Technical Report: Parallel and Distributed Training of Proximal Policy Optimization (PPO)

Includes: A1 (Literature Survey), A2 (Problem Formulation), A3 (Initial Design)

Facing Sheet

Name	Roll Number	Contribution
BUDHBHATTI SAWAN	**2024aa05243**	**100% (Selection of the literature & review with others)**
CHAKKILAM VENKATA PARDHASARADHI	2024ac05021	100% (Exploring Problem Formulation & review with others)
TONY ABRAHAM MAMMEN	2024ac05596	100% (Exploring Problem Formulation & review with others)
CHAUDHARI PARESH DILIP	2023aa05696	100% (Initial Design Idea & review with others)
ASHWINI PRAKASH	2022ac05536	100% (Initial Design Idea & review with others)

Abstract

This report investigates Proximal Policy Optimization (PPO) as a stable and practical policy-gradient algorithm and details a scalable, distributed training design suitable for modern reinforcement learning workloads. A1 surveys PPO's core mechanisms, empirical performance, and its relationship to parallel RL systems (A3C, IMPALA, RLlib, DD-PPO). A2 formulates the problem of on-policy data-parallel scaling with precise metrics (speedup, scaling efficiency, throughput, communication overhead, convergence). A3 proposes a decentralized, synchronous PPO architecture (DD-PPO style) using gradient all-reduce, vectorized environments, and careful batching, along with risk mitigations and validation plans. The design aims to achieve near-linear throughput scaling while preserving on-policy correctness and PPO's clipped-objective stability.

Introduction

Reinforcement learning (RL) with deep neural networks faces two practical bottlenecks: training stability and wall-clock performance. Trust Region Policy Optimization (TRPO) addressed stability via KL-constrained updates but introduced second-order complexity [2]. Proximal Policy Optimization (PPO) simplifies this with a clipped surrogate objective and multi-epoch minibatch updates, achieving competitive stability and strong empirical results on Atari and MuJoCo [1]. Concurrently, parallel/distributed RL architectures such as A3C (asynchronous actor-learners) and IMPALA (decoupled actor-learner with V-trace corrections) demonstrated large throughput gains [3][4]. In practice, PPO is frequently combined with data-parallel rollouts and multi-GPU training (e.g., RLlib), and decentralized synchronous variants (DD-PPO) have shown near-linear scaling on large GPU clusters [5][7]. This report synthesizes these strands: A1 surveys PPO and related systems; A2 formalizes a data-parallel PPO scaling problem; A3 proposes a practical distributed design and validation plan.

A1: Literature Survey — Proximal Policy Optimization (PPO)

Background and Motivation. Policy-gradient methods optimize a parameterized policy to maximize expected return, but naive updates can be unstable and sample-inefficient. TRPO mitigates this via a KL-divergence trust region, providing robust monotonic improvement at the expense of second-order optimization [2]. PPO was introduced to capture TRPO-like stability using simpler first-order updates. Schulman et al. propose two variants—PPO-Clip (clipped probability ratio) and PPO-Penalty (adaptive KL)—and show strong empirical results across Atari and MuJoCo, with better sample reuse via multiple SGD epochs [1].

Core Algorithm and Variants. PPO-Clip limits the policy ratio r_t within $[1-\varepsilon, 1+\varepsilon]$ to prevent overly large updates; this enables multiple minibatch passes over the same on-policy trajectories while maintaining stability. PPO-Penalty adds an adaptive KL term to discourage large deviations from the old policy. Public implementations (OpenAI Spinning Up, RLlib, CleanRL, PyTorch RL) emphasize best practices: Generalized Advantage

Estimation (GAE), advantage normalization, entropy regularization, Atari preprocessing, and MPI-based parallelization for data collection [6][7].

Empirical Results and Benchmarks. The original PPO paper reports superior wall-time and strong returns versus online baselines, often matching or surpassing TRPO on Atari and MuJoCo [1]. Community reproductions corroborate the importance of implementation details (e.g., frame-stacking, reward clipping, value loss and entropy coefficients). Guides such as the ICLR blog on implementation details catalog dozens of small choices that materially affect performance [9].

Distributed/Parallel Context. PPO adapts naturally to synchronous data-parallel training: multiple workers collect on-policy rollouts, aggregate batches, and perform synchronized minibatch SGD. RLlib provides scalable PPO/APPO/DD-PPO implementations with knobs for EnvRunners and Learners [7]. DD-PPO demonstrates decentralized, synchronous training with near-linear scaling (e.g., ~107× on 128 GPUs) and no stale computation [5]. While IMPALA (off-policy) scales via decoupled acting/learning and V-trace corrections, its principles inform engineering choices for efficient transport and high throughput [4].

Open Challenges. PPO's on-policy nature limits reuse of past data; fixed clipping ranges may induce bias or under-utilize informative gradients in high variance regimes. Variants such as dimension-wise importance-weight clipping address high-dimensional action spaces, and adaptive clipping/KL strategies are active areas of exploration [8][6].

A2: Problem Formulation — Parallelization and Distributed Training of PPO

Problem Statement

PPO's stability and simplicity make it a strong baseline; however, single-machine training constrains wall-clock performance and simulation throughput for complex environments and larger models. We seek a data-parallel, distributed PPO system that accelerates on-policy rollout collection and learning while preserving policy consistency across workers and minimizing communication overhead.

Scope and Assumptions

We consider synchronous data-parallel PPO across W workers (CPUs/GPUs). Each worker collects on-policy rollouts with current parameters, aggregates experience into a global batch, and performs K epochs of minibatch SGD. After each update, parameters are synchronized (decentralized all-reduce or equivalent). Environments support vectorization; communication uses efficient collective backends (e.g., NCCL/MPI) [7][5].

Objectives

• Achieve near-linear wall-clock speedup as W increases (within system/network limits).

• Maintain sample efficiency and final returns comparable to single-node PPO.

• Minimize communication overhead for gradient aggregation/parameter sync.

• Ensure on-policy correctness (minimal staleness between rollout and update).

Formalization & Metrics

Let T_1 be time-to-target-return with 1 worker and T_W with W workers. Speedup S = T_1 / T_W; scaling efficiency E = S / W. Track throughput (frames per second, FPS), communication vs. compute time per iteration, and convergence quality (average episodic return across seeds).

Constraints

• On-policy requirement: trajectories must match the latest policy; stale data harms performance.

• Synchronous barriers risk stragglers; network latency/bandwidth can dominate beyond threshold W.

• Large global batches alter optimization dynamics; clip range/entropy/GAE may require retuning at scale.

Evaluation Plan

Run controlled experiments on standard benchmarks (Atari, MuJoCo) with W ∈ {1, 2, 4, 8, 16, 32}. Fix per-update environment steps per worker to keep total batch comparable; measure FPS, S, E, and time-to-target-return. Profile iteration time into sampling, optimization, and communication. Compare parameter-server vs. decentralized all-reduce, and PPO-Clip vs. PPO-Penalty [6][7][5].

A3: Initial Design — Distributed PPO

Design Goals & Principles

• Throughput & Speedup: near-linear scaling in FPS and reduced time-to-target-return (DD-PPO/RLlib evidence).

• On-policy Correctness & Stability: preserve PPO-Clip/GAE stability with fresh trajectories.

• Simplicity: prefer synchronous decentralized all-reduce over parameter servers to avoid staleness/bottlenecks.

High-Level Architecture

Synchronous, decentralized data-parallel PPO (DD-PPO style).
EnvRunners (rollout workers): each hosts E vectorized environments to collect on-policy batches with current parameters.
Learners: one per node (or colocated) participate in gradient all-reduce; optimizer step

applied synchronously across replicas (no parameter server).
Parameter Sync: weights are consistent after each update via all-reduce or explicit all-gather; logging via central tracking service (FPS, speedup, KL, clip fraction). [5][7]

Control Flow (Per Iteration)

• Sync: all learners start with identical parameters θ_k.

• Rollout: each EnvRunner collects T steps from E envs → local batch; global batch B aggregated across W workers.

• Compute GAE(λ)/returns locally with normalization.

• SGD: shuffle B, split into M mini-batches; compute gradients; all-reduce; apply optimizer step synchronously.

• Diagnostics: track approx-KL and clip fraction; adjust LR/epochs if divergence spikes.

• Logging/Eval: periodic evaluation episodes; record FPS, time breakdown (sampling/SGD/comm).

Data & Batch Sizing Strategy

Per-worker rollout T: 128–2048 steps (shorter for Atari, longer for MuJoCo). Total batch B = W × E × T. Start with constant B across scaling sweeps; then increase B after stability validation. Mini-batches M ∈ [4,16]; epochs K ∈ [3,10]; normalize advantages and use domain-specific preprocessing (Atari reward clipping/frame-stacking). [6]

Communication Pattern & Backends

Use gradient all-reduce (NCCL for GPUs; Gloo/MPI fallback). Overlap communication with compute via bucketed gradients where possible. Colocate EnvRunners/Learners to reduce copies. Parameter servers are avoided due to central bottlenecks and staleness. [7][5]

Algorithmic Details (PPO Core)

Optimize PPO-Clip objective with ε ≈ 0.1–0.2; include entropy bonus and value loss; use GAE(λ). Monitor approx-KL and clip fraction each iteration as diagnostics. Optional PPO-Penalty for adaptive KL control; consider dimension-wise IS-weight clipping for high-dimensional actions. [1][6][8]

Resource Plan & Deployment

Cluster: N nodes × G GPUs (e.g., 4–8 GPUs/node); set W = N×G learners; each learner runs E envs to saturate device. Orchestrate with Ray/RLlib, SLURM, or Kubernetes; mixed precision for large models; vectorized envs for efficiency. [7]

Hyperparameter Defaults

Clip ε=0.2, γ=0.99, λ=0.95, entropy coef=0.01–0.02, value coef=0.5; Adam optimizer with LR warmup or constant LR as per baselines; T=128 (Atari) / 2048 (MuJoCo); K=4; M=4–8. Adjust via KL/clip diagnostics. [1][6]

Measurement & Instrumentation

Track global FPS, speedup S, efficiency E, time breakdowns, and learning diagnostics (returns, KL, clip fraction, value loss, entropy). Ensure reproducibility (fixed seeds, deterministic wrappers) and detailed logging of hyperparameters and revisions. [6][7][1]

Risk Analysis & Mitigations

• Communication bottlenecks → bucketed all-reduce; hierarchical reduction; tune W per node.

• On-policy staleness → strict synchronous iteration; cap rollout horizon; avoid async gradients unless APPO is intended.

• Large batch instability → reduce LR; adjust epochs/mini-batches; use PPO-Penalty; monitor KL targets.

• Env heterogeneity/stragglers → balance env assignment; increase vectorization; set timeouts.

• High-dimensional action bias → dimension-wise IS clipping; tune ε per domain.

Alternatives Considered

Parameter Server PPO (simpler but bottleneck-prone), Asynchronous PPO/APPO (higher utilization but policy lag), IMPALA-style off-policy actor-learner (excellent scale; off-policy corrections; different algorithmic goal). [7][4]

Validation Plan

Sanity: $W \in \{1,2,4\}$ small envs; match baseline curves (returns, KL/clip). Scaling sweeps: $W \in \{1,2,4,8,16,32\}$ across Atari/MuJoCo; measure FPS, S, E, time-to-target. Ablations: constant vs. growing B; parameter server vs. all-reduce; PPO-Clip vs. PPO-Penalty. Stress tests for stragglers and uneven env complexities. [6][7][5]

Conclusion

PPO's clipped objective and minibatch updates provide a robust foundation for on-policy RL. By adopting a decentralized, synchronous design (DD-PPO style) with gradient all-reduce, vectorized environments, and careful batching, we can scale PPO to multi-GPU/multi-node clusters while preserving stability and on-policy correctness. The proposed design, metrics, and validation plan target near-linear speedup and reproducible performance on standard benchmarks, while risk mitigations and alternatives provide practical pathways for adaptation to diverse workloads.

References

[1] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv:1707.06347.

[2] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., Abbeel, P. (2015). Trust Region Policy Optimization. arXiv:1502.05477.

[3] Mnih, V., Badia, A. P., Mirza, M., et al. (2016). Asynchronous Methods for Deep Reinforcement Learning. ICML 2016 / arXiv:1602.01783.

[4] Espeholt, L., Soyer, H., Munos, R., et al. (2018). IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. arXiv:1802.01561.

[5] Wijmans, E., Kadian, A., Morcos, A., et al. (2020). Decentralized Distributed PPO (DD-PPO). ICLR 2020 / arXiv:1911.00357.

[6] OpenAI Spinning Up — PPO documentation and reference implementations (accessed).

[7] Ray RLlib — PPO/APPO/DD-PPO documentation and scaling guides; RLlib ICML 2018 systems paper.

[8] Han, S., Sung, Y. (2019). Dimension-Wise Importance Sampling Weight Clipping for Sample-Efficient RL. ICML 2019.

[9] Huang, S., Dossa, R., Raffin, A., Kanervisto, A., Wang, W. (ICLR Blog Track, 2022). The 37 Implementation Details of PPO.