



JOHNS HOPKINS

WHITING SCHOOL
of ENGINEERING

How can language models dynamically allocate resources for parallel thinking?

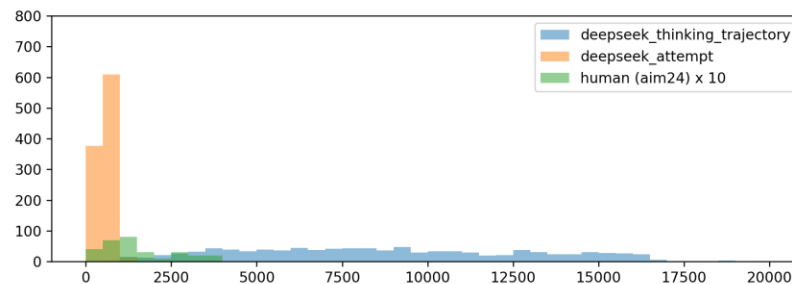
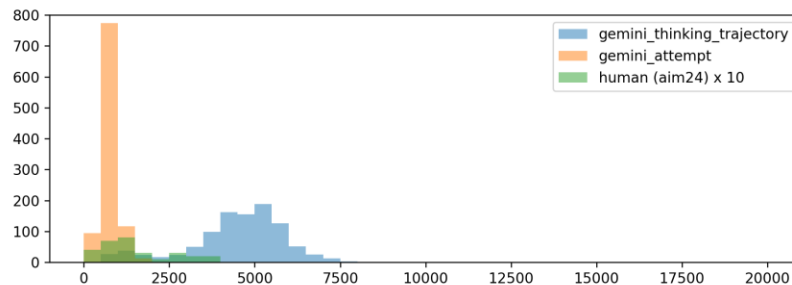
Wufei Ma and Jalil Rezek

Nov 11, 2025

Inference-Time Computation

Single model:

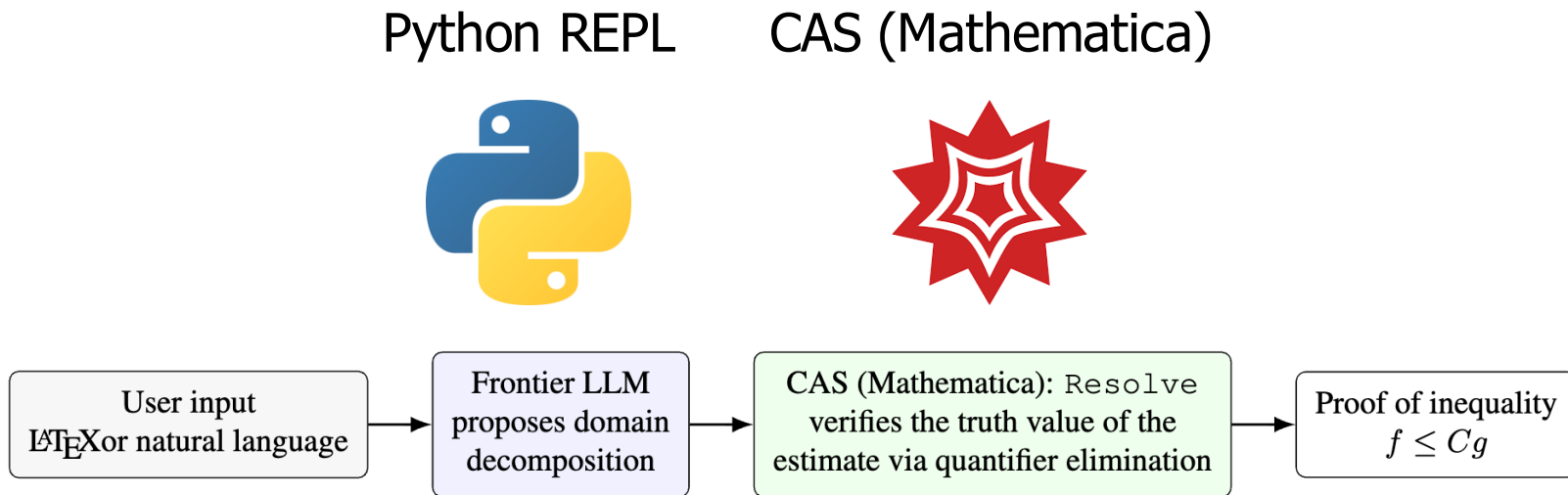
- Data: s1k-1.1 (math reasoning)
- Top: Gemini; bottom: deepseek.
- Orange: attempt; blue: thinking trajectory; green: human baseline.



[1] Muennighoff et al. s1: Simple test-time scaling.

Inference-Time Computation

- Tool use (math reasoning & proof)

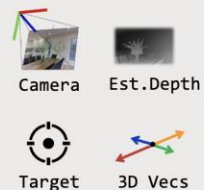


Inference-Time Computation

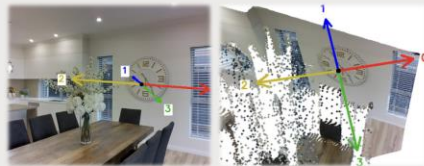
- Tool use (multi-modal)

Target: Direction of facing away from the wall in the clock area.

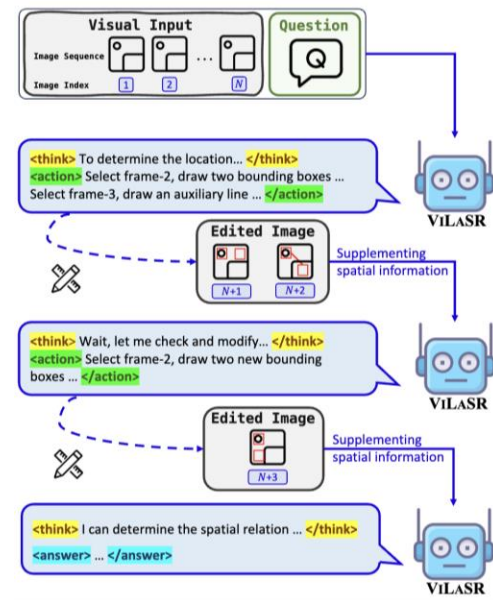
Renderer



1 Input: `<img_coarse><prompt>`
 Output: `<Reasoning> Vector3.`
`<Render around vector3>`



2 Input: `<img_fine><prompt>`
 Output: `<Reasoning> Vector1`
 Return: `<x1, y1, z1>`



Inference-Time Computation

■ Multi-agent debate

User Input: A treasure hunter found a buried treasure chest filled with gems. There were 175 diamonds, 35 fewer rubies than diamonds, and twice the number of emeralds than the rubies. How many of the gems were there in the chest?

Round 1

Agent 1: Let x be the number of rubies. Then the number of diamonds is 175. And the number of emeralds is $2(x-35)$, since there are twice as many emeralds as rubies and there are 35 fewer rubies than diamonds.

To find the total number of gems, we add the number of diamonds, rubies, and emeralds: Total gems = $175 + x + 2(x-35) = 3x + 105$.

We don't know the exact value of x , but we do know that it represents the number of rubies, which is less than 175 (since there are 35 fewer rubies than diamonds). Let's try some values of x : If $x = 50$, then the total gems = $3(50) + 105 = 255$. If $x = 100$, then the total gems = $3(100) + 105 = 405$.

Since we know there were fewer rubies than diamonds, the answer must be closer to 255 than to 405. After some trial and error, we can find that $x = 40$ gives us: Total gems = $3(40) + 105 = 225$. ❌

Agent 2: If there were 175 diamonds, then the number of rubies is 35 fewer, or $175 - 35 = 140$ rubies.

And since there were twice as many emeralds as rubies, we can represent the number of emeralds as $2 * 140 = 280$ emeralds.

To find the total number of gems in the chest, we can add up the number of diamonds, rubies, and emeralds: $175 + 140 + 280 = 595$. Therefore, there were 595 gems in the chest. ✅

User Input: Using the solutions from other agents as additional information, can you give an updated response...

(Copy Agent 2's reply)

(Copy Agent 1's reply)

Round 2

Agent 1: Given the information provided in other agents' solutions, we have two answers: One agent found that the number of rubies is 40 and the total number of gems is 225. Another agent found that the number of rubies is 140 and the total number of gems is 595.

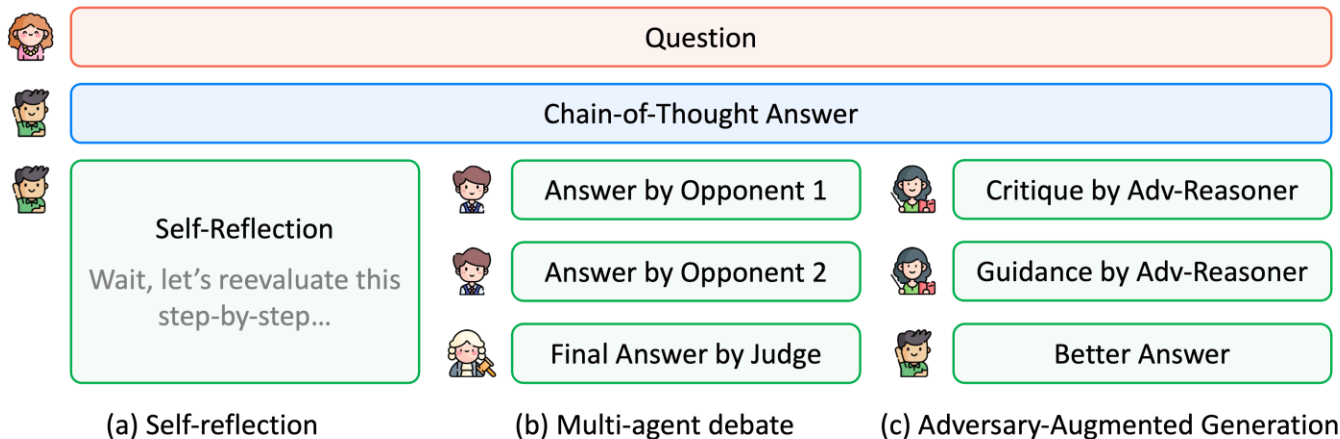
We know that the number of rubies is 35 fewer than the number of diamonds, which is 175. So the number of rubies should be $175 - 35 = 140$, as the second agent found. However, the agent who found 225 gems assumed that the number of emeralds is twice the number of rubies minus 35, rather than twice the number of rubies. If we use this assumption, we get: Number of rubies = 140. Number of emeralds = $2 * 140 - 35 = 245$. Total number of gems = $175 + 140 + 245 = 560$. Our final answer is 560. ❌

Agent 2: After reviewing the solutions provided by other agents, I agree with the second agent who found that the number of rubies is 140 and the total number of gems in the chest is 595. Therefore, my answer to the math problem is 595. ✅

User Input: Using the solutions from other agents as additional information, can you give an updated response...

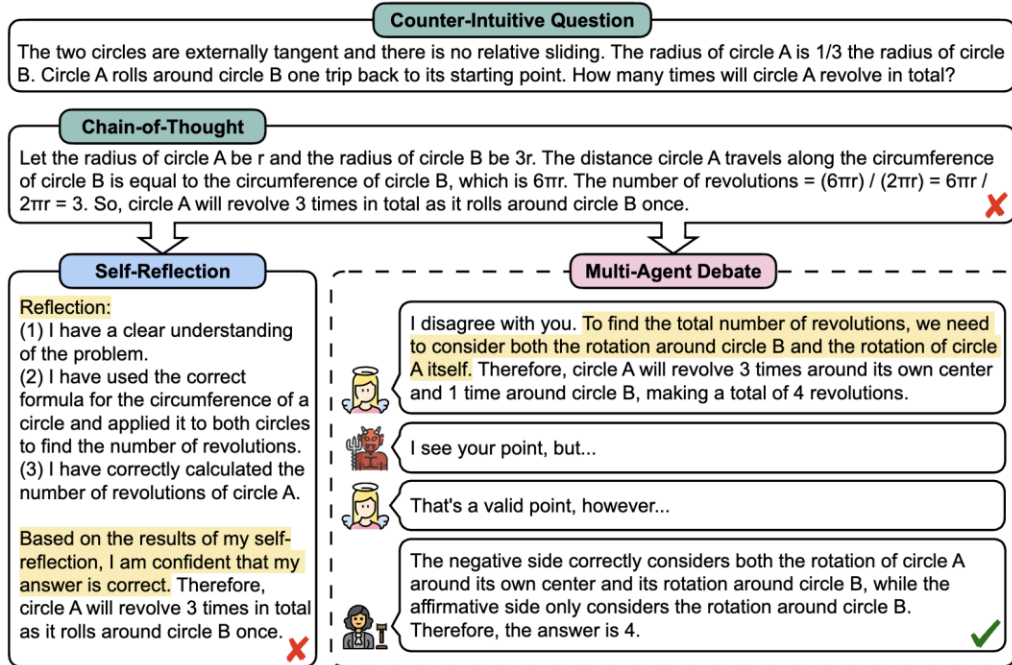
Inference-Time Computation

- Guidance from judger



Inference-Time Computation

■ Divergent thinking



Challenge

- Very long reasoning at test-time.
 - Self-reflection, tool-use, multi-agent, etc.
- One solution: parallelism

Parallelism

- Saves memory
- Running larger models
- Bigger batch size
- Improving latency or throughput

Data Parallelism

Model (Tensor) Parallelism

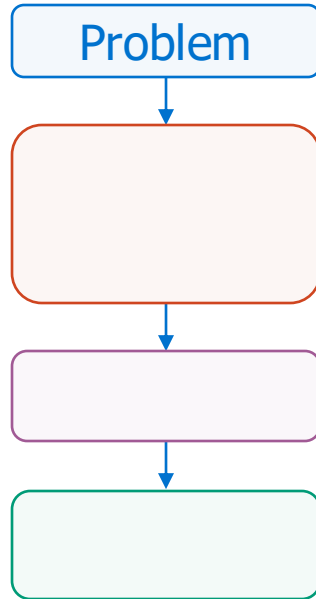
Pipeline Parallelism

Sequence Parallelism

MoE Parallelism

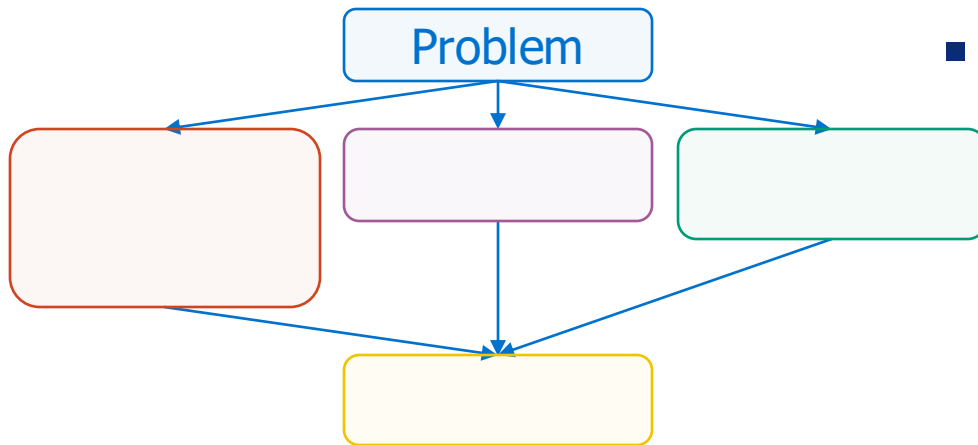
Task (Thinking) Parallelism

Challenge



- **Sequential**

Challenge

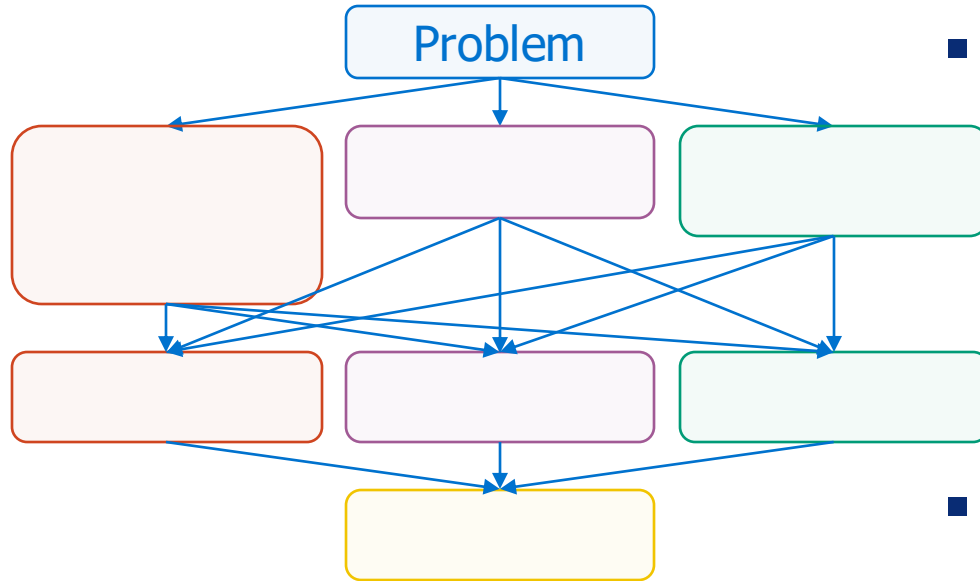


- **Well-defined sub-tasks**

- Synthetic datasets where each sample consists of 5 math questions
- Real-world problems where we consider different conditions?

- Nice parallelism

Challenge

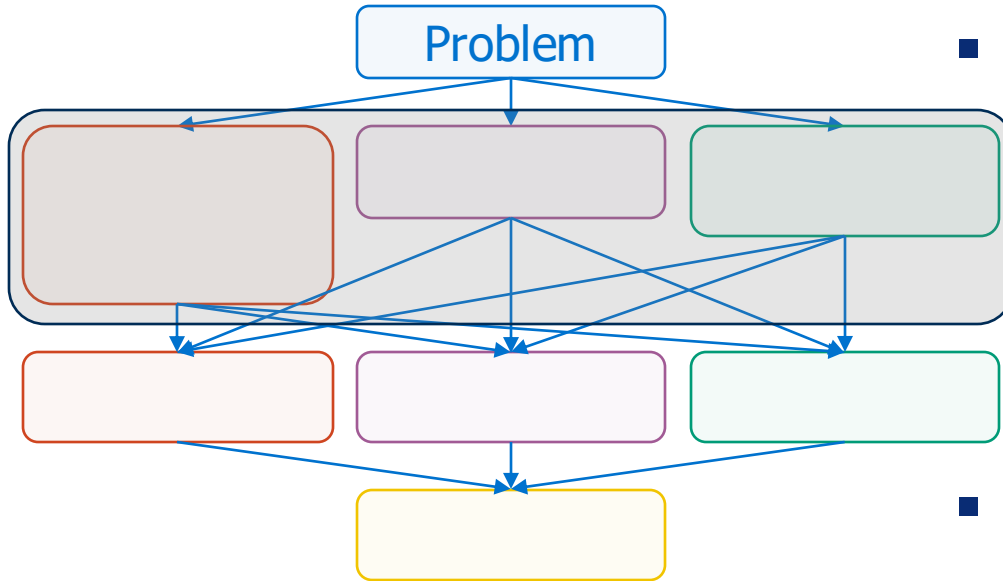


- **Multi-agent solving similar problems**

- Communication (dependency) between agents
- Multi-agent “debate”

- **Not fully parallelizable**

Challenge



- **Multi-agent solving similar problems**

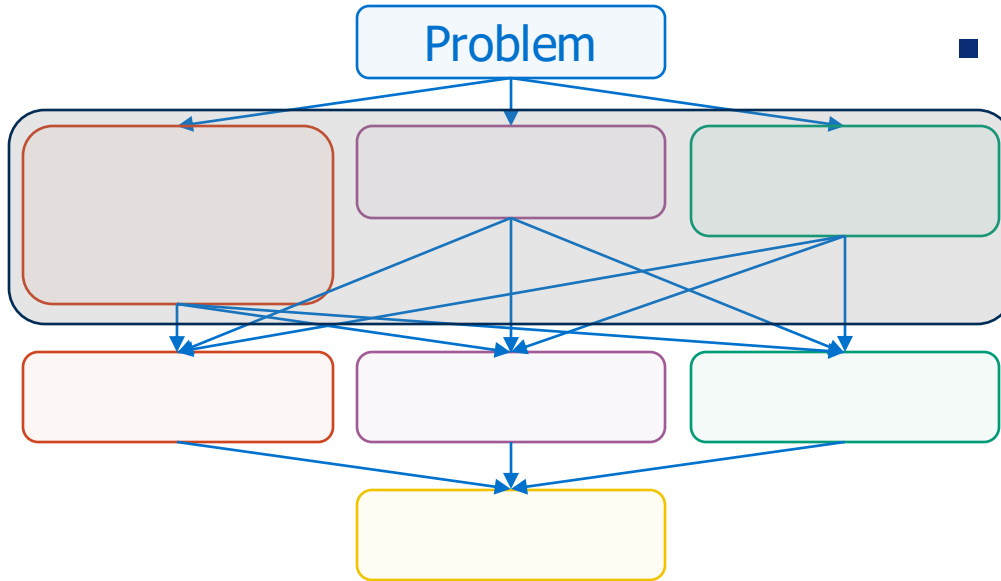
- Communication (dependency) between agents
- Multi-agent “debate”

- Not fully parallelizable
- Straggler effect

Hogwild! Inference: Parallel LLM Generation via Concurrent Attention

Rodionov et al.

Challenge



- We don't want threads waiting long for each other:
 - Better parallelism and efficiency
 - Effective reasoning in one agent can quickly benefit other agents too

KV Cache Structure

- When is KV cache shared?
- When a worker generates “\n\n”, copy and share the KV cache, resume the decoding.

KV Cache Structure

Problem: Compute $\sqrt{(31)(30)(29)(28) + 1}$.

Alice [1]: Hi, I'm Alice. Here's how we can solve the problem step by step...

Bob [1]: Hi, I'm Bob. Let me see... We need to compute the square root of $(31 \cdot 30 \cdot 29 \cdot 28)$ plus 1. That's $\sqrt{31 \cdot 30 \cdot 29 \cdot 28 + 1}$.

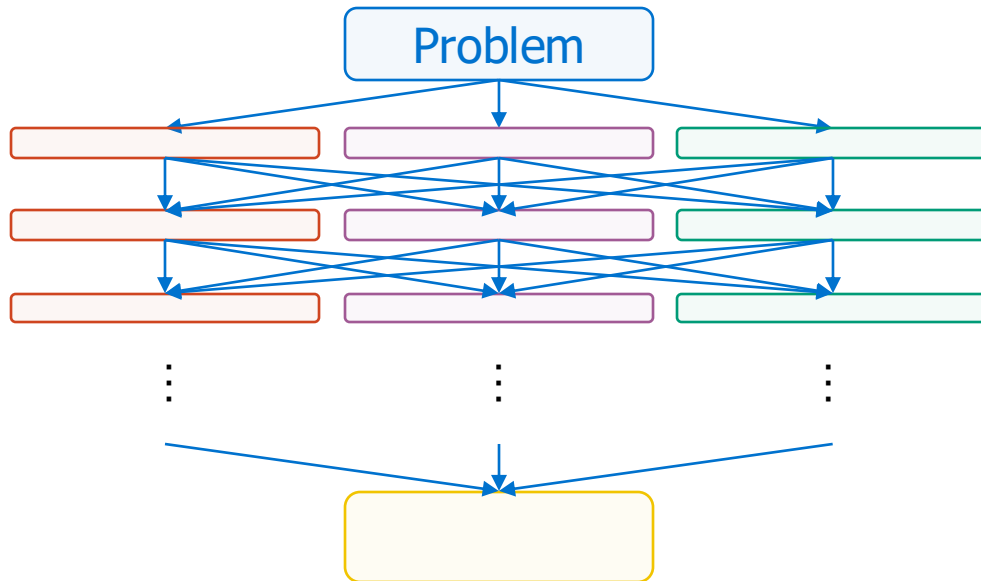
Alice [2]: The expression is $\sqrt{31 \cdot 30 \cdot 29 \cdot 28 + 1}$. Hmm, maybe there's a pattern here. I remember that sometimes products of consecutive numbers plus 1 have square roots that are integers. Like, for example, $n(n-1)(n-2)(n-3) + 1$ might form a square. Wait, but here it's four consecutive numbers? Or maybe three?

Bob [2]: Let me calculate the product first. $31 \cdot 30$ is 930, then $29 \cdot 28$ is 812. So $930 \cdot 812$. Let me multiply those. $930 \cdot 800 = 744,000$ and $930 \cdot 12 = 11,160$. Total is $744,000 + 11,160 = 755,160$. Then add 1: $755,161$. Now $\sqrt{755,161}$. Hmm, $\sqrt{755161}$. Let me see, 870 squared is 756,900 which is higher. So maybe 869 squared?

Alice [3]: Alternatively, maybe there's a formula for the product of four consecutive numbers plus one? Wait, actually, let me think of smaller numbers to test. Let's try $n = 4$: $4 \cdot 3 \cdot 2 \cdot 1 + 1 = 24 + 1 = 25$, $\sqrt{25} = 5$. Which is 5, which is $4 + 1$? Wait, $4 + 1 = 5$, yes. Wait, maybe for n , the sqrt is $n^2 + (n-1)^2$? Wait $4^2 + 3^2 = 16 + 9 = 25$, which is 5 squared. Hmm interesting.

Bob [3]: Wait, 869 squared is $(870 - 1)^2 = 870^2 - 2 \cdot 870 + 1 = 756900 - 1740 + 1 = 755161$. Oh! That's exactly the number we had earlier. So $\sqrt{755161}$ is 869. Therefore the answer is 869. **But let me check Alice's idea too.**

Alice [4]: Let me test another number. Let's take $n = 5$. Then $5 \cdot 4 \cdot 3 \cdot 2 + 1 = 120 + 1 = 121$. $\sqrt{121} = 11$. 11 is $5 + 6$? No, $5 + 6 = 11$. Wait $5^2 + 4^2 = 25 + 16 = 41$, which is not 121. Hmm. Alternatively, maybe $(n(n-3) + 1)^2$? Let's see with $n = 4$: $4 \cdot 1 + 1 = 5$, squared is 25. For $n = 5$: $5 \cdot 2 + 1 = 11$, squared is 121. So that works! So the formula is $(n(n-3) + 1)^2$. Wait, let me check with $n = 31$. Then $n = 31$, so $31 \cdot 28 + 1 = 868 + 1 = 869$. Then 869 squared is the value. So that works. So the answer is 869. **That matches Bob's calculation!**



Hogwild!: Shared KV Cache

Common Cache

Task: compute $x^2 + x^4$ for x in $\{1, 2, 3\}$.

Workers history:

Alice [1]: Hey! Let's decide how we should collaborate.

Bob [1]: Hi, Alice! Let me suggest that I do $x=1$ and $x=3$, and you will do $x=2$. What do you think?

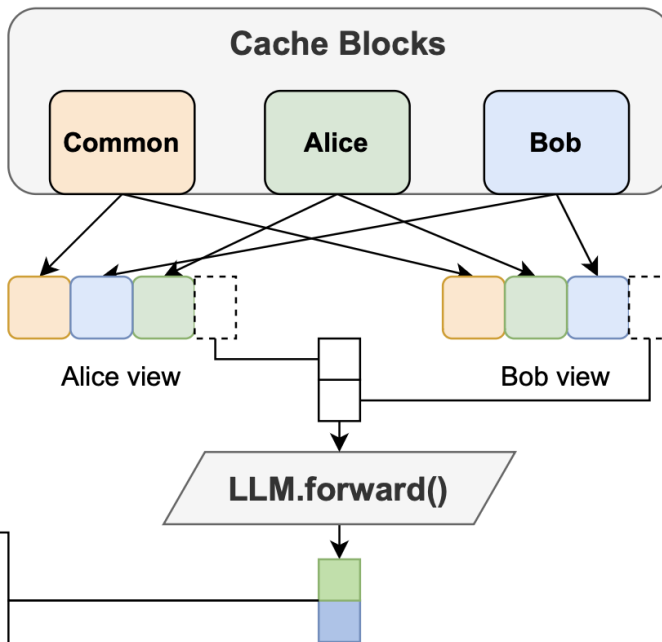
Alice [2]: Okay, let's go with that plan.

[Alice] Current Cache

Alice [3]: For $x=2$: ...

[Bob] Current Cache

Bob [2]: Perfect. Starting with $x=1$: ...

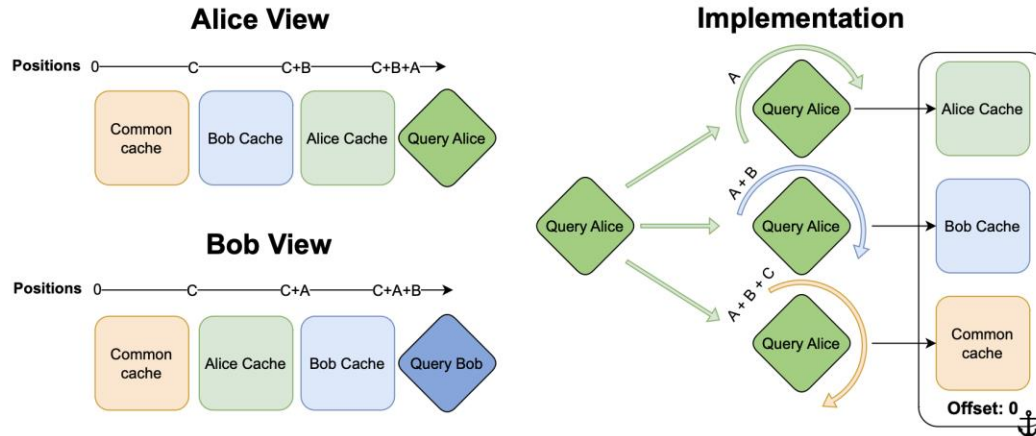


Token Ordering

- Shuffle ordering of cache blocks from other agents
 - We want randomness in ensemble models.
- Assume base LLM uses RoPE.
- How to deal with new token positions?
- Naïve way: KV cache before RoPE; overhead scales cubically.

Token Ordering

- Assumption: a lot of “key” tokens from other (n-1) agents.
- Better way: rotate query to accommodate new positions.



Trick: Prompting

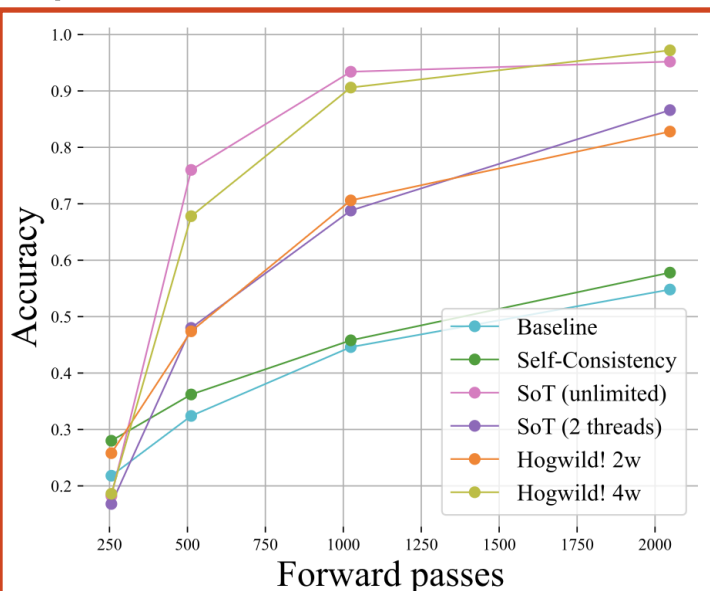
- System prompt: describe the rules of “shared cache”.
- Collaboration prompt: “Wait, am I doing redundant work? (yes/no):”.

Experiments: Baseline Methods

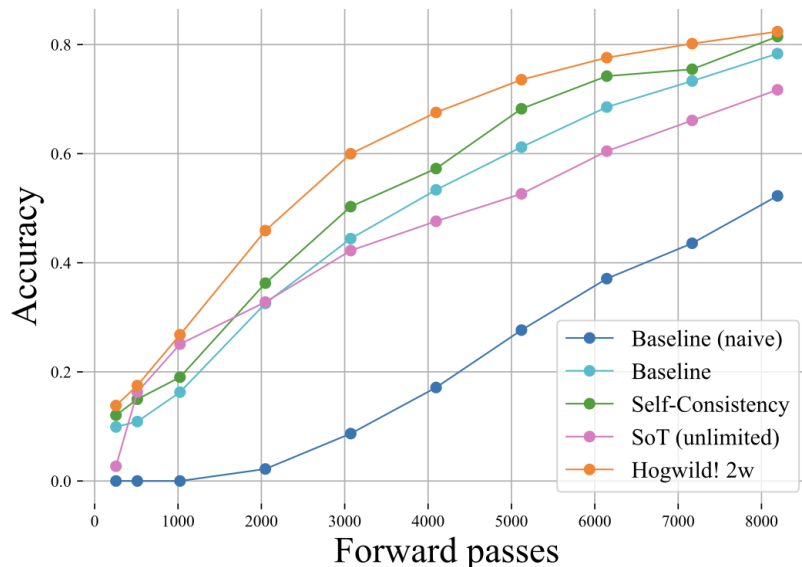
1. Skeleton-of-thought (SoT)
 - A planner LLM generates outline and sub-tasks
 - Parallel threads solve sub-tasks
 - Aggregate results for final prediction
2. Self-consistency:
 - LLM instances solve problem independently
 - View concatenated solutions before final prediction

Experiments: Results

SoT performs well when there are well-defined sub-tasks.



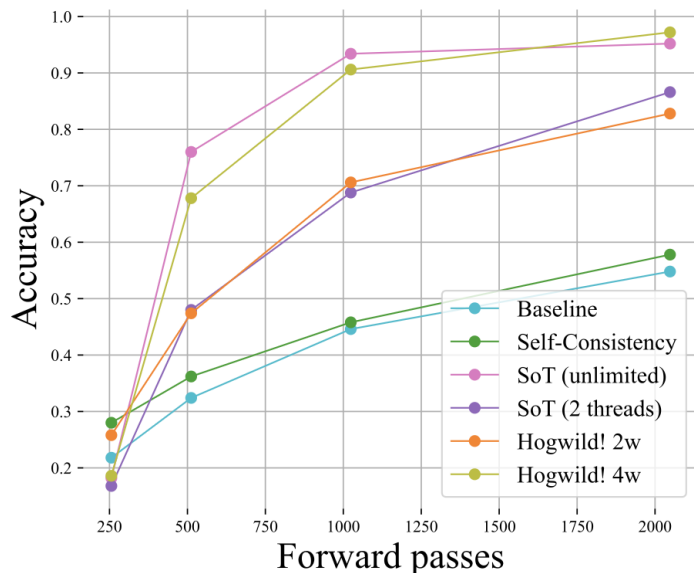
Synthetic tasks with
5 GSM8k questions



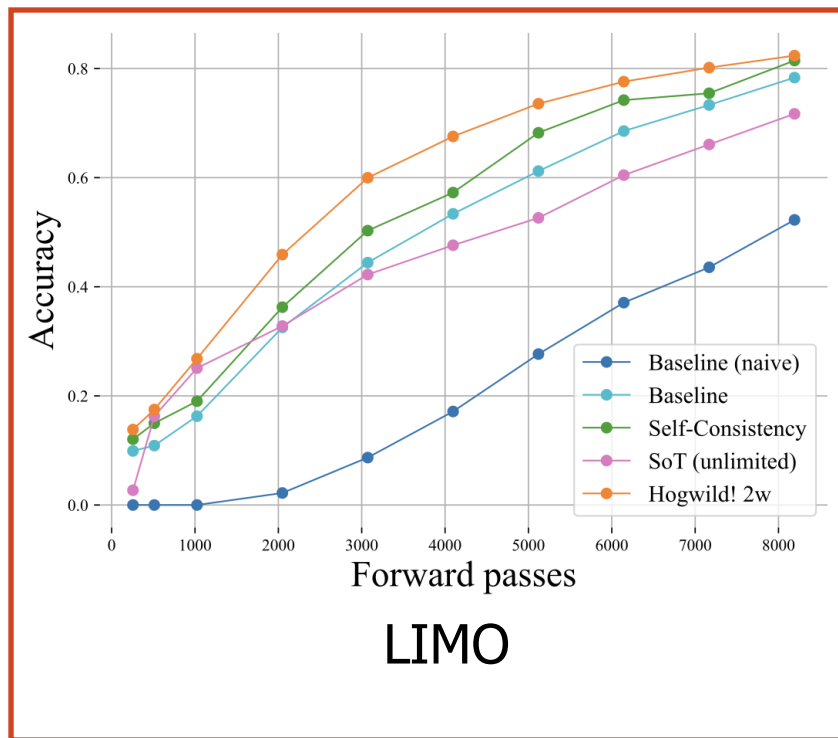
LIMO

Experiments: Results

Hogwild! and self-consistency outperforms when planning ahead is hard.



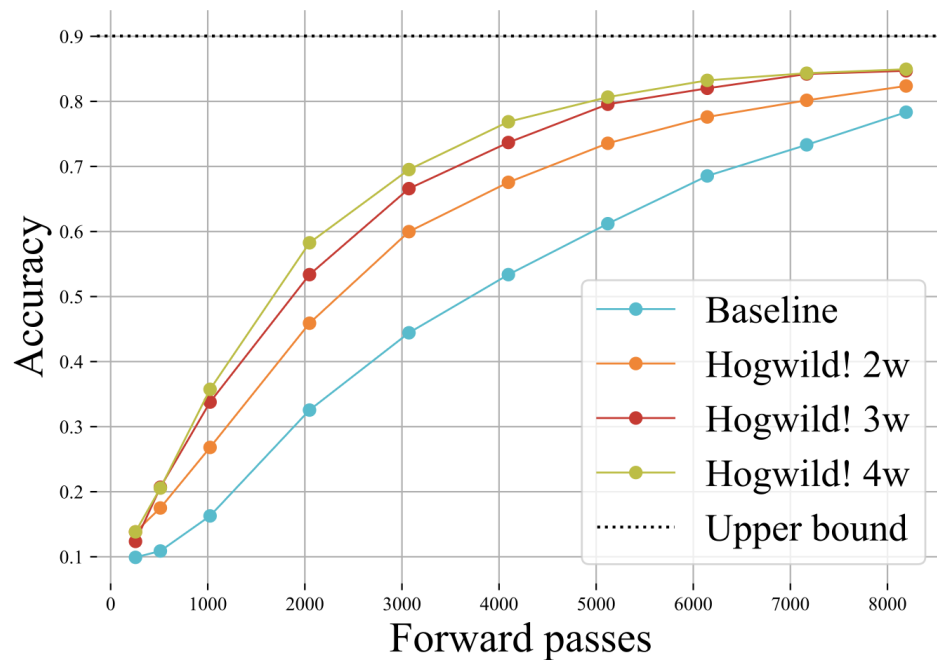
Synthetic tasks with
5 GSM8k questions



LIMO

How many workers?

- Diminishing returns



“Collaborativeness”

- LLM-as-a-judge: GPT-4o score collaboration
 - 1 as “no collaboration” and 6 as “optimal collaboration”

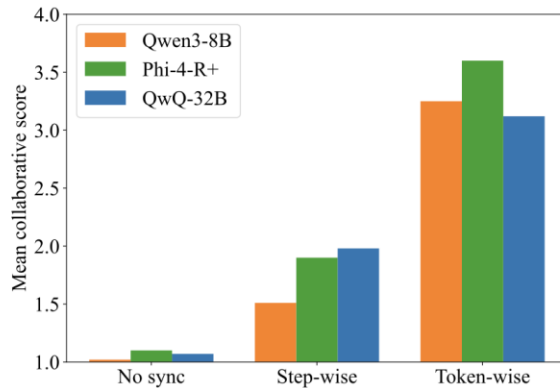


Figure 7: Mean collaborativeness score from GPT-4o. **No sync** is independent generation, **Step-wise** is restricted Hogwild! where worker can only see each-other’s past steps, **Token-wise** is full Hogwild! with instant cache exchange.

Efficiency

- Small latency overhead
- Near-linear scaling
- (However, a lot of duplicate reasoning & diminishing returns of performance.)

# Workers	1024	2048	4096	8192	16384
Tokens per second					
1	20.1	20.0	19.7	19.3	18.3
2	36.3	36.2	36.1	36.1	34.3
4	68.9	69.0	69.1	66.3	60.3
Latency per forward (ms)					
1	49.7	50.0	50.9	51.7	54.5
2	55.1	55.3	55.4	55.3	58.3
4	58.1	58.0	57.9	60.4	66.4
Time to generate # tokens (s)					
1	52.3	103.3	206.5	416.7	853.5
2	29.9	58.1	114.6	228.0	454.4
4	16.7	31.6	61.3	120.7	239.2

How Hogwild! improves performance?

- Model self-ensemble
- More timely interaction (compared to standard multi-agent systems)
- Divergent thinking

How Hogwild! Inference works?

Example: divergent thinking

Task 122

> Alice: *Quick check: am I doing redundant work? (yes/no):* no. Bob, I see you're working on the same product. Let me think of another approach. Let me write the product as $\prod_{k=1}^{n-1} \frac{3k+5}{3k+2}$. Let's consider writing out the first few terms to see if there's a pattern.

Task 9

> Bob: *Quick check: am I doing redundant work? (yes/no):* yes, I was considering similar things to Alice. Let me switch to a different angle. Let's consider the distances between the frogs.

Task 40

> Bob: *Quick check: am I doing redundant work? (yes/no):* Yes, Alice is computing a_3 , so I'll do something else. Let me try to find a pattern or transformation that could simplify the recurrence.

Task 466

> Bob: *Quick check: am I doing redundant work? (yes/no):* Yes, Alice is already calculating the x-coordinate of A. I'll do something else. Let me find the coordinates of L instead.

Takeaways

- Task (thinking) parallelism: some of the thinking process can happen in parallel.
 - Consider the deep research mode of ChatGPT – we are exploring a lot of possible directions, each taking very long thinking.
- What kind of problems are suitable for parallel thinking?
 - Well-defined sub-tasks
 - If we want self-ensemble – multi-agent debate.
- When do threads “sync”?
 - Waiting long – after finishing sub-tasks.
 - More frequently – quick pruning and converging thoughts.

Hogwild!



Learning Adaptive Parallel Reasoning with Language Models

Pan, Lu, et al.

Motivation

- **Serialized chain-of-thought methods**
 - Increased latency
 - Strain context window limits
- **Parallel methods**
 - Lack coordination between inference paths
 - Redundant computation, limiting improvement and scalability
- **Structured inference-time search methods**
 - Hand-designed search structures (graphs, reasoning trees, etc)
 - Limited flexibility and scalability

Adaptive Parallel Reasoning (APR)

- Models learn **when** and **how** to parallelize their inference operations.
 - No need for fixed search structures
- Generalizes existing approaches
 - Model learns to combine breadth (parallelism) with depth (of individual branches)

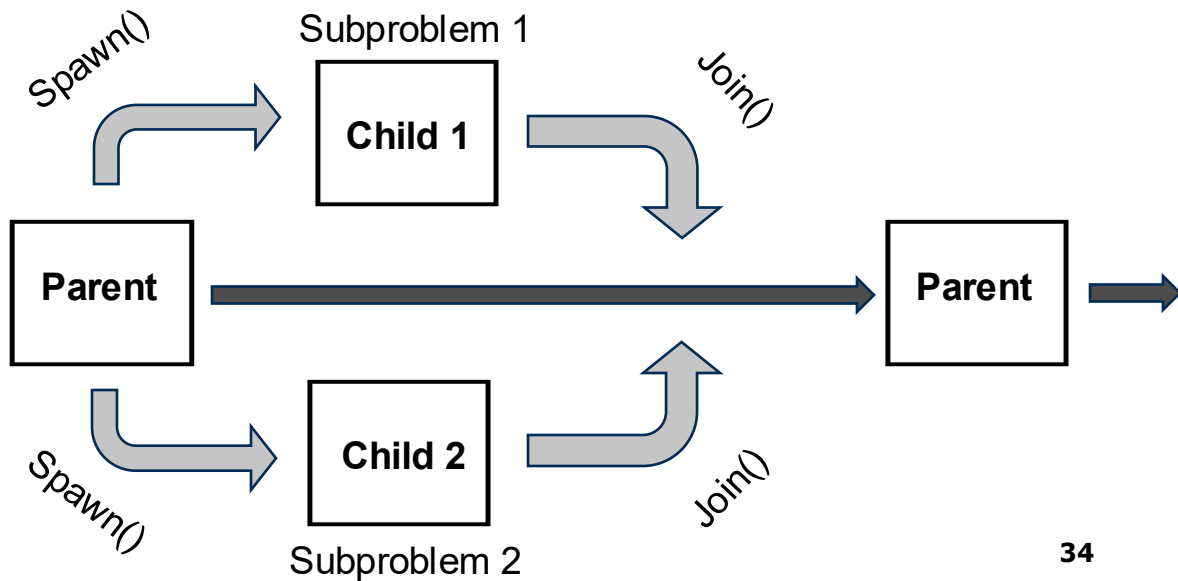
Key Innovation 1: Parent-Child Threading

- **Parent-child threading**

- Parent thread spawns child threads via **spawn()** to explore subproblems
- Each child runs independently in parallel and returns concise results via **join()**

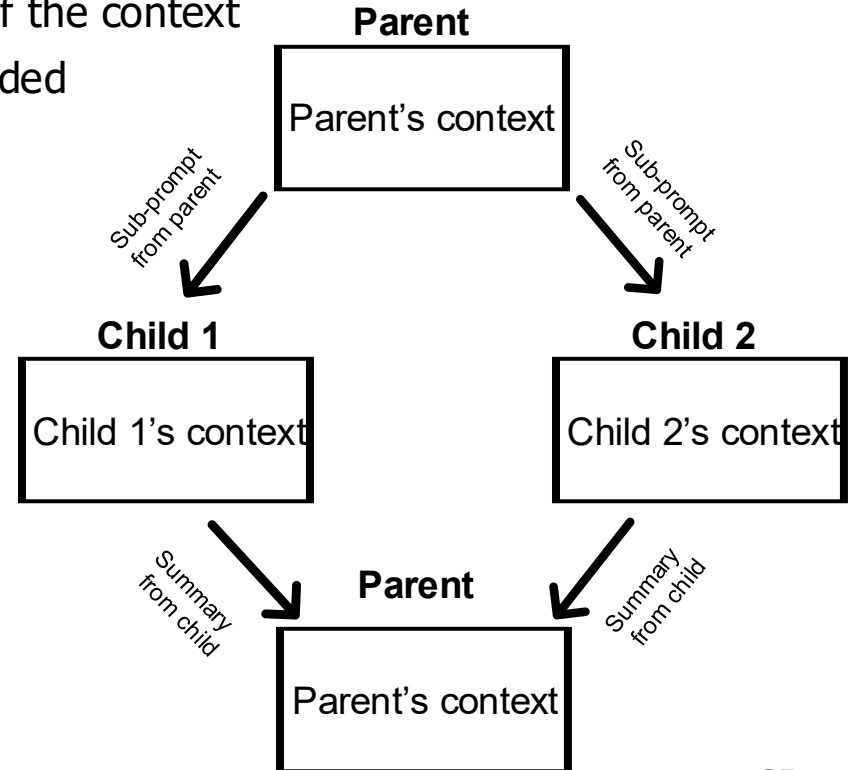
- RL trains model to decide:

- When to branch or join
- Prompts to feed to children
- How many children to spawn
- How to integrate results



Advantage: Context Partitioning

- Each thread (parent, children) has its own copy of the context
- Each child thread's reasoning does not get appended to the parent's context stream.
- **Children:**
 - Each can explore its subproblem using its entire context in parallel with others
- **Parent:**
 - Gets concise summaries/results from children



Key Innovation 2: End-to-End Reinforcement Learning

- Traditional:
 - Hand-designed reasoning tree
 - APR: Use RL to teach the model...
 - When to call **spawn()**
 - How many children to spawn
 - What sub-prompts to give them
 - When/how to call **join()** and integrate results
- 1. Rollout: LM performs reasoning (with spawn/join events)
 - 2. Evaluate final answer → compute reward (e.g., 1 if correct, 0 otherwise)
 - 3. Update LM parameters with policy gradient (GRPO algorithm)

Stream-of-Search (SoS/SoS+)

- Serialized CoT technique using either BFS or DFS to solve reasoning problems
- Authors use **SoS+**
 - Hybridizes BFS and DFS
 - Represents best serial CoT technique

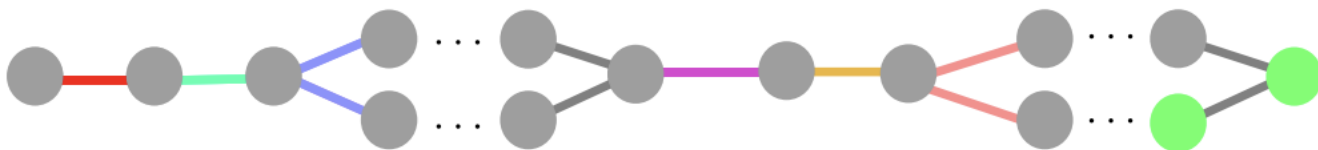
Method	Temp = 0.0	Temp = 0.1	Temp = 0.5	Temp = 1.0
SoS	49.5%	49.6%	47.1%	37.9%
SoS+	57.3%	57.1%	52.0%	48.1%

SoS+ vs APR, General

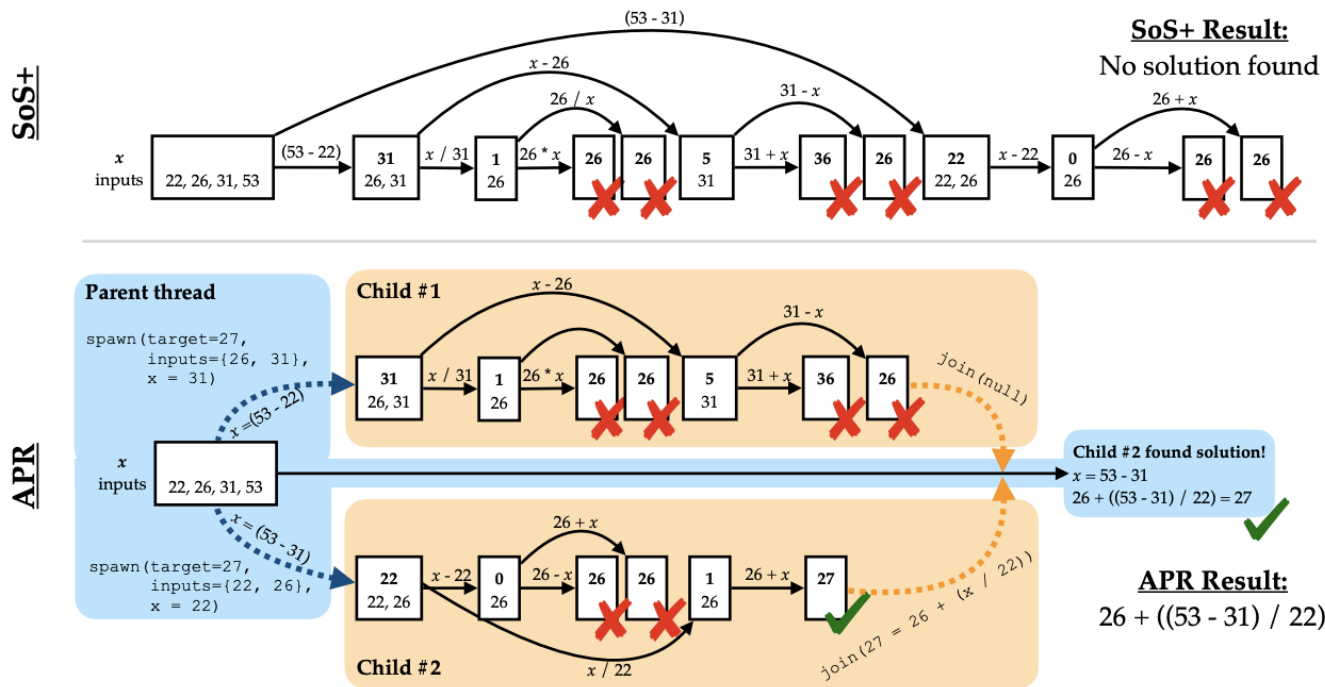
Previous chain-of-thought methods serialize the reasoning tree even if they have parallelizable components.



APR trains the model to spawn child threads that work on sub-tasks in parallel. Once completed, they are joined into the main thread.



SoS+ vs APR, Countdown Task



Training Setup

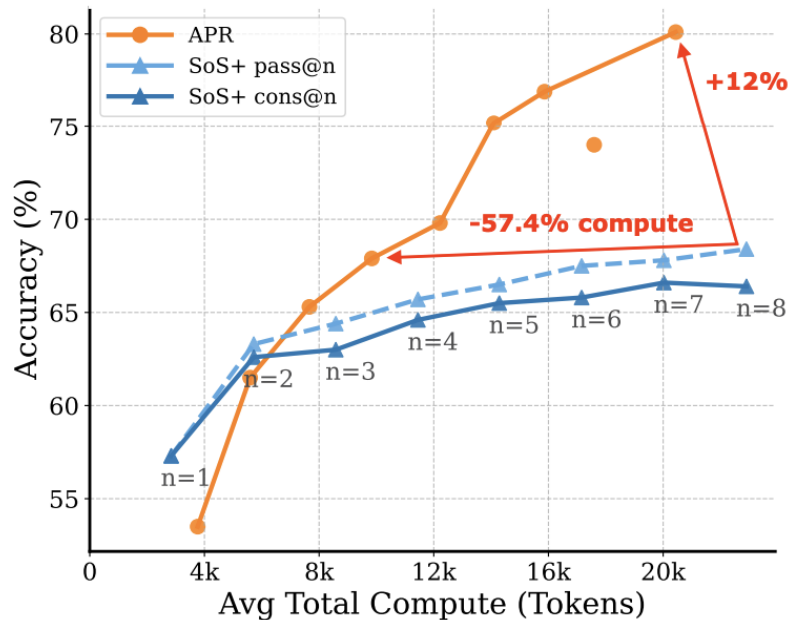
- Step 1: **SFT**
 - Teach models how to use spawn/join syntax
 - Sample solution traces generated by symbolic solvers
- Step 2: **Reinforcement Learning (GRPO)**
 - Reward = correctness of solution (Countdown task)
 - Model learns how to allocate compute (child threads)

Experiments: Countdown Task

- Goal: Find arithmetic expression using given numbers to reach target
- Baselines:
 - **SoS+**
 - Serialized CoT hybridizing BFS and DFS
 - Represents best serial CoT technique
 - **SoS + cons@n:**
 - N parallel independent SoS runs
 - Take majority vote
 - Used to scale up SoS+ compute
- Metrics: Accuracy, total token usage, latency
 - Latency measured using both **sequential tokens** and **wall-clock** time

Experiments: Results

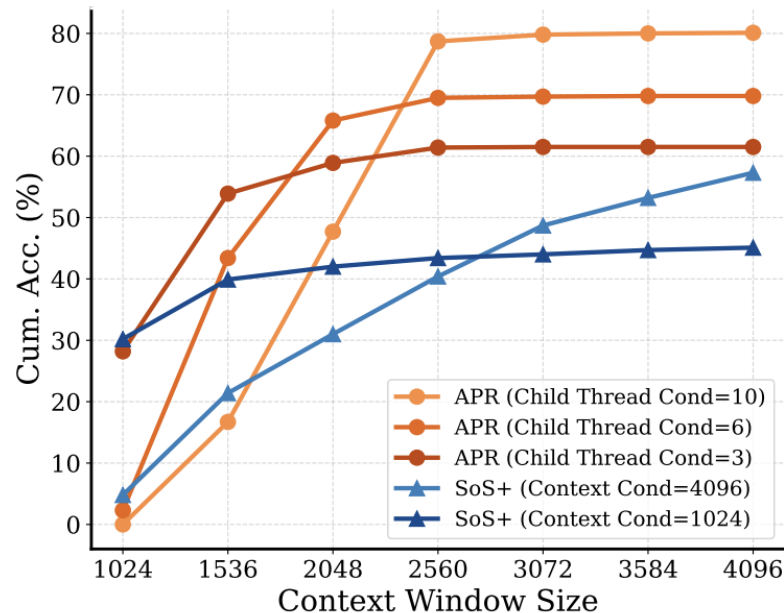
- Baseline: SoS with cons@n
- Total compute scaling:
 - SoS: Increase N in Best-of-N sampling
 - APR: Increase number of child threads
- “Parallelism overhead”: Worse in low-compute regimes (<4k tokens)
- Superior scaling by enabling parallel execution of child threads



(a) Performance vs. Total Compute

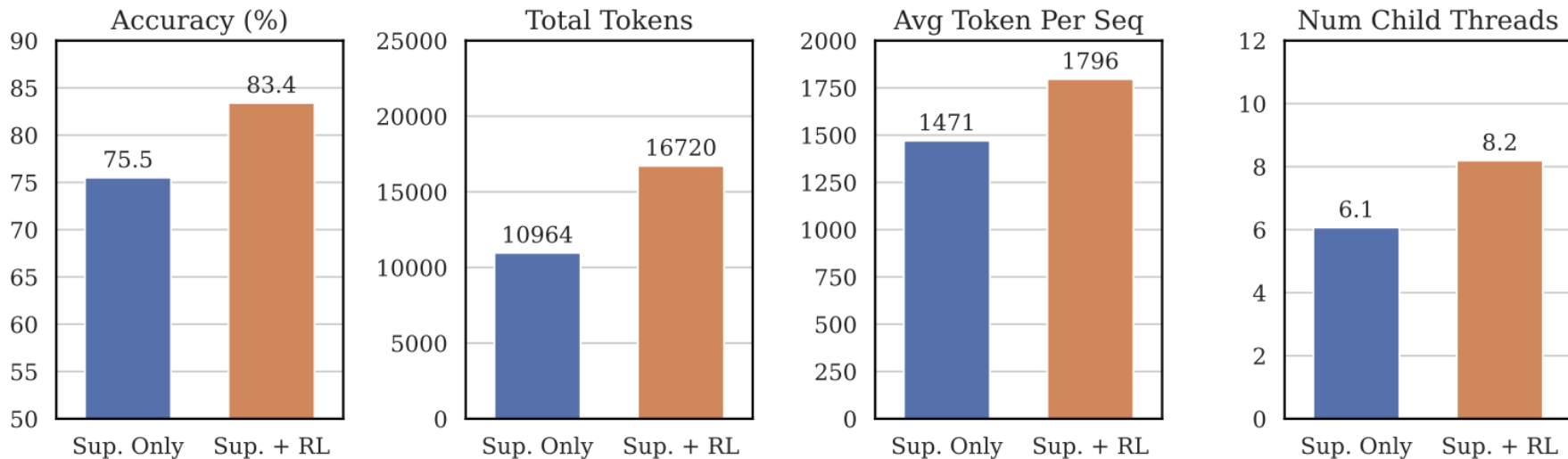
Experiments: Results

- Baseline: SoS+
- APR superior at exploiting context
- Parallel reasoning:
 - More total tokens
 - Not packed into one context window



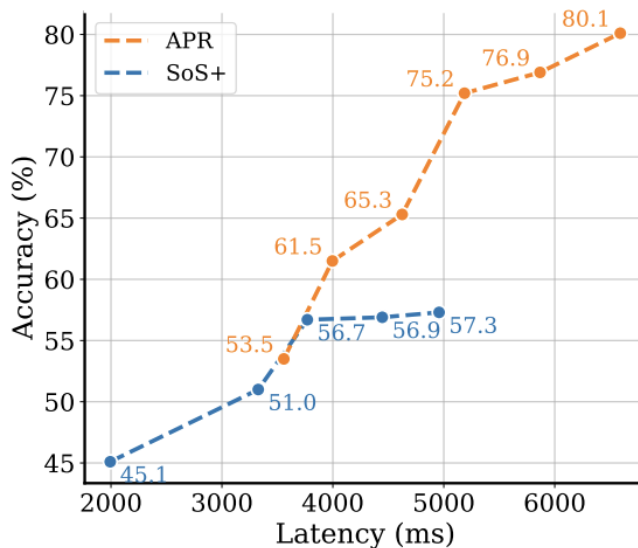
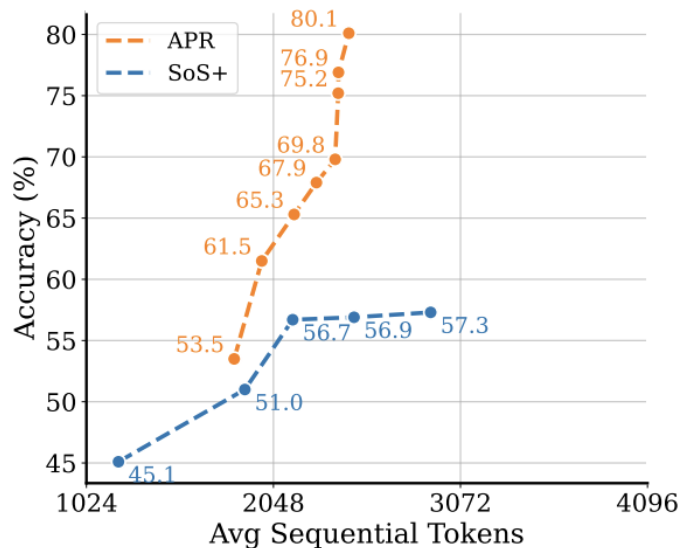
(b) Performance vs. Context Window Size

Experiments: APR Pre- vs Post- RL



Breadth (num threads) scaled up more than depth (tokens/seq)

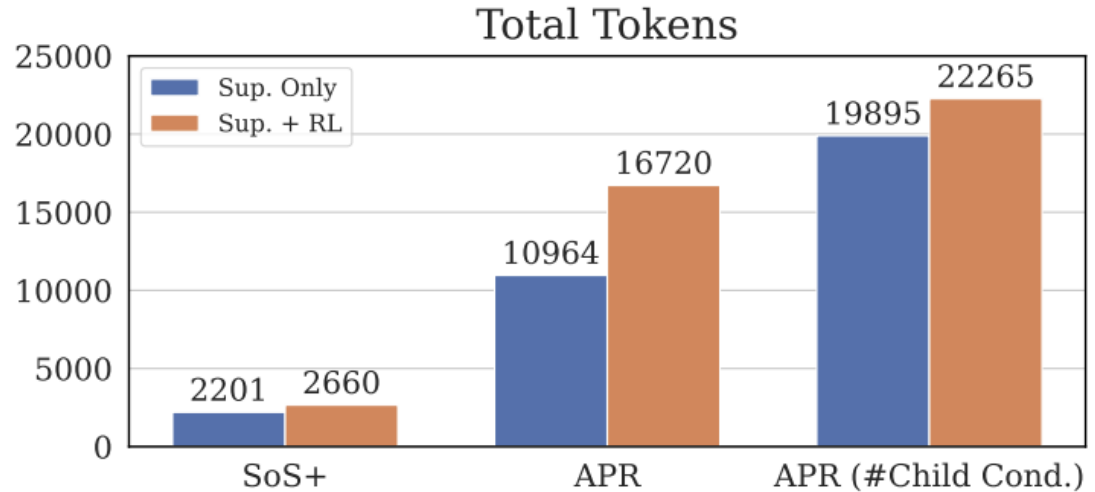
Experiments: Efficiency



- APR achieves higher accuracy with fewer tokens and lower latency
- Superior efficiency

Ablation: RL in APR and SoS+

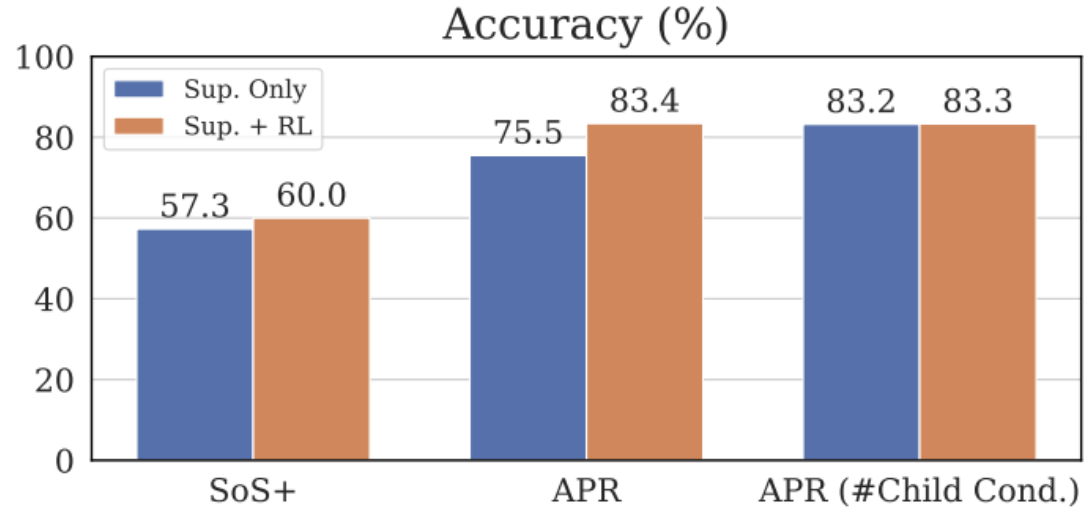
- RL improves performance and **increases token usage**
- APR saw a greater increase in token usage
 - Parallelization allows scaling beyond context limitations



APR (# Child Cond.): Model always uses max number of child threads (10)

Ablation: RL in APR and SoS+

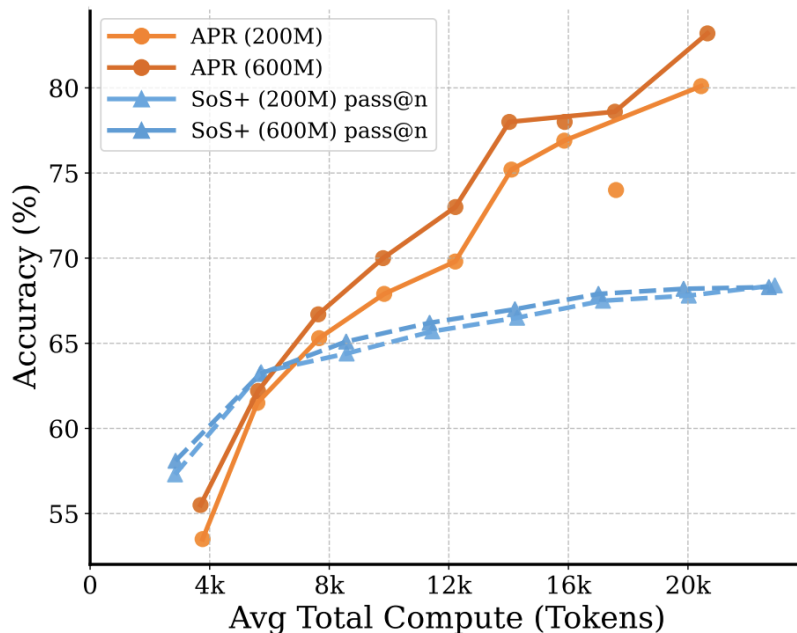
- APR saw a greater increase in accuracy than SoS
- Far less when fix number of child threads at 10:
- RL teaches model to **scale compute**, rather than improving decision quality



APR (# Child Cond.): Model always uses max number of child threads (10)

Results with Larger Models

- Previous results use a 293M-parameter model following Llama2
- Findings scale to 600M-parameter models
- APR scales better with model size than SoS+



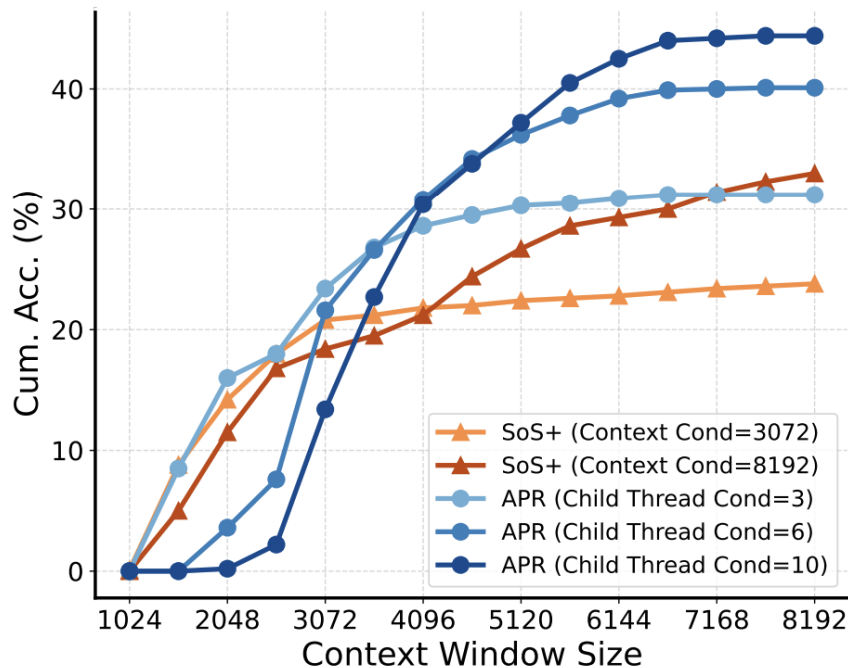
Results with pretrained models

- Models used in main study were trained from scratch
- Goal: Verify findings scale to large pretrained models

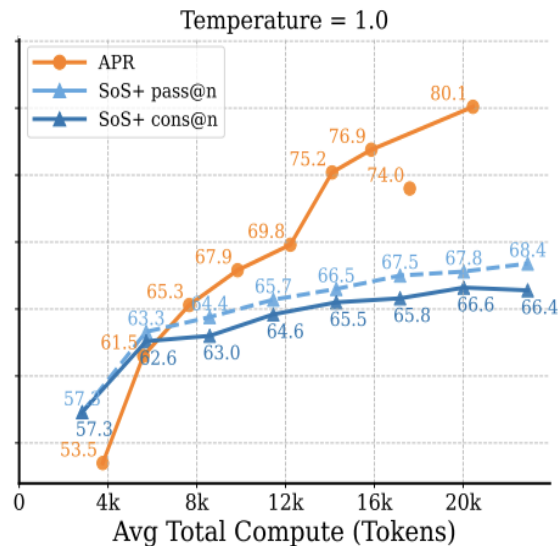
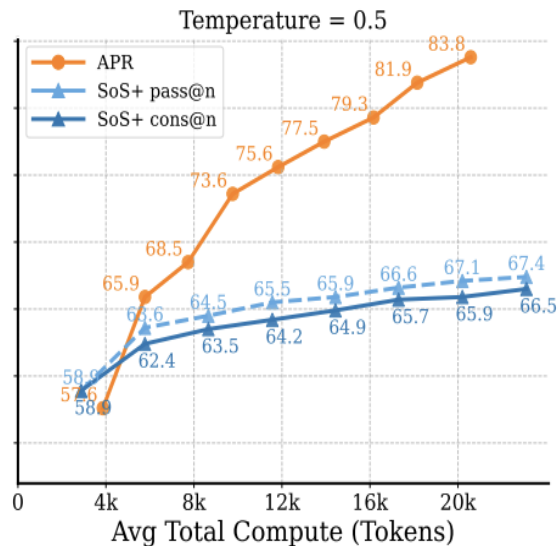
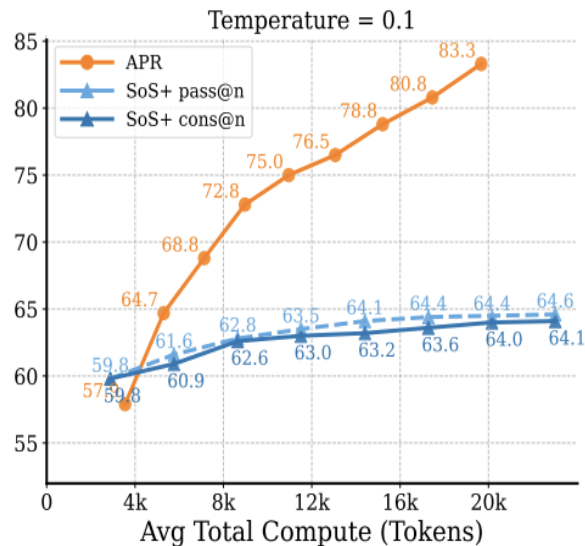
	Llama2 200M	Qwen2.5 1.5B (pretrained)
SoS+	57.4 %	57.5 %
APR	83.2 %	80.2 %

Results on Larger Countdown Task

- Countdown with 4 numbers previously; now 5
 - Search space 40x larger
- Context budget expanded to 8k tokens



Results Consistent Across Temperatures



Summary

- **Motivation:** Scaling LLM inference via **intelligent parallelism**
 - Serial CoT: Context window limitations; latency
 - Existing parallel methods: Poor coordination; redundancy
 - Hand-designed reasoning trees: Lack flexibility
- **Key contribution:** Intelligent, model-powered parallelism
 - Language models provided **parent-child threading** mechanism
 - End-to-end training via Reinforcement Learning

Discussions

1. Can Hogwild! Inference enable more effective collaboration beyond “prompted” divergent thinking?
 - Theoretically yes but unclear from current experiments.
2. Future of task (thinking) parallelism.
 - Trade-off between performance gains and overhead from multi-agent debate.
 - Nicely splitting big problems into sub-tasks is nice. What problems are suitable for sub-tasking and can LLMs achieve this with simple prompting?

You are negative side. You disagree with the affirmative side’s points. Provide your reasons and answer.

Discussion Questions

- Do we expect this improvement to generalize to other tasks?
- What kinds of tasks benefit from parallelization?
- Brute force in disguise?
 - Main benefits are from scaling threads for greater token usage



JOHNS HOPKINS

WHITING SCHOOL
of ENGINEERING

© The Johns Hopkins University 2023, All Rights Reserved.