Extended Abstract

Motivation Chain-of-thought reasoning has proven effective for enhancing language model capabilities, but its inherently serial nature limits efficiency on complex reasoning tasks. This limitation becomes critical on benchmarks like ARC-AGI, where reasoning chains require minutes of sequential computation. The inefficiency is especially pronounced in problems with large generation-verification gaps—while solving a maze requires extensive exploration, verifying a solution is trivial. This asymmetry extends to mathematical proofs, puzzles, and code verification. Techniques like Majority@N sampling demonstrate that even simple parallel inferencing strategies work. Our question is whether models can learn more complex parallel inferencing strategies that involve planning and task distribution of tasks through policy-gradient style RL training.

Method We propose Hydra, a training method that enables language models to perform parallel reasoning through dynamic thread spawning and coordination. Our approach introduces `<fork budget=X>["task1", "task2", ...]</fork>`, allowing models to decompose problems into sub-tasks, solve them concurrently, and synthesize results. We employ Group Relative Policy Optimization (GRPO) to optimize models on verifiable rewards across parallel reasoning trajectories. GRPO leverages relative performance within trajectory groups to provide baselines, naturally handling the complex structures that emerge from fork-join operations.

Implementation We extend the TRL framework with distributed training capabilities using torch.distributed for multi-node scaling and integrate vLLM as the inference engine during GRPO training. The off-policy collection stage involves gathering prompts, generating parent reasoning with vLLM, detecting fork calls through custom stop tokens, and spawning child processes. The engine constructs result JSONs from child outputs and injects them into the parent's context. During online learning, the policy model computes the whole batch in a DDP fashion across GPUs, and we all-gather the results to compute the GRPO advantages before scattering the tensors for computing the backward passes.

Results Our experiments targeted mathematical reasoning on the Countdown dataset, with a 2048 serial tokens budget per problem. We qualitatively discuss the results, and show a curve of an early training run.

Discussion Our work focuses on the early observations and challenges in training parallel reasoning models. The engineering complexity of coordinating high-performance inference servers with distributed training frameworks while managing dynamic thread spawning was quite challenging to us. In addition, prompt engineering was complex, as we needed to clearly communicate the existence of two distinct roles for the LLMs.

Conclusion Hydra represents an attempt to reduce the wall-clock time of chain-of-thought reasoning through allowing models to dynamically reasoning in parallel. While full validation remains incomplete, we make an attempt towards this direction, and observe hints that with our inference pattern and system prompt construction alone, the models already display signs of being able to split their work. Our contributions include: (1) a parallel reasoning paradigm using fork/join primitives, (2) TRL framework extensions for distributed parallel reasoning optimization, and (3) a distributed training infrastructure for multi-turn, parallel reasoning. Code available at `https://github.com/punwai/parallel`.

Hydra: Training End-to-End Parallel Reasoners

Suppakit Waiwitlikhit
Department of Computer Science
Stanford University
suppakit@stanford.edu

Abstract

Chain-of-thought reasoning has proven effective for enhancing language model capabilities, but its inherently serial nature limits efficiency on complex reasoning tasks. We propose Hydra, a training method that enables language models to perform parallel reasoning through dynamic thread spawning and coordination. Our approach introduces fork and join operations that allow models to decompose problems into sub-tasks, solve them concurrently, and synthesize results. We extend the TRL framework with our parallel reasoning training methodology, implementing Group Relative Policy Optimization (GRPO) to optimize models on verifiable rewards across parallel reasoning trajectories. Our implementation leverages torch.distributed for multi-node scaling during initial testing phases and integrates vLLM as the inference engine during GRPO training to efficiently handle the increased computational demands of parallel rollouts. Our key contributions include: (1) a parallel reasoning paradigm for language models using fork/join primitives, (2) an extension of the TRL framework supporting distributed parallel reasoning optimization, (3) a distributed training infrastructure built on torch.distributed for multi-node training integrated with vLLM for scalable inference during training.

All code is open-sourced at ‘https://github.com/punwai/parallel‘

1 Introduction

Recent works on chain-of-thought prompting have led to groundbreaking improvements in model capabilities. However, this approach inherently constrains reasoning to a serial process, where each thought must follow sequentially from the previous one. On challenging benchmarks such as ARC-AGI, this serialization can lead to reasoning chains that require minutes of sequential computation before arriving at a solution.

This limitation is particularly pronounced in problems where the generation-verification gap is large. Imagine trying to solve a maze - while generating a correct solution may require extensive search through the complex solution space of a maze, verifying the correctness of a proposed solution is often computationally trivial. Such gap extends to domains like mathematical proofs can be checked algorithmically, and puzzle solutions can be verified instantly. This asymmetry suggests that reasoning should involve parallel exploration of multiple candidate solutions rather than committing to a single sequential path. The demonstrated effectiveness of techniques like Majority@N sampling provides empirical evidence for this intuition: diverse reasoning trajectories, when properly combined, can significantly outperform single-threaded approaches within the same computational budget, precisely because verification allows us to identify and select the best among many parallel attempts.

This observation motivates the following research question: can we train language models to reason in parallel? Multiple reasoning threads to explore different aspects of a problem simultaneously? We know that human-coded heursitics like majority@N, but if we give the models free reign to

parallelize themselves dynamically, learning through experience the best parallelism strategies to tackle a problem, what strategies will they come up with?

We believe that this problem is well-suited to policy-gradient based methods. Although current models may exhibit the ability to communicate parallelism patterns, there is fundamentally limited amounts of data in pre-training datasets that displays how to collaborate on problems. One reason for this is that humans rarely can collaborate in a manner in which communication occurs frequently, as LLMs do.

In this work, we introduce Hydra, an end-to-end reinforcement learning algorithm for training language models to perform dynamic parallel reasoning through learned fork and join operations. Our approach allows models to decompose complex problems into sub-tasks, solve them concurrently across multiple threads, and synthesize the results into coherent solutions. Unlike existing parallel reasoning methods, Hydra enables child threads to inherit full parent context while maintaining the flexibility to dynamically determine the appropriate level of parallelism for each problem instance.

We implement our approach by extending the TRL framework with distributed training capabilities and integrating vLLM for efficient inference during Group Relative Policy Optimization (GRPO). Our system leverages torch.distributed to scale across multiple nodes, enabling experimentation with parallel reasoning at scale. While our initial experimental results are preliminary due to ongoing implementation challenges, early observations demonstrate that models can successfully learn to utilize fork and join operations, suggesting the fundamental viability of our approach.

2 Related Work

2.1 Inference-Time Scaling and Chain-of-Thought Reasoning

The scaling of inference-time compute through chain-of-thought reasoning has emerged as a powerful paradigm for enhancing language model capabilities. By encouraging models to articulate their reasoning process step-by-step, chain-of-thought prompting has unlocked significant performance improvements across diverse reasoning tasks. This approach has been further enhanced through techniques like self-consistency decoding, where multiple reasoning paths are sampled and the most consistent answer is selected. However, these methods remain fundamentally constrained by their serial nature. Each reasoning step must be generated sequentially, limiting the ability to explore multiple solution strategies simultaneously. Recent work on inference-time scaling has begun to explore alternatives, including tree-of-thought reasoning and other structured approaches to problem decomposition, but these still operate within the constraint of sequential generation.

2.2 Parallel Inference Techniques

Several recent works have explored methods for parallelizing inference-time computation in language models. Pan et al. introduced Adaptive Parallel Reasoning (APR), which enables models to dynamically call "fork" and "join" operations to spawn new threads for solving sub-tasks. APR demonstrated a 24HogWild! Inference proposed an alternative approach using static allocation of multiple language model threads that communicate through a shared scratchpad. This method computes the KV-cache of the shared scratch space only once, improving efficiency. However, static allocation prevents dynamic adaptation of parallelism levels to task complexity, requiring manual specification by end users. These approaches demonstrate the potential of parallel reasoning but highlight key limitations that our work aims to address: the need for better context sharing mechanisms and dynamic parallelism adaptation.

2.3 Reinforcement Learning for Reasoning

Group Relative Policy Optimization (GRPO) has emerged as an effective method for optimizing language models on tasks with verifiable rewards. Unlike approaches such as PPO that require training separate verifier models, GRPO leverages relative performance within groups of sampled trajectories to provide baselines. This approach is particularly well-suited for reasoning tasks where external verification is available through unit tests, mathematical proof checkers, or puzzle solvers. GRPO operates by sampling multiple reasoning trajectories for each problem, evaluating their correctness through verification, and optimizing the policy to increase the likelihood of correct solutions relative to incorrect ones within the same group. This framework provides a natural foundation for optimizing

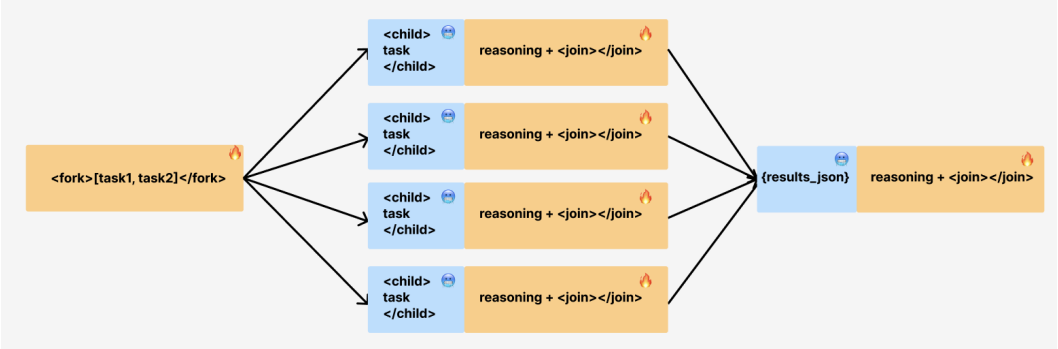

Figure 1: Hydra's computation graph. Blue indicates that the text is not model generation, and therefore are not optimized by the RL algorithm

parallel reasoning, as it can handle the complex trajectory structures that emerge from fork-join operations.

2.4 Distributed Training for Language Models

The scale of modern language model training has necessitated sophisticated distributed training frameworks. Tools like the TRL (Transformer Reinforcement Learning) library provide high-level abstractions for training language models with reinforcement learning, while frameworks like torch.distributed enable scaling across multiple nodes and devices.

However, existing frameworks are primarily designed for traditional serial generation patterns. Extending these systems to handle the complex synchronization and communication requirements of parallel reasoning presents significant engineering challenges, particularly around ensuring efficient co-location of related computations and managing the variable-length nature of parallel reasoning trajectories. Working with DDP is non-trivial when working with a high-performance language modeling server such as vLLM/SGLang, as we will manually have to do all-reduces and all-scatters when switching between performing inference on the high-throughput inference server to collect the trajectories, and performing inference on the torch to collect log probabilities of the online policy.

3 Method

3.1 Fork-Join Tool Calling

We introduce a set of tool calls, `fork` and `join` which serve as an interface for models to distribute tasks, and control the flow of parallel reasoning execution. The `fork` tool enables dynamic spawning of parallel reasoning threads that can work concurrently on different aspects of a problem before rejoining to synthesize results. We formally define the `fork` tool as follows:

$$\text{fork} : (\text{budget}, [task_i]i = 1^N) \rightarrow ([response_i]i = 1^N) \tag{1}$$

Upon invoking this tool, the parent process halts execution and spawns N child worker threads. The child workers will perform inference until they hit the set `budget`, an argument of the `fork` function, or until they call `join` to return the control flow back to the parents.

3.2 Parent-Child Mode Separation

In order to infuse the models with sufficient prior to use the tools, we introduce to the models the concept of two distinct operational modes, `parent mode` and `child mode` through clear instructions given in the system prompt. We decide to include the description of the two distinct modes through the system prompt for two reasons.

1. We want as much explanation of the tool's expected behavior to exist in the system prompt as much as possible, to minimize explanation metadata inserted into the model during the rollouts, which may confuse the models.
2. We want parent and child trajectories to share the greatest amounts of prefix as possible to take advantage of KV-caching, and avoid wasteful recomputation of logits.

While the model is in `child mode`, it is instructed to always end its response with the `join` tool. After a parent invokes `fork`, a child is spawned by appending the `<child>task</child>` metadata to inform the child of the specific task that they are instructed to complete.

3.3 Formal Thread Definition and GRPO Optimization

We now rigorously define the parallel reasoning process and optimization objective. Let x denote the initial problem prompt. The execution proceeds through the following stages:

3.3.1 Parent Thread Execution

The parent thread begins with prompt x and generates:

1. **Initial reasoning**: $z_{\text{reason},1} \sim p_\theta(z_{\text{reason},1} \mid x)$
2. **Task decomposition**: $z_{\text{tasks}} \sim p_\theta(z_{\text{tasks}} \mid x, z_{\text{reason},1})$

where z_{tasks} represents the fork call with specific sub-tasks: $z_{\text{tasks}} = \texttt{<fork>}[task_1, task_2, \ldots, task_N]\texttt{</fork>}$.

3.3.2 Child Thread Execution

Upon forking, N child threads are spawned. Each child thread i inherits the complete parent prefix and generates reasoning based on its assigned task:

$$z_{\text{child},i} \sim p_\theta(z_{\text{child},i} \mid x, z_{\text{reason},1}, z_{\text{tasks}}, \texttt{<childtask>}task_i\texttt{</childtask>}) \tag{2}$$

Each child thread produces reasoning $z_{\text{child},i}$ followed by a join call containing its solution.

3.3.3 Result JSON Injection

After all child threads complete, our execution engine constructs a result JSON object R that summarizes the child outputs:

$$R = \text{jsonformat}([z_{\text{child},i}]_{i=1}^{N}) \tag{3}$$

Crucially, R is **not** directly generated by the language model and therefore **not** included in the optimization objective. Instead, R is deterministically constructed by the engine and injected into the parent's context.

3.3.4 Parent Post-Join Reasoning

The parent thread resumes execution with the injected result JSON and generates final reasoning:

$$z_{\text{reason},2} \sim p_\theta(z_{\text{reason},2} \mid x, z_{\text{reason},1}, z_{\text{tasks}}, R) \tag{4}$$

Note that $z_{\text{reason},2}$ conditions on the engine-generated R rather than the raw child outputs $[z_{\text{child},i}]_{i=1}^{N}$.

3.3.5 GRPO Loss Formulation

For optimization, we apply GRPO across all model-generated tokens in the parallel execution. Let r denote the reward obtained through external verification. The GRPO loss is defined as:

$$\begin{aligned}\mathcal{L}_{\text{GRPO}} = [&\log p_\theta(z_{\text{reason},1} \mid x) + \log p_\theta(z_{\text{tasks}} \mid x, z_{\text{reason},1}) \\ &+ \sum_{i=1}^{N} \log p_\theta(z_{\text{child},i} \mid x, z_{\text{reason},1}, z_{\text{tasks}}, \texttt{<childtask>}task_i\texttt{</childtask>}) \\ &+ \log p_\theta(z_{\text{reason},2} \mid x, z_{\text{reason},1}, z_{\text{tasks}}, R)] \cdot \frac{r - \mu}{\sigma}\end{aligned} \tag{5}$$

where μ and σ are the group mean and standard deviation of rewards respectively. The result JSON R appears in the conditioning context for $z_{\text{reason},2}$ but is **not** included in the log-probability terms being optimized, as it is deterministically generated by the engine rather than sampled from the model.

3.4 Implementation Details

Our implementation extends the TRL framework with distributed training capabilities using `torch.distributed` for multi-node scaling. We integrate vLLM as the inference engine during GRPO training, with custom modifications to support fork detection through specialized stop tokens. The system performs the following steps:

Off-policy Collection Stage

1. **Prompt Collection**: Process-0 gathers all prompts via distributed communication
2. **Initial Generation**: vLLM generates parent reasoning and task decomposition
3. **Fork Detection**: Custom stop token detection identifies fork calls
4. **Child Spawning**: New vLLM processes are launched for child thread generation
5. **Result Aggregation**: Engine constructs result JSON R from child outputs
6. **Parent Resumption**: Parent continues with R injected into context
7. **Advantage Computation**: GRPO advantage calculated for each trajectory.
8. **Offline Log-Probabilities**: Collection model computes log-probabilities for all model-generated tokens in each trajectory

Online Learning Stage

1. **Online Log-Probabilities**: Current policy model computes log-probabilities for the same tokens. This is done using torch, distributed across 4 GPUs using $torch.distributed$.
2. **GRPO Loss**: Final loss computed using Equation (5) with KL regularization
3. **Gradient Update and synchronization**: Policy parameters θ updated via backpropagation. Handled fully by TRL's existing infrastructure.

4 Experimental Setup

For the experiments, we aim to run 200 steps of GRPO on a Qwen-1.7B model using our parallel reasoning framework. Our planned experimental configuration includes training on mathematical reasoning tasks from the Countdown dataset, where models must explore multiple solution paths within a fixed token budget of 5120 tokens per problem. We compare our parallel reasoning approach against a serial baseline using the same token budget to evaluate the effectiveness of dynamic thread spawning and synthesis.

In addition, we also plan to evaluate the model on MATH after RL on Countdown to investigate whether. We serve to investigate whether performing RL to learn parallelism on one narrow domain would result in generalization of parallelism capabilities to other domains.

5 Results

Disappointingly, we just got our distributed training infrastructure to work, due to the difficulty of debugging the distributed data parallel (DDP) implementation on top of the TRL infrastructure, which is required for training even a 1.7B model. The integration of vLLM with custom fork detection, multi-process spawning, and GRPO optimization has proven technically challenging, resulting in frequent crashes during training runs. As a result, we are unable to obtain the full training curve at this time, though we continue to work on stabilizing the distributed training infrastructure.

As a small display of our progress, we provide a short hundred-step reward curve that we have of the parallel reasoner on the Countdown task. We also offer some qualitative observations of the traces, and our learnings for the prompt to give the models.

Figure 2: Early reward curve for Hydra run

5.1 Qualitative Analysis

We discuss some learnings of the traces. First, we observed that models struggle to understand the distinction between parent and child modes, often attempting to fork from within child threads or continuing to generate content after issuing a fork command. This suggests that the mode-switching paradigm requires careful prompt engineering and potentially architectural modifications to make the distinction more explicit. While we currently maintain the same system prompts for parent and child modes, it may be beneficial to have a separate system prompt. We maintain two system prompts because we wanted to ensure that prefix-caching works in this method, however, one can design custom attention masks so that altering the system prompt on-the-fly would not result in the re-computation of the prefix cache.

Second, we found that the base model already exhibits a certain baseline ability to split tasks for the Countdown task. For example, the models would instruct their child tasks to explore computations that involve a certain kind of operand such as additions or multiplications.

Finally, we observed that the quality of child thread is heavily determinant on the parent thread's formulation the sub-tasks. Overly specific reasoning task descriptions such as compute two numbers led to child reasoning traces that were not useful. However, we did observe that vague reasoning traces such as "combine the numbers W, X, Y, Z to produce the results" led to successful results, as each of the child independently explores different search spaces. We think that it will be interesting to observe whether over the course of training, these vague patterns are favored more or less compared to specific task descriptions that encourages each child to reason towards different directions.

6 Conclusion

We presented Hydra, an end-to-end algorithm and infrastructure for training parallel reasoning in language models, through giving models control of how they express parallelism. We made three key contributions. Despite incomplete results due to infrastructure difficulties, our work demonstrates that models exhibit initial baseline levels of chain-of-thought reasoning, and that early training results in hillclimbing. Future work should focus on stabilizing the infrastructure, conduct a training run with more compute, and compare the results between an optimized parallel reasoner and a optimized serial reasoner who are given similar token budgets to operate with.

7 Team Contributions

- **Suppakit Waiwitlikhit** Implemented the custom GRPO and parallel inference framework logic, training harness, data loader. Started debugging

Changes from Proposal We now start with Qwen3-1.7B due to compute constraints. We realized that RL requires quite a lot of GPU memory, and while our parallel inference training would accelerate inference, it ultimately consumes more FLOPs, resulting in a greater wall clock training time.

References

A Appendix A: Prompt

PROMPT_TEMPLATE = """ **PARENT MODE**:

- You start in Parent Mode.
- Decide whether splitting the job is worthwhile.
- To spawn workers, emit ONLY this tag and nothing else: <fork budget=X>["task 1", "task 2", . . .]</fork> – X = MAX tokens each child may use for its entire reply – Each item in the JSON list is a plain-text instruction for one child – **STOP immediately after . Do not continue writing.** – IMPORTANT: You may only fork 4 children at a time. Any child beyond the 4th will be ignored.
- Once the children have finished their tasks, the following tag will be appended to the conversation: [{{"child task": "task 1", "child response": "answer 1"}}, {{"child task": "task 2", "child response": "answer 2"}},] **Once you get the responses. Note that you are now switched back into parent mode.** Once you switch back into this tag it is important that:
  - Do NOT call join (you are now in parent mode)
  - Do NOT fork again (we only allow one fork)
  - Use child responses to formulate your final answer.
  - End your response with tags.

PARENT MODE EXAMPLE: I need to find an equation using [2, 3, 5, 7] that equals 24. This is complex - I should try different operation combinations. Let me split this into subtasks:

1. Try multiplication-based approaches
2. Try addition/subtraction combinations
3. Try division-based approaches
4. Try mixed operations with parentheses

["Using numbers 2, 3, 5, 7, find equations that equal 24 using primarily multiplication", "Using numbers 2, 3, 5, 7, find equations that equal 24 using addition and subtraction", "Using numbers 2, 3, 5, 7, find equations that equal 24 using division", "Using numbers 2, 3, 5, 7, find equations that equal 24 using mixed operations with parentheses"]

CHILD MODE:

- You are switched into the child mode when the following is appended to the conversation: Subtask that the parent has given you to solve. Once you see this, you are in child mode. You can proceed to solve the subtask in childmode going forward. When you are in child mode, you MUST think through the subtask, and return the answer in tags.
- In child mode you MUST:
  1. NOT fork (no nested forking allowed)
  2. Stay within X-token budget
  3. End with ONLY: Your result
  4. **STOP immediately after . Do not continue writing.**

CHILD MODE EXAMPLE: Using numbers 2, 3, 5, 7, find equations that equal 24 using primarily multiplication I'm in child mode. My task is to find equations using multiplication. Let me try different combinations:

- 2 * 3 = 6, then 6 * 5 = 30, too high
- 3 * 5 = 15, need to add 9 more... 7 + 2 = 9, perfect!
- 3 * 7 = 21, need 3 more... can't make 3 with 2 and 5
- 5 * 7 = 35, too high even before using other numbers So 3 * 5 + 7 + 2 = 24 works! 3 * 5 + 7 + 2 = 24

YOUR TASK: Create an equation using numbers {numbers} that equals {target}.

- Use basic arithmetic operations (+, -, *, /)
- Each number can be used ONLY ONCE
- Show work in tags
- You MUST use at least once during thinking
- Final answer must be in: (1 + 2) / 3

EXECUTION ORDER:

1. Start thinking in tags
2. Fork subtasks (STOP after) 2.5 if you are a child, solve the subtask, and return the answer in tags.
3. Wait for
4. Use child responses to formulate final answer
5. Provide answer in tags.

/think """