(\mu) \\
&\geq \frac{1}{1-\gamma} \mathop{\mathbb{E}}_{\substack{s\sim\mu \\ a\sim\pi_i}} \Big[A^{\mu} - \frac{2\gamma\epsilon^{\pi_i}}{1-\gamma} D_{TV}(\pi_i \| \mu) \Big], && (5) \\
&\geq \frac{1}{1-\gamma} \mathop{\mathbb{E}}_{\substack{s\sim\mu \\ a\sim\pi_i}} \Big[- \frac{2\gamma\epsilon^{\pi_i}}{1-\gamma} \sqrt{\frac{D_{KL}(\pi_i \| \mu)}{2}} \Big], && (6) \\
&= \frac{1}{1-\gamma} \mathop{\mathbb{E}}_{\substack{s\sim\mu \\ a\sim\pi_i}} \Big[- \frac{\gamma\epsilon^{\pi_i}}{1-\gamma} \sqrt{2 \sum_a \pi_i(a|s) \log(\frac{\pi_i(a|s)}{\mu(a|s)})} \Big], && (7) \\
&\geq \frac{1}{1-\gamma} \mathop{\mathbb{E}}_{\substack{s\sim\mu \\ a\sim\pi_i}} \Big[- \frac{\gamma\epsilon^{\pi_i}}{1-\gamma} \sqrt{2 \log \rho \sum_a \pi_i(a|s)} \Big], && (8) \\
&= -\frac{\gamma\epsilon^{\pi_i}\sqrt{2\log\rho}}{(1-\gamma)^2}.
\end{aligned}
$$

where $D_{TV}(\pi_i||\mu)$ and $D_{KL}(\pi_i||\mu)$ are the Total Variation (TV) and KL divergence between action distributions of π_i and μ, respectively. Eq. 5 follows the Corollary 1 from Achiam et al. [2]. We then replace TV divergence with KL divergence in Eq. 6 by Pinsker's inequality [14], *i.e.*, $D_{TV}(\pi_i||\mu) \leq \sqrt{\frac{D_{KL}(\pi_i||\mu)}{2}}$. From Eq. 7 to Eq. 8, we apply our importance sampling truncation and its clip threshold ρ in Eq. 2. Finally, we can reach the form of Theorem 2. □

VII. Implementation

Stellaris is designed to be a generic asynchronous learning paradigm for accelerating distributed DRL training with serverless computing. For concreteness, we describe its implementation in the context of AWS EC2 instances [6]. We implement *Stellaris* with 5K lines of Python, which is open-sourced[1]. We describe the detailed implementation of *Stellaris*'s components and features below.

Serverless parameter and learner function. Due to the lack of serverless GPUs in existing serverless platforms [4, 7, 8, 18, 29, 52], we implement our own serverless container cluster using AWS EC2 accelerated instances with GPU accelerators for parameter and learner functions. On each instance, we use Docker container [47], which is used by many open-source serverless frameworks [4, 13, 52], to implement serverless containers. We use NVIDIA container runtime [51] to enable GPU support for serverless functions. The core logic of the parameter and learner function is implemented using PyTorch [54], including the neural networks of the policy model, staleness-aware gradient aggregation, and the importance sampling truncation process. The dependencies of the functions are installed and packaged as Docker container images for fast deployment. Before training DRL tasks, *Stellaris* profiles information about the execution time and resource demand of the parameter and learner functions with the task to run. The profiled information is collected by *Stellaris* in actual training. We pre-warm the containers prior to the invocations of the functions based on estimated completion time and keep the containers alive for ten minutes (as the same in OpenWhisk) to further reduce the function startup overhead.

[1] https://github.com/IntelliSys-Lab/Stellaris-SC24

TABLE II: Neural network architecture used in DRL training.

Task	Layer	Activation	Size
MuJoCo	Fully-connect	Tanh	2 × 256
Atari	Convolutional	ReLU	16, 8 × 8 32, 4 × 4 256, 11 × 11

TABLE III: PPO's and IMPACT's hyperparameters.

Parameter	PPO	IMPACT
Learning rate	0.00005	0.0005
Discount factor (γ)	0.99	0.99
Batch size (Mujoco)	4096	4096
Batch size (Atari)	256	256
Clip parameter	0.3	0.4
KL coefficient	0.2	1.0
KL target	0.01	0.01
Entropy coefficient	0.0	0.01
Value function coefficient	1.0	1.0
Target update frequency	N/A	1.0

Actors. For baselines with serverful actors [43, 45], we use AWS EC2 compute instances as the backend. We use the Python multiprocessing library to implement and run concurrent actors for sampling trajectories. For serverless actors, we follow the methodology described in [79] and use Docker containers to implement actor functions on EC2 instances.

Distributed cache. We use Redis [59], an in-memory key-value cache, to implement the distributed cache in *Stellaris*. The Redis instance resides on the EC2 instances that holds serverless containers and provides high-performant communications between different system components. Upon completing sampling, actors serialize the trajectories using Pickle [55] and submit the serialized sample batch to Redis. Learner functions then deserialize trajectories and fetch latest policy model weights from Redis to compute gradients. After calculation, the gradients are also serialized and sent to Redis. The parameter function picks up the gradients, performs the aggregation, and sends the updated policy weights back to Redis.

VIII. Evaluation

A. Experimental Setup

Testbeds. We deploy all baselines and *Stellaris* to a cluster of AWS EC2 VMs, including two `p3.2xlarge` and one `c6a.32xlarge` for regular experiments. The cluster contains two NVIDIA V100 GPUs, 32 GB GPU memory, 128 AMD EPYC 7R13 CPU cores, and 317 GB CPU memory for training DRL tasks. We also evaluate *Stellaris*'s scalability and large-scale deployment on a simulated HPC cluster with singularity containers [38]. The HPC cluster contains two

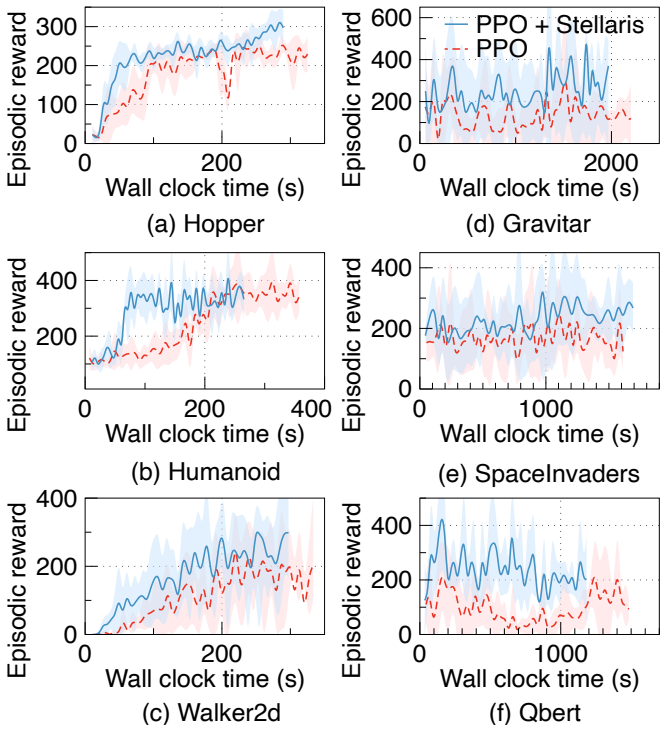

Fig. 6: *Stellaris* accelerates PPO training.

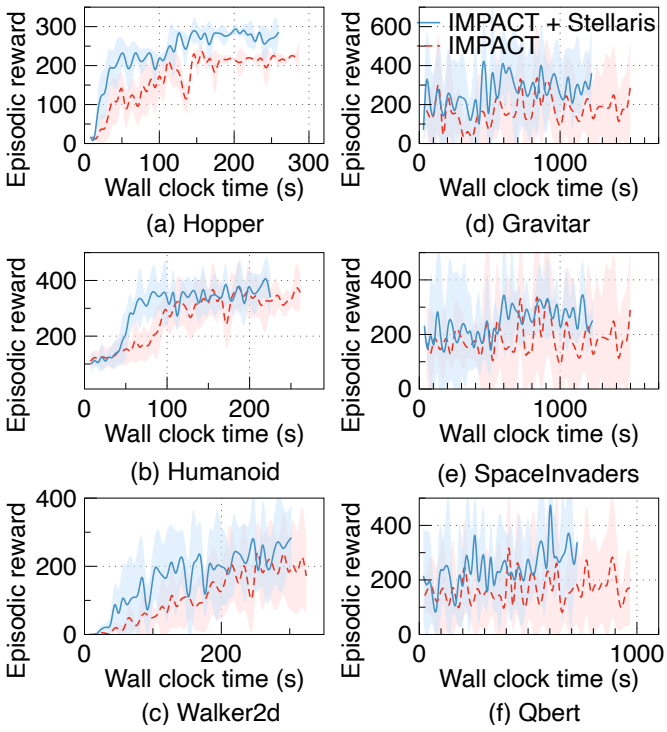

Fig. 7: *Stellaris* accelerates IMPACT training.

`p3.16xlarge` instances with 16 NVIDIA V100 GPUs and five `hpc7a.96xlarge` instances with 960 AMD EPYC 9R14 CPU cores[2]. We use the V100 GPUs to execute parameter and learner functions and the remaining CPU cores to host actors.

Cost model. Following the methodology of existing research [60], we charge each serverless function invocation based on the function resource size and invocation latency. The price unit is dollar-per-resource-second, calculated by dividing the cost per second from AWS EC2 instances by the maximum capacity of concurrent running learner functions allowed per VM. For instance, if we limit the capacity of learner functions to four per VM, the cost of a function invocation with a V100 GPU is computed by dividing the price of `p3.2xlarge` by four and then multiplied by its execution time. Similar to existing serverless platforms [7, 8, 18, 29], we exclude the cost of container pre-warming and keep-alive services.

Workloads. Six popular environments from OpenAI Gym are used to evaluate *Stellaris* and SOTA baselines, including three continuous-action MuJoCo environments (Hopper, Humanoid, and Walker2d) and three discrete-action Atari environments (SpaceInvaders, Qbert, and Gravitar). Table II characterizes the policy networks used in our evaluation. For three MuJoCo environments, the policy network consists of two fully-connected layers of 256 hidden units with `Tanh` activation. For three Atari environments, the policy network consists of three convolutional layers of 8×8, 4×4, and 11×11 kernel sizes with `ReLU` activation, respectively. The input sampled from Atari games is a stack of three 84×84 images. In both cases, the critic networks share the same architecture as the policy networks.

[2]We use AWS services in the US East 2 region. The hourly unit prices for `p3.2xlarge`, `c6a.32xlarge`, `p3.16xlarge`, and `hpc7a.96xlarge` are $3.06, $4.896, $24.48, and $7.2, respectively.

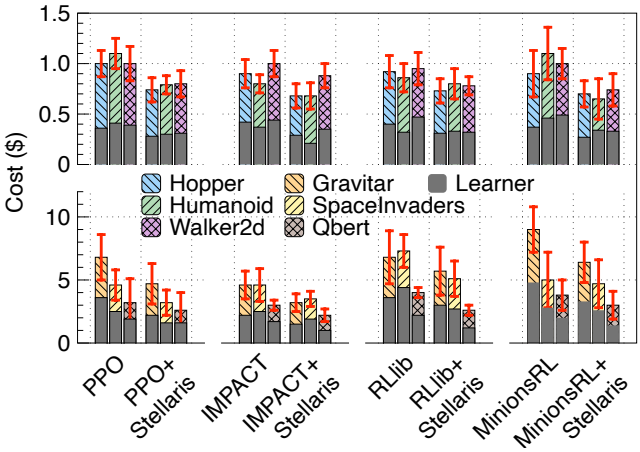

Fig. 8: *Stellaris* reduces training costs of PPO, IMPACT, RLlib, and MinionsRL. Grey bars represent the time spent on the learner, and the rest indicate time spent on actors.

***Stellaris*'s setting.** We set the maximum capacity of learner functions to be four per V100 GPU and allocate one actor per CPU core. The exponential decay factor d in Eq. 3 is set to 0.96 when evaluating *Stellaris*. The learning rate smoothness factor v in Eq. 4 is set to 3. The importance sampling truncation threshold ρ in Eq. 2 is set to 1.0. We further evaluate the parameter sensitivity in §VIII-E.

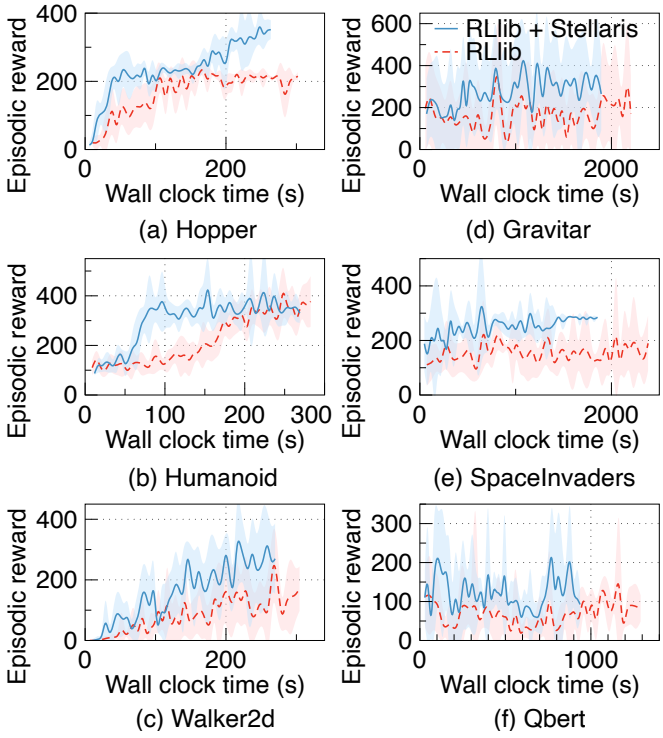

Fig. 9: *Stellaris* improves RLlib tasks in time efficiency.

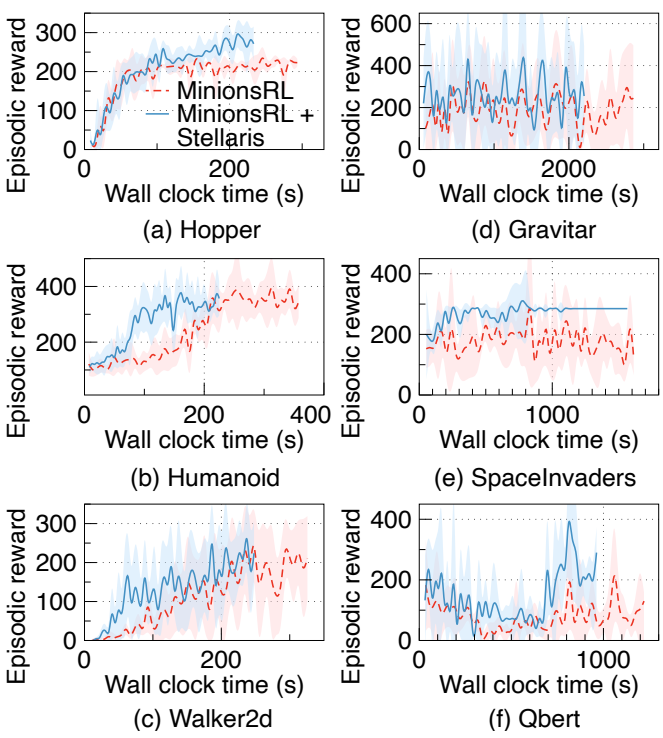

Fig. 10: *Stellaris* improves MinionsRL tasks in time efficiency.

B. Overall Performance

1) Integrating with DRL Algorithms: We evaluate how *Stellaris* boosts SOTA DRL algorithms. Specifically, we integrate *Stellaris* with two algorithms, one on-policy and one off-policy: 1) **PPO** [66] is the most famous on-policy DRL algorithm that has been employed in extensive applications [9, 12, 56, 57]. We implement a standard distributed PPO with Generalized Advantage Estimation (GAE) [65] and surrogate objective clipping [66]. 2) **IMPACT** [45] is a SOTA off-policy algorithm. IMPACT itself builds on a long list of improvements over PPO and combines various tricks for asynchronous training, such as V-trace importance sampling [16] and the surrogate target network [44]. We use PPO and IMPACT as baselines and integrate them with *Stellaris* for comparison. Table III describes the hyperparameter settings of PPO and IMPACT used in the evaluation. We employ the same hyperparameter settings from tuned examples in RLlib [43]. Both PPO and IMPACT training use Adam optimizer [36]. We train each algorithm for 50 rounds in six environments. The results are averaged over ten repeated experiments, each with a different random seed.

Training efficiency. Figs. 6 and 7 show the episodic rewards through training in six environments for PPO and IMPACT, respectively. IMPALA completes training faster than PPO in most of the environments due to the advantage of being off-policy. *Stellaris* outperforms the vanilla PPO and IMPACT by training faster in both statistical efficiency and wall clock time. Compared to PPO and IMPACT, *Stellaris* improves the final reward by up to 2.2× and 1.3×, respectively.

Training cost. Fig. 8 shows the training costs of the two vanilla baselines and variants integrated with *Stellaris*. Compared to original PPO and IMPACT, *Stellaris* reduces training costs by up to 31% and 30%, respectively.

2) Integrating with DRL Frameworks: We also evaluate how *Stellaris* improves SOTA DRL frameworks. Two popular DRL frameworks are integrated with *Stellaris* in the evaluation: 1) **Ray RLlib** [43] is an open-source industrial-grade RL library with a comprehensive implementation of algorithms. For integration, we implement the logic of our asynchronous serverless learner functions inside RLlib's default learner group. 2) **MinionsRL** [79] is a state-of-the-art DRL training framework that also leverages serverless computing. MinionsRL employs a DRL-based actor scheduler to dynamically scale serverless actors, which tries to solve the scheduling problem via black-box optimization. We keep MinionsRL's serverless actors while replacing its synchronous learners with our asynchronous serverless learner functions. We run PPO with two frameworks in six environments using the same experimental setting in §VIII-B1.

Training efficiency. Figs. 9 and 10 show the episodic rewards through training in six environments for RLlib and MinionsRL, respectively. In both frameworks, we observe similar improvements with *Stellaris*. *Stellaris* accelerates PPO training in two frameworks by improving both statistical and training efficiency. Compared to the RLlib and MinionsRL, *Stellaris* improves the final reward by up to 1.3× and 1.6×, respectively.

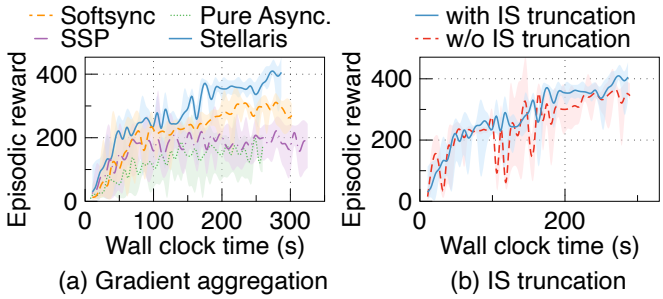

Fig. 11: Ablation study of *Stellaris*.

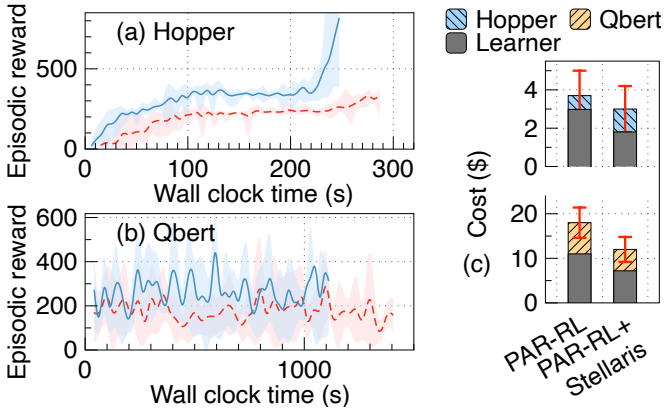

Fig. 12: *Stellaris* with PAR-RL on HPC cluster.

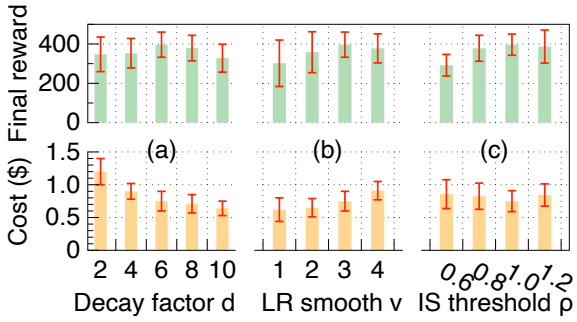

Fig. 13: Sensitivity analysis in the Hopper environment.

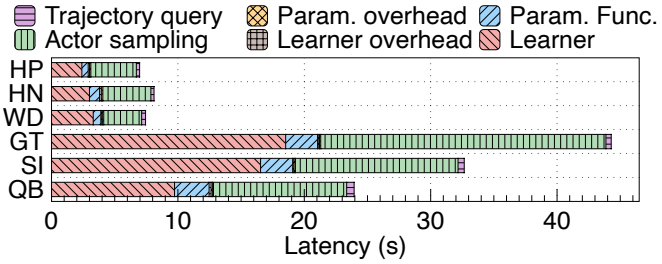

Fig. 14: Latency breakdown of six environments with *Stellaris*'s one-round training.

Training cost. Fig. 8 shows the training costs of the two frameworks and variants integrated with *Stellaris*. Compared to RLlib and MinionsRL, *Stellaris* reduces training costs by up to 38% and 41%, respectively.

C. Ablation Study

We verify the effectiveness of two key designs: staleness-aware gradient aggregation and importance sampling truncation, by training PPO in the Hopper environment using the same experimental setting described in §VIII-B.

To evaluate the gradient aggregation, we compare *Stellaris* with three baselines: 1) **Softsync** [82] delays the aggregation until receiving a certain number of gradients to control staleness. 2) **Stale Synchronous Parallel (SSP)** [25] pauses fast learners to make slow ones to keep up, thus reducing staleness. 3) **Pure asynchronous** is a baseline without any controls on the staleness. We implement the above gradient aggregation methods in *Stellaris* while keeping the use of serverless computing for a fair comparison. Fig. 11(a) shows the episodic reward for the four baselines. Though pure asynchronous baseline trains faster than others, it fails to provide satisfactory rewards due to slow convergence. *Stellaris* outperforms other baselines by achieving the best cumulative reward.

We also compare *Stellaris* with a variant of itself by disabling the importance sampling truncation. Fig. 11(b) reports the training performance of *Stellaris* and its variant. Without the truncation, *Stellaris* experiences unstable training and performance oscillation. The result demonstrates that importance sampling truncation is necessary to stabilize *Stellaris* training.

D. Scalability on HPC cluster

We conducted a large-scale experiment on an AWS EC2 cluster with singularity containers [38] to simulate running on HPC clusters. An RL workload developed by the Argonne Leadership Computing Facility for HPC, PAR-RL [5], is used to evaluate *Stellaris*'s performance. Due to budget limits, we only evaluate PAR-RL with Hopper and Qbert environments. Fig. 12 presents the episodic rewards through training and total cost. In an HPC setting, *Stellaris* outperforms PAR-RL by improving the final reward by 2.4× and 1.1× while reducing training costs by 19% and 34%, for Hopper and Qbert, respectively.

E. Sensitivity Analysis

We analyze the sensitivity of three parameters in *Stellaris*: decay factor d, learning rate smoothness v, and importance sampling threshold ρ. We run the same experiment in §VIII-B, *i.e.*, training PPO in the Hopper environment, but with different parameter values for analysis. The results are reported in Fig. 13. Other combinations of algorithms and environments show similar sensitivity results.

Decay factor d. We set the decay factor to 0.96 in the evaluation. Fig. 13(a) shows the achieved final reward and training cost when gradually increasing the factor from 0.92 to 1.0 in the step of 0.02. When the factor increases, the final reward cost increases while the cost decreases because *Stellaris* allows higher staleness and training more asynchronously. The final reward stops growing at 0.96.

Learning rate smooth factor v. In our evaluation, we set the learning rate smoothness to 3. Fig. 13(b) shows the

final reward and training cost when gradually increasing the learning rate smoothness from 1 to 4. Large staleness may modulate the learning rate to be too small and diminish the policy updates. By setting larger v, *Stellaris* allows policy updates to be less modulated by staleness, hence keeping large step sizes of policy updates. We observe that *Stellaris* achieves optimal performance with a smooth factor of 3.

Importance sampling threshold ρ. Fig. 13(c) shows final rewards and training cost of running PPO with different importance sampling thresholds. We gradually increase the value from 0.6 to 1.2 in the step of 0.2. With larger thresholds, *Stellaris* allows the action distributions generated by actor policy to deviate more from the learner policies, whereas small ones keep *Stellaris*'s learning more conservative. The optimal performance and minimal cost are achieved with $\rho = 1.0$.

F. Latency Breakdown and Overheads

We report the latency breakdown and overheads of parameter and learner function in *Stellaris*'s one-round training. Fig. 14 characterizes the latency breakdown of six environments used in evaluation with *Stellaris*'s one-round training. The latency is measured when running PPO in the same experimental setting as §VIII-B. The total overhead of all components incurs less than 5% delay, which is negligible for the one-round latency while providing accelerated training and cost-efficiency.

IX. Related Work

DRL frameworks. Recently, the DRL community has developed extensive open-source training frameworks [1, 26, 28, 43, 58, 85]. Acme [26] is a research-oriented DRL framework. Stable-Baselines3 [58] is developed for reliable DRL implementation. CleanRL [28] aims to provide high-quality single-file DRL implementations. SpinningUP [1] focuses on educational purposes for DRL beginners. RLlib [43] provides industry-grade DRL framework. MSRL [85] uses fragmented dataflow graphs to execute DRL algorithms. *Stellaris* can be integrated with existing DRL frameworks to further optimize the training process.

Distributed DRL training. A3C [50] firstly introduced a simple actor-leaner prototype. IMPALA [16] is the first off-policy (asynchronous) actor-learner architecture with V-trace correction. IMPACT [45] stabilized DRL training performance by adding a surrogate target network to the actor-learner architecture. SEED RL [17] aimed to accelerate off-policy actor-learner architectures by shifting actor inferences to centralized GPU servers. Most existing distributed RL solutions [16, 17, 41, 45, 46, 50] are designed for serverful architectures, thus can hardly exploit the agile scalability and fine-grained resource provisioning of serverless computing. MinionsRL [79] is the closest work to us, which also leverages serverless computing to design DRL training systems. However, MinionsRL is limited to serving on-policy algorithms and synchronous learning. Unlike the above, *Stellaris* leverages serverless computing to accelerate DRL training with an efficient asynchronous learning paradigm.

Serverless ML training. Serverless computing has recently attracted the ML community to design novel training frameworks [10, 21, 31, 60, 75]. Cirrus [10] proposes a serverless framework that simplifies end-to-end ML training. Siren [75] designs a DRL function scheduler to automate distributed ML training on serverless computing platforms. Jiang et al. [31] conducts a comprehensive comparison between serverful and serverless ML training, which indicates serverless training is cost-efficient. Hydrozoa [21] presents a deep neural network (DNN) training framework on top of serverless containers with dynamic data and model parallelism. However, none of the above studies exploits the benefits of serverless computing for asynchronous learning in distributed DRL training.

Staleness in asynchronous training. Many studies have proved that staled gradients (*i.e.*, staleness) in asynchronous ML training can degrade the training performance and delay convergence [15, 42, 82, 84]. Extensive studies have been proposed to bound the asynchrony in ML training [25, 75, 82] and federated learning [11, 37, 53, 70]. In distributed RL training, asynchrony usually refers to the policy lag between learners and actors, which has been focused by many works [16, 17, 45, 50, 64]. Unlike existing works, *Stellaris* is the first to address the staleness of serverless asynchronous learners in DRL.

X. Conclusion

This paper proposes *Stellaris*, the first work to introduce a generic asynchronous learning paradigm for distributed DRL training with serverless computing. *Stellaris* supports both serverful and serverless training infrastructures as well as on- and off-policy DRL algorithms. We leverage the agile scalability and fine-grained allocation of serverless computing for *Stellaris* to meet the dynamic and transient resource demands in distributed DRL. We devise a novel importance sampling truncation technique to stabilize the asynchronous DRL training process. To deal with dynamic staleness, we develop a gradient aggregation method with staleness-awareness for accelerating DRL training and guaranteeing convergence. We evaluate *Stellaris* with popular DRL algorithms and frameworks on AWS EC2 regular testbeds and HPC clusters with 16 GPUs and 960 CPU cores. Extensive experiments show that *Stellaris* outperforms existing state-of-the-art DRL baselines by achieving up to 2.2× higher rewards (*i.e.*, training quality) and reducing up to 41% training costs.

XI. Acknowledgements

We thank anonymous reviewers for their valuable feedback. The work of Hanfei Yu and Hao Wang was supported in part by NSF 2153502, 2403247, 2403398, and the AWS Cloud Credit for Research program. The work of Devesh Tiwari was supported by NSF 2124897. The work of Jian Li was supported in part by NSF 2148309 and 2337914. The work of Seung-Jong Park was supported in part by NSF 2403248 and 2403399. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Joshua Achiam. Spinning up in deep reinforcement learning. https://spinningup.openai.com, 2018.

[2] Joshua Achiam, David Held, Aviv Tamar, and Pieter Abbeel. Constrained Policy Optimization. In *International Conference on Machine Learning (ICML)*, 2017.

[3] Shinyoung Ahn, Joongheon Kim, Eunji Lim, Wan Choi, Aziz Mohaisen, and Sungwon Kang. ShmCaffe: A Distributed Deep Learning Platform with Shared memory Buffer for HPC Architecture. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018.

[4] Apache. Apache OpenWhisk Official Website. https://openwhisk.apache.org, 2018.

[5] Argonne Leadership Computing Facility. PAR-RL. https://github.com/Romit-Maulik/PAR-RL, 2022.

[6] AWS. AWS EC2: Secure and Resizable Compute Capacity in the Cloud. https://aws.amazon.com/ec2/, 2006.

[7] AWS. AWS Lambda: Serverless Compute. https://aws.amazon.com/lambda/, 2014.

[8] Azure Functions. Azure Functions. https://azure.microsoft.com/en-us/products/functions/, 2022.

[9] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv preprint arXiv:1912.06680*, 2019.

[10] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *ACM Symposium on Cloud Computing (SoCC)*, 2019.

[11] Zheng Chai, Yujing Chen, Ali Anwar, Liang Zhao, Yue Cheng, and Huzefa Rangwala. FedAT: A High-Performance and Communication-Efficient Federated Learning System with Asynchronous Tiers. In *International Conference for High-Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[12] Xiangyu Chen, Zelin Ye, Jiankai Sun, Yuda Fan, Fang Hu, Chenxi Wang, and Cewu Lu. Transferable Active Grasping and Real Embodied Dataset. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

[13] Cloud Native Computing Foundation. Knative. https://knative.dev/docs/, 2018.

[14] Imre Csiszár and János Körner. *Information Theory: Coding Theorems for Discrete Memoryless Systems*. Cambridge University Press, 2011.

[15] Wei Dai, Yi Zhou, Nanqing Dong, Hao Zhang, and Eric Xing. Toward Understanding the Impact of Staleness in Distributed Machine Learning. In *International Conference on Learning Representations (ICLR)*, 2019.

[16] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. In *International Conference on Machine Learning (ICML)*, 2018.

[17] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed RL: Scalable and Efficient Deep-RL with Accelerated Central Inference. In *International Conference on Learning Representations (ICLR)*, 2020.

[18] Google Cloud. Google Cloud Function:Event-Driven Serverless Compute Platform. https://cloud.google.com/functions, 2018.

[19] Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. Variance Reduction Techniques for Gradient Estimates in Reinforcement Learning. *Journal of Machine Learning Research (JMLR)*, 2004.

[20] Shixiang Shane Gu, Timothy Lillicrap, Richard E Turner, Zoubin Ghahramani, Bernhard Schölkopf, and Sergey Levine. Interpolated Policy Gradient: Merging On-Policy and Off-Policy Gradient Estimation for Deep Reinforcement Learning. *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[21] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. Hydrozoa: Dynamic Hybrid-Parallel DNN Training on Serverless Containers. *Proceedings of Machine Learning and Systems (MLSys)*, 2022.

[22] Yu Hanfei, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[23] Wang Hao, Niu Di, and Li Baochun. Distributed Machine Learning with a Serverless Architecture. In *IEEE Conference on Computer Communications (INFOCOM)*, 2019.

[24] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. In *Thirty-second AAAI conference on artificial intelligence (AAAI)*, 2018.

[25] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More Effective Distributed ML via A Stale Synchronous Parallel Parameter Server. In *Advances in Neural Information Processing Systems (NIPS)*, 2013.

[26] Matthew W Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Nikola Momchev, Danila Sinopalnikov, Piotr Stańczyk, Sabela Ramos, Anton Raichuk, Damien Vincent, et al. Acme: A Research Framework for Distributed Reinforcement Learning. *arXiv preprint arXiv:2006.00979*, 2020.

[27] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado Van Hasselt, and David Silver. Distributed Prioritized Experience Replay. In

International Conference on Learning Representations (ICLR), 2018.

[28] Shengyi Huang, Rousslan Fernand JulienDossa Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João GM Araújo. CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms. *The Journal of Machine Learning Research*, 2022.

[29] IBM. IBM Cloud Functions. https://www.ibm.com/cloud/functions, 2021.

[30] Edward L Ionides. Truncated Importance Sampling. *Journal of Computational and Graphical Statistics*, 2008.

[31] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards Demystifying Serverless Machine Learning Training. In *2021 International Conference on Management of Data (SIGMOD)*, 2021.

[32] Tang Jie and Pieter Abbeel. On a Connection Between Importance Sampling and the Likelihood Ratio Policy Gradient. *Advances in Neural Information Processing Systems (NIPS)*, 2010.

[33] Hendrik Jung, Roberto Covino, and Gerhard Hummer. Artificial Intelligence Assists Discovery of Reaction Coordinates and Mechanisms from Molecular Dynamics Simulations. *arXiv preprint arXiv:1901.04595*, 2019.

[34] Sham Kakade and John Langford. A Closer Look at Deep Policy Gradients. In *International Conference on Learning Representations (ICLR)*, 2020.

[35] Steven Kapturowski, Georg Ostrovski, John Quan, Remi Munos, and Will Dabney. Recurrent Experience Replay in Distributed Reinforcement Learning. In *International Conference on Learning Representations (ICLR)*, 2018.

[36] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[37] Anastasiia Koloskova, Sebastian U Stich, and Martin Jaggi. Sharper Convergence Guarantees for Asynchronous SGD for Distributed and Federated Learning. *Advances in Neural Information Processing Systems (NIPS)*, 2022.

[38] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. Singularity: Scientific Containers for Mobility of Compute. *PloS One*, 12(5):e0177459, 2017.

[39] Guillaume Lample and Devendra Singh Chaplot. Playing FPS Games with Deep Reinforcement Learning. In *Thirty-First AAAI Conference on Artificial Intelligence (AAAI)*, 2017.

[40] Dongsheng Li, Zhiquan Lai, Keshi Ge, Yiming Zhang, Zhaoning Zhang, Qinglin Wang, and Huaimin Wang. HPDL: Towards a General Framework for High-Performance Distributed Deep Learning. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[41] Zechu Li, Tao Chen, Zhang-Wei Hong, Anurag Ajay, and Pulkit Agrawal. Parallel Q-Learning: Scaling Off-policy Reinforcement Learning under Massively Parallel Simulation. In *ES-FoMo Workshop at ICML*, 2023.

[42] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization. *Advances in Neural Information Processing Systems (NIPS)*, 28, 2015.

[43] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for Distributed Reinforcement Learning. In *International Conference on Machine Learning (ICML)*, 2018.

[44] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous Control with Deep Reinforcement Learning. *arXiv preprint arXiv:1509.02971*, 2015.

[45] Michael Luo, Jiahao Yao, Richard Liaw, Eric Liang, and Ion Stoica. IMPACT: Importance Weighted Asynchronous Architectures with Clipped Target Networks. In *International Conference on Learning Representations (ICLR)*, 2020.

[46] Zhiyu Mei, Wei Fu, Guangju Wang, Huanchen Zhang, and Yi Wu. SRL: Scaling Distributed Reinforcement Learning to Over Ten Thousand Cores. *arXiv preprint arXiv:2306.16688*, 2023.

[47] Dirk Merkel et al. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014.

[48] Alberto Maria Metelli, Matteo Papini, Francesco Faccio, and Marcello Restelli. Policy Optimization via Importance Sampling. *Advances in Neural Information Processing Systems (NIPS)*, 2018.

[49] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device Placement Optimization with Reinforcement Learning. In *International Conference on Machine Learning (ICML)*, 2017.

[50] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning (ICML)*, 2016.

[51] NVIDIA. NVIDIA Container Runtime. https://github.com/NVIDIA/nvidia-container-runtime, 2023.

[52] OpenFaaS. OpenFaaS - Serverless Functions Made Simple. https://docs.openfaas.com/, 2018.

[53] Jungwuk Park, Dong-Jun Han, Minseok Choi, and Jaekyun Moon. Sageflow: Robust Federated Learning Against Both Stragglers and Adversaries. *Advances in Neural Information Processing Systems (NIPS)*, 2021.

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NIPS)*, 2019.

[55] Python. Pickle — Python Object Serialization. https://docs.python.org/3/library/pickle.html, 2008.

[56] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. {FIRM}: An Intelligent Fine-grained Resource Management Framework for {SLO-Oriented} Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[57] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language Models are Unsupervised Multitask Learners. *OpenAI blog*, 2019.

[58] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable Reinforcement Learning Implementations. *The Journal of Machine Learning Research*, 2021.

[59] Redis. Redis Official Website. http://redis.io/, 2009.

[60] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. In *ACM Symposium on Cloud Computing (SoCC)*, 2021.

[61] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. DayDream: Executing Dynamic Scientific Workflows on Serverless Platforms with Hot Starts. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[62] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. IceBreaker: Warming Serverless Functions Better with Heterogeneity. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022.

[63] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust Region Policy Optimization. In *International Conference on Machine Learning (ICML)*, 2015.

[64] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[65] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. In *International Conference on Learning Representations (ICLR)*, 2016.

[66] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[67] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *USENIX Annual Technical Conference (USENIX ATC)*, 2020.

[68] Zahra Shamsi, Kevin J Cheng, and Diwakar Shukla. Reinforcement Learning based Adaptive Sampling: REAPing Rewards By Exploring Protein Conformational Landscapes. *The Journal of Physical Chemistry B*, 2018.

[69] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 2016.

[70] Justin Stanley and Ali Jannesari. Addressing Stale Gradients in Scalable Federated Deep Reinforcement Learning. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 932–940, 2023.

[71] Ryan Sullivan, Justin K Terry, Benjamin Black, and John P Dickerson. Cliff Diving: Exploring Reward Surfaces in Reinforcement Learning Environments. In *Nineteenth International Conference on Machine Learning (ICML)*, 2022.

[72] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems (NIPS)*, 1999.

[73] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A Physics Engine for Model-based Control. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

[74] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster Level in StarCraft II Using Multi-agent Reinforcement Learning. *Nature*, 2019.

[75] Hao Wang, Di Niu, and Baochun Li. Distributed Machine Learning with a Serverless Architecture. In *IEEE 2019 Conference on Computer Communications (INFOCOM)*, 2019.

[76] Logan Ward, Ganesh Sivaraman, J Gregory Pauloski, Yadu Babuji, Ryan Chard, Naveen Dandu, Paul C Redfern, Rajeev S Assary, Kyle Chard, Larry A Curtiss, et al. Colmena: Scalable Machine-Learning-based Steering of Ensemble Simulations for High performance Computing. In *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*, 2021.

[77] Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. DD-PPO: Learning Near-Perfect PointGoal Navigators from 2.5 Billion Frames. *arXiv preprint arXiv:1911.00357*, 2019.

[78] Tengyang Xie, Yifei Ma, and Yu-Xiang Wang. Towards Optimal Off-policy Evaluation for Reinforcement Learning with Marginalized Importance Sampling. *Advances in Neural Information Processing Systems (NIPS)*, 2019.

[79] Hanfei Yu, Jian Li, Yang Hua, Xu Yuan, and Hao Wang. Cheaper and Faster: Distributed Deep Reinforcement Learning with Serverless Computing. In *Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI)*, 2024.

[80] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and

Bing Xie. RLScheduler: An Automated HPC Batch Job Scheduler using Reinforcement Learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[81] Di Zhang, Dong Dai, and Bing Xie. SchedInspector: A Batch Job Scheduling Inspector Using Reinforcement Learning. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2022.

[82] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-Aware Async-SGD for Distributed Deep Learning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2016.

[83] Xing Zhao, Aijun An, Junfeng Liu, and Bao Xin Chen. Dynamic Stale Synchronous Parallel Distributed Training for Deep Learning. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[84] Yi Zhou, Yaoliang Yu, Wei Dai, Yingbin Liang, and Eric Xing. On Convergence of Model Parallel Proximal Gradient Algorithm for Stale Synchronous Parallel System. In *Artificial Intelligence and Statistics (AISTATS)*, 2016.

[85] Huanzhou Zhu, Bo Zhao, Gang Chen, Weifeng Chen, Yijie Chen, Liang Shi, Yaodong Yang, Peter Pietzuch, and Lei Chen. MSRL: Distributed Reinforcement Learning with Dataflow Fragments. In *2023 USENIX Annual Technical Conference (ATC)*, 2023.

[86] Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven Visual Navigation in Indoor Scenes Using Deep Reinforcement Learning. In *IEEE International Conference on Robotics and Automation (ICRA 2017)*, 2017.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 Asynchronous serverless training for distributed deep reinforcement learning (DRL).

B. Computational Artifacts

A_1 https://zenodo.org/doi/10.5281/zenodo.12589953

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figures 6-13

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

In our paper, we present the demo artifact A_1 that showcases *Stellaris*. A_1 runs the experiments by defining the environmental setup, such as datasets, hyperparameters, and evaluation metrics. It also implements the training system components of *Stellaris* on the top of Ray RLlib. C_1 can be evaluated and supported by A_1.

Expected Results

By integrating *Stellaris* with existing distributed DRL training frameworks and algorithms, we should observe *Stellaris* achieves higher training efficiency with less training costs than the baseline.

Expected Reproduction Time (in Minutes)

The expected time to reproduce artifacts A_1 on a GPU machine is 60 minutes if built from scratch and 20 minutes by pulling the provided Docker container images.

Artifact Setup (incl. Inputs)

Hardware: Recommended hardware specifies for reproducing artifacts:

- Cloud platform: AWS EC2
- Minimal VM instance size: equivalent to `p3.2xlarge`
- Resources: one NVIDIA V100 GPU, 16 GB GPU memory, 8 CPU cores, and 61 GB memory

Software: We provide an installation script to automate the download of all necessary dependencies to reproduce the artifacts. If no version number is specified, the latest version is used.

- OS version: Ubuntu 22.04
- Apt: libosmesa6-dev, libgl1-mesa-glx, libglfw3, libglew-dev, patchelf, build-essential, curl, gpg, gcc-9, gcc-11, g++-9, pkg-config, psmisc, redis-server
- Pip: torch, gymnasium==0.28.1, gymnasium[atari,accept-rom-license]==0.28.1, ray[rllib]==2.8, mujoco, mujoco-py, pygame, pandas, cython<3, redis

Datasets / Inputs: We use two DRL datasets (benchmarks) for running the experiments: Mujoco and Atari environments, which are publicly available and accessible via the links below:

- Mujoco: https://github.com/openai/mujoco-py
- Atari: https://github.com/openai/atari-py

The two datasets can be easily installed using the Gymnasium library at https://github.com/Farama-Foundation/Gymnasium.

Installation and Deployment: We provide an installation script to automate the download of all dependencies needed to reproduce the artifacts. Concretely, our artifacts require stalling `gcc` and `g++` compilers (both version 9 and 11) for building the necessary system components of *Stellaris*. To enable GPU support, an NVIDIA driver (version 525) is also required. Additionally, the Bazel tool (version 7.1.1) is required to compile the Ray RLlib framework.

Artifact Execution

Our experimental workflow contains two tasks: T_1: executing the training process, and T_2: visualizing the results from the training output. For T_1, we automate the execution of training algorithms on all baselines. T_1's output is in CSV files for figure-plotting in T_2.

Artifact Analysis (incl. Outputs)

The training output data is recorded in CSV files with the following attributes: training round index, round duration, number of learner functions invoked per training iteration, episodes executed, evaluation rewards, staleness, and training cost. Most figures in our paper can be plotted by analyzing and extracting key metrics from the output CSV files.

Artifact Evaluation (AE)

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

General hardware prerequisite:

- Operating systems and versions: Ubuntu 22.04
- Resource requirement: CPU $\geq$ 36 cores, memory $\geq$ 100 GB, disk $\geq$ 50 GB, network has no requirement since it's a single-node demo

Chameleon Cloud prerequisite:

- Chameleon Cloud UC access: Instance note type must be **gpu_rtx_6000**
- Chameleon Cloud image: The image must be **CC-Ubuntu22.04**

Chameleon Cloud instance setup:

- Create a lease to reserve hosts: *Reservations* $\rightarrow$ *Leases* $\rightarrow$ *Hosts* $\rightarrow$ *Reserve Hosts* $\rightarrow$ *Resource Properties* $\rightarrow$ *node_type* $\rightarrow$ *gpu_rtx_6000*.
- Launch a *GPU RTX 6000* instance using the image *CC-Ubuntu22.04*. The image can be found by searching the image name: *Project* $\rightarrow$ *Compute* $\rightarrow$ *Images* $\rightarrow$ *Search* $\rightarrow$ *Launch*. Create a key pair if necessary.
- Assign a floating IP to your instance for login: *Project* $\rightarrow$ *Network* $\rightarrow$ *Floating IPs* $\rightarrow$ *Allocate IP To Project*. We refer readers to Chameleon cloud documents for instance creation and login details.

Artifact Execution

Deployment instructions:

- Download the GitHub repository by `git clone` https://github.com/IntelliSys-Lab/Stellaris-SC24.
- Go to the directory `Stellaris-SC24/evaluation` by `cd Stellaris-SC24/evaluation`.
- Install Docker container library by `./install_docker.sh`.
- Install NVIDIA CUDA driver by `./install_nvidia.sh`.
- Pull the Docker images directly from DockerHub by `cd docker && ./pull_docker.sh`. Note that there is a rate limit for image downloading per six hours.
- Start the container cluster using Docker Compose by `docker compose up -d`.
- Run `cd ../ && ./run_experiment.sh` to execute the Stellaris demo. The demo experiment may take up to 20 minutes to complete.

Alternatively, we also provide scripts that build Docker images locally by `cd docker && ./build_docker.sh`, but this can take a significant amount of time if built from scratch. Please refer to the GitHub repository for detailed instructions.

Artifact Analysis (incl. Outputs)

When `run_experiment.sh` finishes, you can check the results and figures of training efficiency and training cost under the directory `evaluation/experiment/figures`.

Expected results:

- Training efficiency. There should be three figures titled "timeline*.png", depicting the episodic rewards achieved by *Stellaris* and Ray RLlib on Hopper, Humanoid, and Walker2d environments, respectively. The results should match Figures 9(a), 9(b), and 9(c) in the paper that shows *Stellaris* achieves higher efficiency over Ray RLlib (the line of *Stellaris* is above Ray RLlib for most of the time).
- Training cost. One figure titled "cost.png" will also be generated under the directory, depicting the training cost of *Stellaris* and Ray RLlib, respectively. This result should match Figure 8, where *Stellaris* achieves a lower training cost than Ray RLlib (bars of *Stellaris* are shorter than Ray RLlib).