Carnegie Mellon University

Message Passing Enables Efficient Reasoning

Xuecheng Liu* **Daman Arora*** **Gokul Swamy** **Andrea Zanette**
Carnegie Mellon University

Abstract

While inference-time scaling has improved the reasoning abilities of large language models (LLMs), the need to generate long chains-of-thought (CoTs) is a computational bottleneck. Thus, in contrast to sequential scaling methods like CoT, recent parallel scaling techniques instead use fork and join (FJ) primitives to divide work across multiple *LLM threads*. However, in the fork-join paradigm, threads are typically transient and do not communicate pointwise with one another which limits scalability. To tackle this, we introduce Message Passing Language Models (MPLMs), a framework for LLM reasoning in which threads communicate directly via lightweight send and receive primitives. MPLMs enable efficient scaling through two key mechanisms: (1) reduced communication costs, achieved by avoiding redundant context sharing, and (2) preemption, which allows threads to terminate early based on partial information from their peers.

We demonstrate the promise of MPLMs on 3 classes of tasks. First, on Sudoku puzzles, we show that MPLMs require an asymptotically smaller context than both serial CoT and parallel FJ. We then fine-tune a single model to solve 25×25 puzzles that remain challenging for standard CoT and FJ approaches, as well as frontier reasoning models without tools. Second, on 3-SAT puzzles, the capability of preemption allows termination of unpromising branches, which results in improved efficiency. Finally, we show that appropriately prompted large pre-trained models follow the MPLM protocol, achieving competitive results on long-context question answering relative to popular fork-join approaches.

1 Introduction

Large reasoning models (Guo et al., 2025; OpenAI et al., 2024) enhance the reasoning capabilities of foundation models by allocating extra compute at test time. This compute is devoted to a *chain of thought* (CoT)—an intermediate sequence of reasoning steps. The dominant reinforcement-learning-based training paradigm used to train reasoning models (Guo et al., 2025) yields *monolithic* CoTs, where the model explores alternative strategies, backtracks on inconsistencies, verifies intermediate results, and aggregates partial solutions into a final answer. However, this process is inherently *serial*, as tokens must be generated autoregressively, creating a computational bottleneck. Moreover, transformer-based architectures (Vaswani et al., 2017) impose practical limits: positional embeddings (Su et al., 2023) constrain usable context and the quadratic cost of self-attention reduces returns on additional compute. As we imagine an exciting future where reasoning models can tackle complex problems, such as engineering a new product, advancing scientific research, or even running a company, autoregressively generated CoTs may ultimately prove to be a limited approach.

Not all reasoning needs to unfold as a single sequential chain. For example, when designing a complex software system, work is often divided across teams: one team may focus on the database layer, another on the user interface, and another on the networking layer and they all work in parallel, coordinating when necessary. An initial attempt to this paradigm of scaling inference-time compute has been investigated in recent literature (Pan et al., 2025; Zheng et al., 2025) where an LLM can *fork* its own execution and decode multiple threads in parallel, before the resulting computation of the threads is aggregated by the parent thread.

While this fork-join paradigm reduces latency and alleviates context pressure, it retains critical limitations: *all coordination is centralized* and *context for workers is transient*. Parallel workers remain

*Equal contribution. Project webpage: Code:

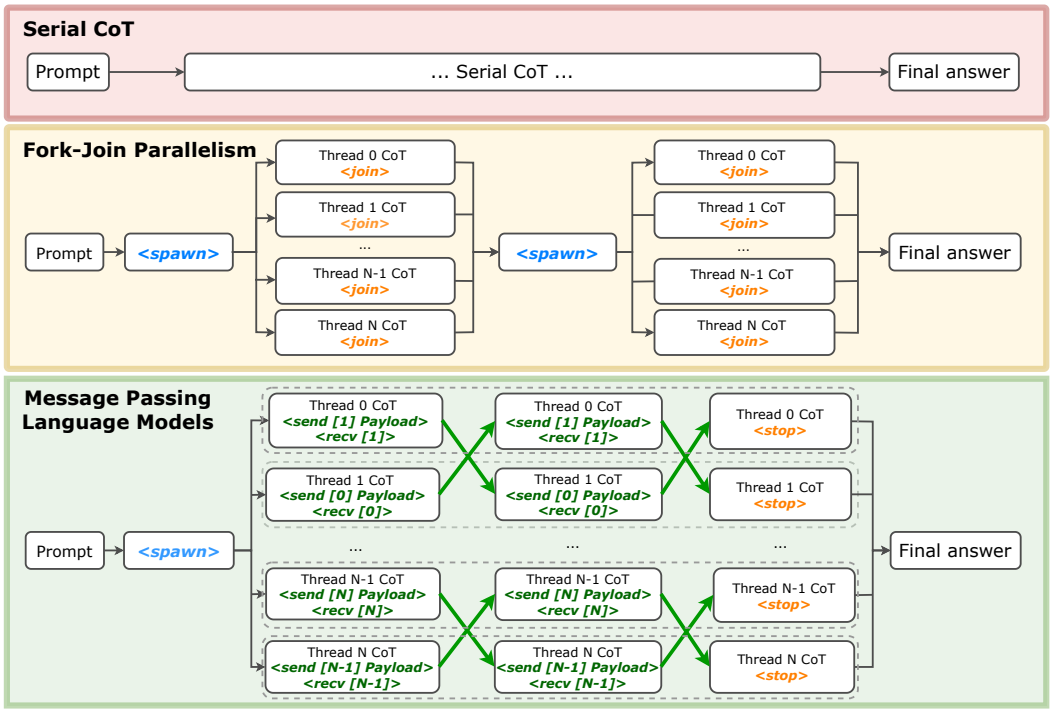

Figure 1: Comparison of reasoning paradigms for scaling test-time compute. **Top:** Serial Chain-of-Thought (CoT), where a single model generates a monolithic reasoning trace sequentially from prompt to answer. **Middle:** Fork-Join parallelism, where multiple independent transient CoT threads are spawned in parallel and periodically synchronized via global join operations before continuing. **Bottom:** Message Passing Language Models (MPLMs), where concurrently decoded threads communicate directly through explicit `send` and `recv` primitives and maintain a persistent context, enabling fine-grained coordination and information sharing throughout inference.

isolated during execution, their contexts are temporary, and all information must flow through the parent thread, which introduces both communication bottlenecks and implicit serialization. Conceptually, this is akin to an organization in which every decision-no matter how local or routine-must be routed through a single CEO: such a structure is inherently brittle, limits parallelism, and defeats the purpose of delegation.

In response, we propose **Message Passing Language Models** (MPLMs), a framework that decomposes reasoning into persistent, semi-independent threads that coordinate via explicit point-to-point message passing. By distributing reasoning across multiple interacting workers that communicate through explicit message passing, MPLMs enable more efficient use of test-time compute and improved scalability as problem complexity grows. Unlike prior multi-agent or debate-based approaches, which rely on independently acting agents with pre-defined roles and fixed communication protocols, MPLMs formulate reasoning as a single, unified process that is decomposed into communicating sub-threads. Rather than optimizing separate agents for different objectives, a single model dynamically decides when, with whom, and what to communicate using its own learned reasoning capabilities, allowing intricate communication patterns and task-specific structures to emerge organically.

We instantiate these ideas on two structured reasoning testbeds: Sudoku and 3-SAT puzzles. These problems are deterministic and naturally parallelizable which allows us to scale complexity incrementally in a way that cleanly isolates the benefits of our reasoning decomposition from confounding factors. Efficient Sudoku solutions exploits the benefits of sparse, local communication, while 3-SAT utilizes a complementary advantage of preemption in search. We show more favorable scaling with complexity of the problem, more specifically the size of the grid in Sudoku and the number of variables in 3-SAT as compared to Serial or Fork-Join baselines. Particularly impressively, we are able to scale to 25×25 Sudoku grids that remain **out of reach even for frontier reasoning models** such as GPT-5 Pro.

Finally, we demonstrate preliminary evidence of the ability of large pretrained models to use the MPLM framework when prompted, without the need for any training. On LongBench-v2 (Bai et al., 2025), we show promising results highlighting improved efficiency as compared to RLMs (Zhang et al., 2026), a contemporary fork-join approach.

In summary, our contributions are three-fold:

1. We introduce MPLMs, a framework for message passing between parallel LLM threads and analyzing the key *mechanisms* through which they unlock improved efficiency for language model inference.
2. We demonstrate *empirical scalability* on Sudoku puzzles well beyond currently existing serial CoT and Fork-join parallelism and improved efficiency of SAT puzzles.
3. We provide evidence on the ability of large pretrained models to *route information appropriately* using the MPLM framework on Long Context Question Answering.

2 Preliminaries

Language Model A language model (LM) computes a distribution over tokens y in a vocabulary $\mathcal{V}$, given a prefix $x = (x_1, x_2, \cdots, x_N) \in \mathcal{V}^*$. Formally, an LM defines the conditional distribution $p(y|x)$ over the vocabulary $\mathcal{V}$. In the standard setting, generation in LMs is *autoregressive*, that is, given a prefix x, the model generates tokens $y_1, y_2, \cdots, y_M$ based on a sampler over the conditional distribution over tokens. Modern LLMs are based on the Transformer (Vaswani et al., 2017) architecture, where each token attends to all previous tokens in the context window causing both training and inference costs to scale quadratically with the sequence length, making long-context generation expensive.

Batched inference Batched inference refers to serving multiple generation requests in a single forward pass (or in tightly packed micro-batches) to improve device utilization and throughput. Frameworks such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024a) are highly optimized for batch inference and exemplify this design pattern by combining runtime scheduling with KV-cache-aware execution allowing high-throughput inference.

MPI The Message Passing Interface (MPI) is a standardized framework for parallel and distributed computation based on explicit message passing between processes, exposing primitives such as send/receive, synchronization, and dynamic process management. Classic references such as Gropp et al. (1994); Snir et al. (1998) describe how persistent workers exchanging messages enables decentralized computation. Our work adapts these principles as an abstraction for distributed reasoning among LLM threads.

3 Message Passing Language Models

In traditional LLM inference, a model generates a *single* stream of tokens autoregressively in response to a user's request. In contrast, MPLMs create multiple *LLM threads* for handling sub-tasks, which are indexed by an ID and associated with distinct prompts and contexts. These threads are decoded concurrently through batched inference (Kwon et al., 2023; Zheng et al., 2024a).

3.1 Execution Control Directives

The foundation of MPLMs is a set of *directives*: explicit instructions that control execution. The model itself generates special tagged sub-strings that the framework interprets as directives. These play a role analogous to process creation, point-to-point communication, synchronization, and termination in message-passing systems. In MPLMs, we have 4 fundamental directives: `spawn`, `send`, `receive` and `stop`:

- **Spawn** The spawn directive requires the model to generate a string of the format:

  $$\texttt{<spawn[}id_1, id_2, \ldots, id_N\texttt{]> prompt </spawn>}$$

  This generates N new LLM threads (similar to forking) with the corresponding textual IDs and the 'prompt' concatenated in their individual contexts.

- **Send** The send directive is one of the two primitives responsible for handling message-passing. To generate a send, the model has to generate a string of the form:

  $$\texttt{<send[}id_1, id_2, \ldots, id_N\texttt{]> message </send>}$$

  This sends the `message` mentioned in the directive to LLM threads with IDs $id_1, id_2 ... id_N$.

- **Receive** The receive directive is the second primitive responsible for handling message-passing. To generate a receive, the model has to generate a string like:

  $$\texttt{<recv[}id_1, id_2, \ldots, id_N\texttt{]>}$$

  The behavior of this primitive is: wait for messages from threads $id_1, id_2 ... id_N$.

- **Stop** This directive is responsible for ending the execution of an LLM thread. A stop is triggered when a thread generates a string of the form `<stop>`. Whenever a stop is generated by a thread its execution is permanently stopped by the inference engine.

3.2 Inference Logic

Now, we describe how these directives are instantiated on a standard batched inference engine. Standard batched inference engines (e.g., vLLM) treat each decoding request independently. MPLMs instead require *coordinated* parallel decoding: threads may spawn new threads, send messages, and block on receives. We implement this coordination via a lightweight *controller* layered on top of the inference engine $\mathcal{E}$ while leaving the underlying token sampling and KV-cache management unchanged. This controller can be visualized as a mini-operating system working on the substrate which is the inference engine $\mathcal{E}$.

The controller repeatedly invokes $\mathcal{E}$ to decode all *runnable* threads *in parallel* until they reach either (a) a directive boundary, (b) EOS, or (c) a maximum step budget. A directive boundary occurs when the emitted token stream matches a directive format (`spawn`, `send`, `recv`, or `stop`). This is implemented by specifying special stop-tokens to $\mathcal{E}$. When a thread emits a `send` directive, the specified message is delivered to the target threads and decoding continues. On the other hand, upon emitting `recv`, the thread is blocked until it has received messages from all the relevant threads listed in the directive. For a more detailed pseudo-code reference, refer Section A of the Appendix.

This framework also gives freedom in terms of certain design choices such as the implementation of the `recv` directive. In particular, we explore a wait-for-all and a wait-for-any implementation. Additionally, it also naturally allows a thread to *re-spawn* itself with a compressed task summary which leads to higher efficiency. Finally, we discuss how the ability of *preempting* another thread is natively supported by the MPLM framework. We defer a detailed discussion on these aspects to Appendix B for clarity of exposition.

4 MPLMs Enable Efficient Reasoning

Next, we formalize the intuitive notion that reduced aggregation costs leads to efficiency in MPLMs. In particular, we design a simplified setup below to theoretically show that MPLMs can be significantly more efficient than other paradigms such as Serial CoT and Fork-Join (FJ).

Setup. We consider a task that proceeds in T iterations. In each iteration, N workers (ranks) run in parallel, each solving a local sub-task and producing at most W tokens. To advance to the next iteration, each worker requires information from at most k workers from the previous iteration. This information is exchanged via messages of size at most M tokens. Thus, rather than requiring access to all previous computations, each iteration depends only on a small, localized subset of the messages constructed in the previous iteration. We compare three paradigms to operationalize this particular task keeping in mind their natural constraints:

1. **Serial CoT:** This is the standard way in modern reasoning models reason, that is, using a Serialized Chain-of-Thought (Guo et al., 2025; OpenAI et al., 2024). Since there is no parallelization, all the work of individual ranks is done sequentially for each iteration.
2. **Fork-Join (FJ):** In this paradigm (Pan et al., 2025; Zheng et al., 2025), a model can spawn threads for various sub-tasks and then executes them in parallel. At the end of an iteration, a master rank collects all messages, constructs new per-rank prompts, and spawns fresh workers for the next iteration.
3. **Message passing (MPLM):** Each rank maintains a persistent context throughout iterations and communicates point-to-point with its k neighbors.

Assumption 1 (Bounded local processing)**.** *The per-iteration local reasoning grows linearly with the total message volume received by the thread, i.e., $W = \mathcal{O}(kM)$. This sparsity condition is naturally satisfied in many structured reasoning tasks. For example, in code generation, modules typically depend on a small set of interfaces rather than the full codebase.*

Now, we analyze the maximum context required for each of these methods since context length is the fundamental bottleneck in scaling inference compute. Denote the maximum context length required by the Serial, FJ and MPLM paradigms as $C_{\max}^{\text{Serial}}, C_{\max}^{\text{FJ}}$, and $C_{\max}^{\text{MPLM}}$.

Theorem 2 (***Maximum context for Serial v. FJ v. MPLM***)**.** [Proof: Appendix K]

$$C_{\max}^{\text{Serial}} = \mathcal{O}(TNkM), \quad C_{\max}^{\text{FJ}} = \mathcal{O}(TNM) \;\; \textit{and} \;\; C_{\max}^{\text{MPLM}} = \mathcal{O}(TkM).$$

We observe that Serial CoT incurs the highest context requirements due to serialization. The Fork-join execution improves this by parallelizing the reasoning per-rank, but still incurs a centralized aggregation cost that scales with N. In contrast, MPLMs exploit sparse communication yielding a maximum context requirement that is smaller by a factor of $\Theta(N/k)$ compared to FJ. This factor is especially significant when communication is sparse. This provides a theoretical justification for the **context efficiency** of MPLMs.

5 Experiments

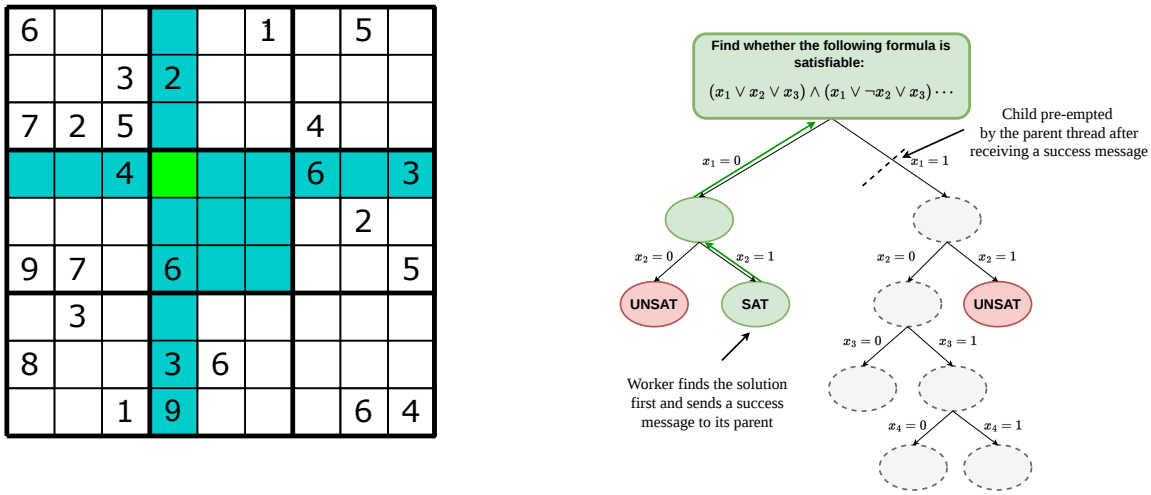

Figure 2: Left: This figure describes the communication pattern for a 9×9 Sudoku grid. Consider the worker for the cell (4, 4) highlighted in green. This cell only sends and receives messages from its neighbors highlighted in blue, reducing context requirement. **Right:** This figure describes how preemption can improve efficiency in 3-SAT problems where a search branch that finds a solution can send a message its parent without having to wait for other threads to finish their computation.

We ground MPLMs in 2 structured reasoning tasks: Sudoku and 3-SAT Puzzles. These experiments allow us to empirically validate the efficiency gains on sparse problems we proved above. For both experiments, we teach a model, using supervised fine-tuning, to reason in a way that allows it to exploit the primitives afforded by MPLMs, more specifically the *send* and *recv* directives. We describe more details below:

Training Procedure: We simulate the message-passing reasoning process in Python and use the resulting trajectories to generate Chain-of-Thought training data. For Sudoku, we simulate execution of the naked-singles strategy iteratively; for SAT, we follow DPLL algorithm (Davis et al., 1962). Example CoTs for each method can be found in Appendix Sections E to J. Using these datasets, we perform supervised fine-tuning (SFT) on Qwen3-0.6B-Base (Yang et al., 2025a). Each training run cost less than 48 H100 hours. More details can be found in Appendix C.

Metrics: To measure the performance of our method, we consider metrics including (i) **Accuracy:** the percentage of puzzles solved successfully; (ii) **Latency:** the average time taken to solve a puzzle in seconds; (iii) **Maximum context:** the maximum context required to train one puzzle; and (iv) **Sequential tokens:** the maximum number of causally dependent tokens that must be processed sequentially by the model, as used by Pan et al. (2025).

Baselines: For baselines we also generate data for (i) Serialized **(Serial)** Chain-of-Thought which generates every token autoregressively with no parallelism and (ii) Fork-Join **(FJ)** parallelism as implemented by Pan et al. (2025) where a parent thread iteratively spawns child threads and the parent thread combines the final responses of the children. Additionally, we evaluate closed-source reasoning models on Sudoku puzzles, including DeepSeek-R1 and GPT-5 Pro, using their respective APIs to assess how performance scales with problem complexity. We disable tool usage to isolate the model's reasoning behavior without external computation.

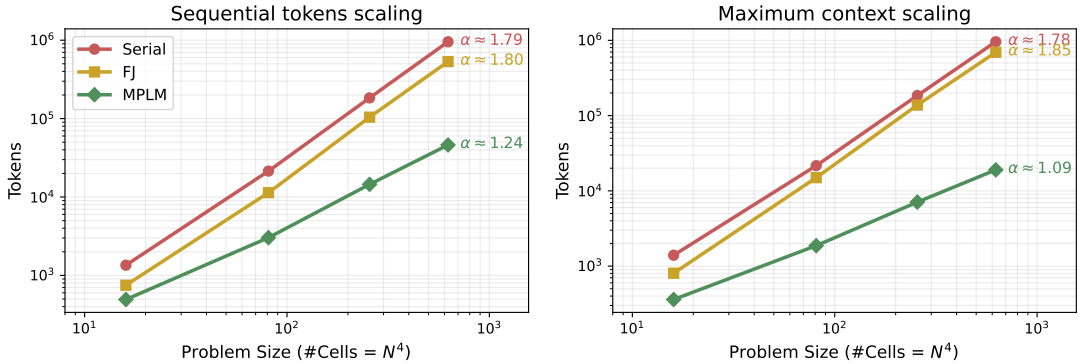

Figure 3: Scaling of sequential tokens (left) and maximum context tokens (right) required to solve a Sudoku puzzle as a function of the problem size (N^4). Curves are shown on a log-log axes with fitted power-law exponents α. Message Passing (MPLM) exhibits significantly **lower scaling exponents** than the Serial and Fork-Join (FJ) reasoning paradigms for both sequential tokens and maximum context required, indicating both better efficiency and the ability to solve larger problems.

Method	$n = 2$ (4×4)	$n = 3$ (9×9)	$n = 4$ (16×16)	$n = 5$ (25×25)
MPLM	**2.105s (100%)**	**14.939s (100%)**	**117.684s (92%)**	**1017.263s (72%)**
FJ	2.940s (100%)	59.562s (93.00%)	✗	✗
Serial	4.545s (99%)	✗	✗	✗
DeepSeek R1	144.271s (90%)	765.982s (0%)	597.939s (0%)	431.751s (0%)
GPT-5 Pro	(100%)	(100%)	(45%)	(20%)

Table 1: Avg. latency and accuracy across Sudoku sizes. We report wall-clock inference time and solution accuracy. A red cross (✗) denotes infeasible training due to context or compute constraints. MPLM remains feasible and accurate at larger grid sizes, while FJ, Serial baselines, and frontier models fail to scale. Due to the high cost of these models, we limit our evaluation to 20 queries per setting, and use the batched API which doesn't provide latency metrics.

5.1 Sudoku

Setup Generalized Sudoku is an $N^2 \times N^2$ constraint satisfaction problem requiring unique values across rows, columns, and sub-grids; we restrict to puzzles solvable by iteratively filling cells with only one remaining feasible value (naked-singles). We generate 1000 puzzles of sizes 4×4, 9×9, 16×16 and 25×25 for training and 200 puzzles for evaluation. See Appendix C for more details.

5.1.1 A Message-Passing Approach to Sudoku

In our MPLM design for Sudoku, we have a single agent for each cell in the Sudoku puzzle, which leads to N^4 workers for a $N^2 \times N^2$ puzzle. First, the parent thread (ID '0') spawns N^4 child threads (IDs $1, 2, \cdots, N^4$) using the `<spawn>` directive and sends them the initial state of the puzzle. Then, each worker computes the possible set of values for its corresponding cell. For instance, in a 9×9 puzzle, worker '12' would compute the possible values possible for cell $(1, 3)$. After computing this set, there are 2 possibilities:

1. **Case 1:** There are multiple possibilities for the current cell. In this case, it is necessary for the worker to gather more information before proceeding. Therefore, it generates the `<recv>` token, signaling that it is waiting for messages from its its 'neighbors'. Neighbors here refer to the workers in control of cells in the same row, column, or sub-grid.
2. **Case 2:** There is only 1 possibility for the current cell. In this case, the worker sends a message using the `<send [list of IDs]>` directive to the parent thread (ID '0') and all its 'neighbors' containing the value filled in the cell. For an $N^2 \times N^2$ grid, each cell has $3N^2 - 2N - 1$ neighbors (see Figure 2 for an example). After sending the message, the model generates `<stop>` to end its execution.

After all threads have stopped, the parent thread collates all the messages and generates the final solved Sudoku, which is parsed to evaluate the success/failure of the model.

5.1.2 Sudoku Results

MPLMs fundamentally improves scaling. We note that MPLMs operate in a qualitatively different scaling regime compared to both **Serial** and **Fork-Join (FJ)** reasoning as seen in Figure 3. Serial and FJ approaches exhibit worse power-law behavior, with both sequential token counts and maximum context requirements scaling with higher exponents of $\alpha \approx 1.8$. In contrast, MPLM achieves *substantially*

improved scaling exponents ($\alpha \approx 1.2$ for sequential tokens and $\alpha \approx 1.1$ for maximum context) owing to its *decentralized context*. This reflects a structural change in computation: MPLMs distribute reasoning across persistent workers that maintain local state and communicate point-to-point. By avoiding repeated global aggregation, MPLMs improve the asymptotic growth of both sequential computation and context usage. We also note that different paradigms might admit different CoT implementations, and alternative designs may improve constant factors; however, such changes *do not affect the asymptotic scaling trends* that form the basis of our comparison. A more detailed theoretical analysis is provided in Appendix K.2.

Message passing enable capabilities beyond frontier reasoning models. The improved scaling behavior of MPLM reasoning results in qualitative gains in problem-solving capability. In standard $N = 3$ Sudoku puzzles, we find that DeepSeek-R1 fails to reliably solve instances whereas *GPT-5 Pro* struggles in 25×25 Sudoku puzzles solving just 4 of 20 puzzles. In contrast, MPLM successfully solves 72% of 25×25 instances after being trained on just 1000 puzzles. We note that this comparison is not intended as a direct head-to-head evaluation however, the results illustrate what the MPLM framework can unlock when a model is trained to exploit structured parallelism, a capability that is difficult to achieve within serial CoT reasoning regardless of model scale. Together, these results show that MPLMs enable a new level of structured problem-solving. Additionally, we show that respawning can also improve efficiency for Sudoku in Appendix L. We also conduct additional measurements of quantities such as KV-Cache hit rate and GPU utilization in Appendix M.

5.2 3-SAT

Setup. We also study Boolean satisfiability (3-SAT), where *preemption* enabled by *asynchronous message passing* is critical. Solving SAT problems requires a search over possible assignments to variables. Our key insight to improve efficiency is that, once a satisfying assignment is found in one branch of the search tree, continuing computation in other branches is unnecessary. Such behavior is not possible with the Fork-Join paradigm, where the master only aggregates worker's responses after they have finished. In contrast, preemption can be easily implemented in the MPLM scaffold, where a worker can immediately send a message to its parent (which is actively waiting to receive messages) and the parent can decide to preempt the execution of other workers. To empirically verify this preemption capability, we generate random 3-SAT instances with variable counts ranging from 8 to 20 and train the model to solve the problems using SFT on the CoT. Training details are provided in Appendix D.

5.2.1 A Message-Passing Approach to 3-SAT

For 3-SAT, our MPLM design follows a distributed DPLL procedure (Davis et al., 1962). Each worker first performs unit propagation on its local subproblem. If the formula is resolved immediately, the model generates either `SAT` or `UNSAT`. Otherwise, it branches on a variable x_i and spawns 2 children corresponding to the assignments $x_i = 0$ and $x_i = 1$, and waits for messages from its descendants. MPLM supports preemption through asynchronous message passing. When a thread stops, the controller terminates its entire sub-tree to improve efficiency and avoid unnecessary computation. In particular, a parent does not need to wait for all children to complete before making progress. Instead, it reacts to **the child that returns first**. If a child reports `SAT`, the parent immediately terminates the search. This behavior propagates upward recursively to the root node as shown in Figure 2.

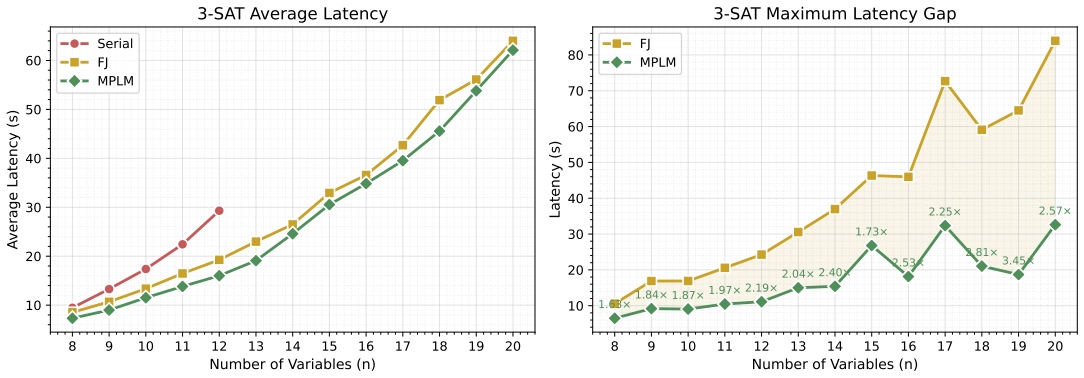

Figure 4: Left: Average inference latency across 100 test problems per variable count for MPLM, FJ, and Serial. **Right:** Latency and speedup on the maximum-speedup problem at each variable count, observed in the problems whose search tree is highly unbalanced. Note that Serial can't be measured after 12 variables because it exhausts the context window.

Partition	Qwen3-30B-A3B				Qwen3.6-35B-A3B			
	MPLM Accuracy (↑)	MPLM Latency (↓)	RLM Accuracy (↑)	RLM Latency (↓)	MPLM Accuracy (↑)	MPLM Latency (↓)	RLM Accuracy (↑)	RLM Latency (↓)
I. Single-Document QA	**38.1%**	**58.2s**	36.0%	117.3s	**48.0%**	**107.9s**	47.4%	233.0s
II. Multi-Document QA	**35.0%**	**58.8s**	21.6%	104.8s	**47.2%**	**104.2s**	40.0%	235.4s
III. Long In-context Learning	**34.9%**	**68.3s**	26.5%	98.5s	**49.4%**	**118.1s**	38.3%	327.6s
IV. Long-dialogue History Understanding	**51.3%**	**60.1s**	32.0%	92.6s	**61.6%**	94.1s	59.0%	**72.7s**
V. Code Repository Understanding	**35.0%**	**65.5s**	32.5%	93.3s	28.0%	**56.7s**	**52.0%**	187.8s
VI. Long Structured Data Understanding	**42.4%**	**64.2s**	28.1%	99.8s	39.4%	**103.5s**	**66.7%**	104.2s
Average	**37.8%**	**61.3s**	29.7%	105.7s	46.5%	**102.2s**	**46.7%**	223.5s

Table 2: LongBench-v2 results aggregated by major task category using task-count weighted averages. We compare MPLM against the RLM fork-join baseline using Qwen3-30B-A3B and Qwen3.6-35B-A3B without fine-tuning.

5.2.2 3-SAT Results

MPLMs reduce unnecessary computation through preemption. Figure 4 shows the latency results on 3-SAT for increasing complexity of the problem. The left plot shows the *average latency* over the full test set for each number of variables, and the right plot shows the *maximum speedup case* at each variable count, i.e., the test instance for which MPLM achieves the largest latency advantage over Fork-Join. Accuracy results are in Appendix D. We observe that MPLMs are faster than other baselines for all sizes of the problem. Additionally, as shown in the right plot of Figure 4, the greatest gains arise when the SAT search tree is highly unbalanced, where the two child branches of a parent differ dramatically in the amount of computation required before reaching a decisive node. In such cases, Fork-Join suffers from its centralized synchronization structure, and its overall latency is determined by the time required to reach the last leaf node. In contrast, MPLM allows for early termination, so the overall latency is determined by the time to the *first* satisfying node rather than the *last* explored node.

5.3 Long Context QA

Setup. While Sudoku and SAT isolate the algorithmic benefits of message passing in controlled settings, we also evaluate MPLMs on a more naturalistic long-context reasoning benchmark. We use LongBench-v2 (Bai et al., 2025), which contains 503 multiple-choice questions with contexts ranging from 8K to 2M words. The benchmark covers a diverse set of long-context tasks, including single-document QA, multi-document QA, long in-context learning, long-dialogue history understanding, code repository understanding, and long structured-data understanding. This setting is a natural testbed for MPLMs because solving long-context QA can be tackled by decomposing the context to multiple threads and cross-referencing evidence distributed across different chunks of the context. We describe our approach in detail below.

5.3.1 A Message-Passing Approach to Long Context Tasks

The parent thread first spawns worker for different parts of the large context based on a maximum threshold. For instance, if the threshold is 25K tokens and the entire context is 200K tokens, then 8 parallel threads are spawned. Then, each worker retrieves its assigned chunk into its local context and sends a concise summary to the parent. The parent then reasons over the aggregated summaries and can **selectively** and **iteratively** communicate with workers. If the parent needs more precise evidence from a specific region of the context, it sends a targeted query to the relevant workers; if the summaries are insufficient, it can query a broader subset or all workers. Once enough evidence has been collected, the parent terminates and produces the final answer. An example can be seen in Appendix O.

Baseline. We compare against Recursive Language Models (RLMs) (Zhang et al., 2025), a reasoning framework which uses fork-join style parallelism to spawn multiple threads which are equipped with specialized tools to gather context and write code. RLM workers are stateless, after returning their response to the parent, they do not maintain persistent local state and cannot be queried again through point-to-point communication. This makes RLM a strong and relevant baseline for evaluating whether persistent workers and selective message passing provide benefits beyond generic parallel decomposition.

5.3.2 LongBench-v2 Results

Prompted models can already use MPLM directives. Unlike our Sudoku and SAT experiments, this evaluation does not rely on any task-specific training. Both Qwen3-30B-A3B and Qwen3.6-35B-A3B are prompted to use the MPLM scaffold directly. The results therefore indicate that sufficiently capable models can already follow message-passing directives and make use of persistent point-to-point

communication at inference time. For future work, these prompted MPLM traces can be used as initial data for training stronger MPLM policies through supervised fine-tuning or reinforcement learning.

Persistent workers and point-to-point communication enable efficient iterative evidence aggregation. As shown in Table 2, MPLM improves average accuracy from 29.7% to 37.8% on Qwen3-30B-A3B while reducing average latency from 105.7s to 61.3s, which corresponds to approximately 1.7× lower latency. On the stronger Qwen3.6-35B-A3B model, MPLM achieves comparable average accuracy to RLM (46.5% vs. 46.7%) while reducing average latency from 223.5s to 102.2s, which corresponds to approximately 2.2× lower latency. We observe that in MPLM, the parent can repeatedly query the same worker for finer-grained details without reloading that part of the document. At the same time, point-to-point communication allows the parent to contact only the workers that are relevant to the current uncertainty. This reduces unnecessary context consumption and makes iterative evidence refinement more efficient. In contrast, fork-join workers are transient. Once a worker returns its response, its local context is discarded, making subsequent follow-up queries require re-spawning workers and reconstructing context.

6 Related Literature

Scaling test-time compute. Various techniques such as Self-Consistency (Wang et al., 2023), Best-of-N (Cobbe et al., 2021), Monte Carlo Tree Search (Xie et al., 2024), Tree-of-Thoughts (Yao et al., 2023) have been proposed to improve performance by scaling test-time compute. Another direction of scaling is sequential scaling which includes techniques such as Chain-of-Thoughts (Wei et al., 2022), Self-Refine (Madaan et al., 2023), Self-Correction (Kumar et al., 2024). However, with the advent of large reasoning models, long Chain-of-Thoughts learnt using reinforcement learning (Shao et al., 2024; Zelikman et al., 2022) have been more effective in scaling test-time compute (Guo et al., 2025) compared to other contemporary methods.

Efficient Reasoning. As test-time compute for reasoning is scaling, it is important to scale it efficiently. Various methods have been proposed to increase the efficiency of reasoning including length based regularization (Team et al., 2025; Arora and Zanette, 2025; Aggarwal and Welleck, 2025; Luo et al., 2025) and iterative summarization (Aghajohari et al., 2025; Yan et al., 2025; Yang et al., 2025b). However, these works primarily target the monolithic CoT setting whereas we focus on the self-parallelized reasoning setting.

Parallelizing LLM Threads. Our work is most closely related to parallelizing reasoning. Although techniques such as Self-Consistency improve performance, all generations are independent and lack coordination. Several papers have proposed using parallel reasoning structures such as in the works by Yao et al. (2023); Du et al. (2024); Ning et al. (2024); Teng et al. (2025). Adaptive Parallel Reasoning by Pan et al. (2025) proposes the **fork-join** parallelism model and optimizes LLMs via RL for the game of Countdown. Multiverse introduced by Yang et al. (2025c) also parallelizes inference whilst also introducing modifications in the attention mechanism. Another body of research (Chen et al., 2025; Zheng et al., 2025; Yang et al., 2025c; Macfarlane et al., 2025; Lian et al., 2025) focuses on extracting parallelizable reasoning structures from CoTs for math problems. Relatedly, Jin et al. (2025); Ning et al. (2024); Liu et al. (2024) focus on decoding different parts of a Chain-of-Thought in parallel. However, none of these works focus on an explicit framework to pass messages between independent LLM threads with a persistent context.

Multi-Agent Frameworks. Multi-agent frameworks such as CAMEL (Li et al., 2023), MetaGPT (Hong et al., 2024), AutoGen (Wu et al., 2024), GPTSwarm (Zhuge et al., 2024) and AgentVerse (Chen et al., 2024) study how multiple specialized LLM instances can collaborate through natural-language communication. These systems show that coordination and role specialization can improve performance on complex tasks. However, in most existing frameworks, agent roles, communication patterns, and agent creation mechanisms are largely pre-defined by the developer. In contrast, our work focuses on decomposing a single reasoning task using *runtime*-level communication primitives inspired by MPI, where spawn/send/recv/stop operations are first-class decoding directives which allow task specific agent creation and communication. Crucially, the use of these primitives is learned rather than hard-coded, allowing the model to flexibly allocate computation and more effectively leverage its internal reasoning capabilities.

7 Discussion, Conclusions and Limitations

In this work, we introduce MPLMs, which enable LLMs to dynamically create new threads and communicate through point-to-point messages. This capability allows LLMs to break down complex tasks and reason about individual sub-problems in a decentralized fashion, while communicating only when necessary. This

resembles how an actual human organization, such as a company, might function. To study the benefits of MPLMs, we instantiate them across both structured reasoning tasks and realistic long-context tasks. On Sudoku and SAT, we show that MPLMs require less context and enable more efficient inference compared to other reasoning paradigms. On LongBench-v2, we further find that sufficiently capable prompted models can already use MPLM directives to perform iterative evidence aggregation more efficiently, reducing latency while maintaining or improving accuracy.

We also acknowledge several limitations of our work. A key limitation is that, MPLMs require each worker to determine which other workers it should communicate with and when such communication is necessary. In structured domains such as Sudoku and SAT, this communication pattern can be specified relatively clearly, but in open-ended tasks it may require careful prompting or additional training. We believe that an important direction for future work is to generate parallel message-passing CoT data for training stronger MPLM policies beyond predefined scaffolds. This could enable MPLMs to generalize to less structured domains like agentic reasoning, code generation, and theorem proving. We also believe that additional communication primitives, such as broadcasting and all-gather, may further improve the expressiveness and efficiency of MPLM systems.

8 Acknowledgements

We would like to thank Yiding Jiang, Yuda Song, Fahim Tajwar, Abitha Thankaraj, Bhavya Agrawalla, Pranjal Aggarwal, Gaurav Ghosal, Jatin Prakash and Lili Chen for their feedback on the project and suggestions on improving the manuscript.

References

Pranjal Aggarwal and Sean Welleck. L1: Controlling how long a reasoning model thinks with reinforcement learning. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=4jdIxXBNve.

Milad Aghajohari, Kamran Chitsaz, Amirhossein Kazemnejad, Sarath Chandar, Alessandro Sordoni, Aaron Courville, and Siva Reddy. The markovian thinker: Architecture-agnostic linear scaling of reasoning, 2025. URL https://arxiv.org/abs/2510.06557.

Daman Arora and Andrea Zanette. Training language models to reason efficiently. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=AiZxn84Wdo.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025. URL https://aclanthology.org/2025.acl-long.183/.

Keyu Chen, Zhifeng Shen, Daohai Yu, Haoqian Wu, Wei Wen, Jianfeng He, Ruizhi Qiao, and Xing Sun. Aspd: Unlocking adaptive serial-parallel decoding by exploring intrinsic parallelism in llms, 2025. URL https://arxiv.org/abs/2508.08895.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=EHg5GDnyq1.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors,

Proceedings of the 41st International Conference on Machine Learning, volume 235 of *Proceedings of Machine Learning Research*, pages 11733–11763. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/du24e.html.

William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0262571048.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Tian Jin, Ellie Y Cheng, Zachary Ankner, Nikunj Saunshi, Blake M Elias, Amir Yazdanbakhsh, Jonathan Ragan-Kelley, Suvinay Subramanian, and Michael Carbin. Learning to keep a promise: Scaling language model decoding parallelism with learned asynchronous decoding. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=ZfX43ZZRZR.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. Training language models to self-correct via reinforcement learning, 2024. URL https://arxiv.org/abs/2409.12917.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative agents for "mind" exploration of large language model society. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=3IyL2XWDkG.

Long Lian, Sida Wang, Felix Juefei-Xu, Tsu-Jui Fu, Xiuyu Li, Adam Yala, Trevor Darrell, Alane Suhr, Yuandong Tian, and Xi Victoria Lin. Threadweaver: Adaptive threading for efficient parallel reasoning in language models, 2025. URL https://arxiv.org/abs/2512.07843.

Mingdao Liu, Aohan Zeng, Bowen Wang, Peng Zhang, Jie Tang, and Yuxiao Dong. Apar: Llms can do auto-parallel auto-regressive decoding, 2024. URL https://arxiv.org/abs/2401.06761.

Haotian Luo, Li Shen, Haiying He, Yibo Wang, Shiwei Liu, Wei Li, Naiqiang Tan, Xiaochun Cao, and Dacheng Tao. O1-pruner: Length-harmonizing fine-tuning for o1-like reasoning pruning, 2025. URL https://arxiv.org/abs/2501.12570.

Matthew Macfarlane, Minseon Kim, Nebojsa Jojic, Weijia Xu, Lucas Caccia, Xingdi Yuan, Wanru Zhao, Zhengyan Shi, and Alessandro Sordoni. Instilling parallel reasoning into language models. In *2nd AI for Math Workshop @ ICML 2025*, 2025. URL https://openreview.net/forum?id=a3o4b3hkwp.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=S37hOerQLB.

Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Prompting LLMs for efficient parallel generation. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=mqVgBbNCm9.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, et al. Openai o1 system card, 2024. URL https://arxiv.org/abs/2412.16720.

Jiayi Pan, Xiuyu Li, Long Lian, Charlie Victor Snell, Yifei Zhou, Adam Yala, Trevor Darrell, Kurt Keutzer, and Alane Suhr. Learning adaptive parallel reasoning with language models. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=YgwQ7sXPXU.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998. ISBN 0262692155.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, et al. Kimi k1.5: Scaling reinforcement learning with llms, 2025. URL https://arxiv.org/abs/2501.12599.

Fengwei Teng, Quan Shi, Zhaoyang Yu, Jiayi Zhang, Yuyu Luo, Chenglin Wu, and Zhijiang Guo. Atom of thoughts for markov LLM test-time scaling. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=qXSFkP0ELS.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Neural Information Processing Systems*, 2017. URL https://api.semanticscholar.org/CorpusID:13756489.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_VjQlMeSB_J.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

Yuxi Xie, Anirudh Goyal, Wenyue Zheng, Min-Yen Kan, Timothy P Lillicrap, Kenji Kawaguchi, and Michael Shieh. Monte carlo tree search boosts reasoning via iterative preference learning. In *The First Workshop on System-2 Reasoning at Scale, NeurIPS'24*, 2024. URL https://openreview.net/forum?id=s004OmYP2P.

Yuchen Yan, Yongliang Shen, Yang Liu, Jin Jiang, Mengdi Zhang, Jian Shao, and Yueting Zhuang. Inftythink: Breaking the length limits of long-context reasoning in large language models, 2025. URL https://arxiv.org/abs/2503.06692.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, et al. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

Chenxiao Yang, Nathan Srebro, David McAllester, and Zhiyuan Li. PENCIL: Long thoughts with short memory. In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=6wglsDXIei.

Xinyu Yang, Yuwei An, Hongyi Liu, Tianqi Chen, and Beidi Chen. Multiverse: Your language models secretly decide how to parallelize and merge generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025c. URL https://openreview.net/forum?id=r9YDEErKXU.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=5Xc1ecxO1h.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. STar: Bootstrapping reasoning with reasoning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_3ELRdg2sgI.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2025. URL https://arxiv.org/abs/2512.24601.

Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2026. URL https://arxiv.org/abs/2512.24601.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024a. URL https://openreview.net/forum?id=VqkAKQibpq.

Tong Zheng, Hongming Zhang, Wenhao Yu, Xiaoyang Wang, Runpeng Dai, Rui Liu, Huiwen Bao, Chengsong Huang, Heng Huang, and Dong Yu. Parallel-r1: Towards parallel thinking via reinforcement learning, 2025. URL https://arxiv.org/abs/2509.07980.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024b. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. GPTSwarm: Language agents as optimizable graphs. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 62743–62767. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/zhuge24a.html.

A MPLM Inference Psuedo-code

Algorithm 1 MPI-style Parallel Decoding (wait-for-all version)

Require: Inference engine $\mathcal{E}$ (vLLM) that decodes multiple threads in parallel
Require: Single initial prompt `prompt` for thread 0
`Running, Runnable` $\leftarrow \{0\}, \{0\}$ {all alive threads}
`Inbox`[0], `WaitingFor`[0] $\leftarrow [\,], [\,]$
while `Running` $\neq \emptyset$ **do**
  $\mathcal{E}$ decodes all threads in `Runnable` in parallel until a directive boundary (or EOS)
  for all $i \in$ `Runnable` **do** {conceptually in parallel}
    if thread i reached a directive boundary (or EOS) **then**
      Parse one event e_i from thread i
      if e_i is `<spawn [`$\mathcal{J}$`]> prompt' </spawn>` **then**
        for all $j \in \mathcal{J}$ **do**
          `Running, Runnable` $\leftarrow$ `Running` $\cup \{j\},$ `Runnable` $\cup \{j\}$
          `Inbox`[j], `WaitingFor`[j] $\leftarrow [\,], [\,]$
          Submit `prompt'` to $\mathcal{E}$ as thread j
        end for
      end if
      if e_i is `<send [`$\mathcal{K}$`]> msg </send>` **then**
        for all $k \in \mathcal{K}$ **do**
          Append $(i,$ `msg`$)$ to `Inbox`[k]
        end for
      end if
      if e_i is `<recv [`$\mathcal{R}$`]>` **then**
        `WaitingFor`[i] $\leftarrow \mathcal{R}$
        `Runnable` $\leftarrow$ `Runnable` $\setminus \{i\}$ {block immediately on recv}
      end if
      if e_i is `<stop>` **or** EOS **then**
        `Running, Runnable` $\leftarrow$ `Running` $\setminus \{i\},$ `Runnable` $\setminus \{i\}$
        `WaitingFor`[i] $\leftarrow [\,]$
      end if
    end if
  end for
  `Blocked` $\leftarrow$ `Running` $\setminus$ `Runnable`
  for all $i \in$ `Blocked` **do** {conceptually in parallel}
    if `WaitingFor`[i] $= [\,]$ **then**
      `Runnable` $\leftarrow$ `Runnable` $\cup \{i\}$ {all required senders died}
    else
      `HaveSenders` $\leftarrow \{\, s \mid \exists (s, m) \in$ `Inbox`[i] $\,\}$
      if `WaitingFor`[i] $\subseteq$ `HaveSenders` **then**
        Consume the needed messages from `Inbox`[i] for all $s \in$ `WaitingFor`[i]
        `WaitingFor`[i] $\leftarrow [\,]$
        `Runnable` $\leftarrow$ `Runnable` $\cup \{i\}$
      end if
    end if
  end for
end while

B Additional Aspects of MPLMs

B.1 Implementation of `recv`

The `recv` directive requires inference for a thread to be stopped to ensure all incoming messages are accumulated. We note that `recv` can be implemented in various ways. The implementations we consider here are:

1. **Asynchronous (wait-for-all):** When a thread executes `recv(S)`, it blocks until it has received all required messages from all active threads in the set S . Once every required sender has produced a message for that worker, the thread becomes runnable immediately. This is the main version we consider in the work.

2. **Asynchronous (wait-for-any):** When a thread executes `recv(S)`, it blocks until it receives at least one message from any sender in S. The runtime then immediately unblocks the thread, allowing it to continue decoding with partial information.

These two implementations can have different use cases which could potentially benefit in different types of applications (see Appendix N for an example on a toy setup). It is also possible to have different directives for each of these implementations, but we leave this for future work.

B.2 Preemption via Early Subtree Cancellation

In addition to reducing aggregation cost, MPLMs support *preemption*. Here, we use *preemption* to mean early cancellation of active descendants once they can no longer affect the final output. This behavior is natural in MPLMs because the runtime maintains the spawn tree and thread identities are globally addressable, so messages can be sent not only between siblings but also across different levels of the threads.

Consider a thread p that spawns a set of children $C(p)$ to solve a hierarchical sub-problem. In many reasoning tasks, local decision of p is determined by a predicate over incoming child messages that may become satisfied before all children finish. For example, in OR-type search problems, a single positive certificate may already suffice. Once p has received enough information to determine its output, it can immediately send a message to its own parent (or any relevant ancestor) and terminate. The runtime then cancels the still-active descendants in the subtree rooted at p whose outputs are no longer reachable from a live consumer.

This capability is difficult to express in standard fork-join execution, where progress is tied to join directives, and differs from controller-based tree search, where branch pruning is externalized to a centralized controller. Importantly, MPLM preemption does not require introducing a new decoding primitive; it emerges from persistent thread identities, direct message passing, and runtime cancellation of unreachable descendants.

B.3 Context Management via Respawning

The `spawn` directive can be used in a special manner where a thread can spawn a thread that is already existing, including itself. Note that in this case, the controller merely creates a new thread with the same logical identity but with a different prompt. We term this capability: *respawning*. This capability can be used by the LLM in interesting ways. For instance, a worker can construct a compact *inheritance payload* that summarizes the minimal information required for future progress (e.g., verified intermediate results, current tasks or constraints) and then respawn. By discarding the redundant history, the newly spawned worker begins from this curated and compressed context, effectively resetting its working memory without losing task-critical state.

This is highly beneficial because an MPLM can summarize its own context resulting in: **(i) Bounded per-step cost.** By controlling the context length of each worker, respawning prevents per-token computation from becoming unbounded over time, reducing wall-clock latency for long-running tasks. **(ii) Unbounded reasoning.** Repeatedly summarizing each worker's state and continuing from a refreshed context could, in theory, allow threads to run for arbitrarily many iterations without exceeding the model's context window.

C Sudoku Generation Procedure

For generating Sudoku puzzles, we use a deletion probability p which is a measure of the *sparsity* of the puzzle. The higher the value of p, the sparser it will be and in general would require more iterations to solve using the naked-singles strategy. In our work, we use $p = 1.0$ for both training and inference.

Algorithm 2 Sudoku Dataset Generation (Naked-Single Solvable)

Require: Block size `base_n`, grid size $N \leftarrow$ `base_n`^{2}
Require: Target size `size`, deletion probability $p \in [0, 1]$
$\mathcal{D} \leftarrow [\,]$
while $|\mathcal{D}| <$ `size` **do**
  `solution` $\leftarrow$ RandomSolvedGrid(N, `base_n`)
  `original` $\leftarrow$ `solution`
  `cells` $\leftarrow$ all cell indices (r, c)
  for all $(r, c) \in$ `cells` **do**
    `saved` $\leftarrow$ `original`$[r][c]$; `original`$[r][c] \leftarrow 0$
    if NakedSingleSolvable(`original`, `solution`, `base_n`) **then**
      {with probability p remove this cell, otherwise restore it}
      if Bernoulli(p) $= 0$ **then**
        `original`$[r][c] \leftarrow$ `saved`
      end if
    else
      `original`$[r][c] \leftarrow$ `saved`
    end if
  end for
  Append (`original`, `solution`) to $\mathcal{D}$
end while
return $\mathcal{D}$

Predicate NakedSingleSolvable(`original`, `solution`, `base_n`)
$N \leftarrow$ `base_n`^{2}; `board` $\leftarrow$ copy of `original`
while there exists an empty cell in `board` that has exactly one feasible value **do**
  Pick such a cell (r, c)
  Compute `used` $\leftarrow$ numbers already present in row r, column c, and the `base_n` $\times$ `base_n` box of (r, c)
  Let `cand` $\leftarrow \{1, \ldots, N\} \setminus$ `used`
  if $|$`cand`$| \neq 1$ **then**
    Continue
  end if
  Fill `board`$[r][c]$ with the unique value in `cand`
end while
return (`board` $=$ `solution`)

C.1 Training details

We generate CoT for each of the training paradigms using a simple Python program. An example of the type of CoTs we use is given in Sections E to G. This gives us a dataset of (prompt, response) pairs which we use for SFT using the LlamaFactory (Zheng et al., 2024b) library. We use a batch size of 128 with a learning rate of 10^{-5} using the Adam (Kingma and Ba, 2015) optimizer. We observe that as we scale the size of the Sudoku puzzle, the master traces in the CoT become less populous which causes lower learning signal on them and the model fails to perform the work required by the master. In order to counteract this, we upweight the number of master samples in the dataset by a factor of 5 for $N = 3$, a factor of 20 for $N = 4$ and 100 for $N = 5$. We choose the model based on the validation token accuracy on a subset of 5k prompts.

D SAT Training Details

For SAT, satisfiable instances are generated by first sampling a random assignment and then constructing clauses that are consistent with it, whereas unsatisfiable instances are produced by random sampling and verified using a symbolic SAT solver. Then, we generate CoT for each of the training paradigms using a simple Python program. An example of the type of CoTs we use is given in Sections H to J. This gives us a dataset of (prompt, response) pairs which we use for SFT using the LlamaFactory (Zheng et al., 2024b) library. We use a batch size of 128 with a learning rate of 10^{-5} using the Adam (Kingma and Ba, 2015) optimizer. We construct a training set of 10,000 instances by sampling from problems with 8 to 20 variables. To better support performance on harder instances, we upsample problems with larger numbers of variables during training. We then evaluate the resulting model on test sets consisting of 100 instances for each variable count from 8 to 20. The SAT accuracy results are reported below.

Model	8	9	10	11	12	13	14	15	16	17	18	19	20
MPLM	100%	91%	99%	91%	95%	94%	92%	87%	92%	89%	90%	86%	85%
FJ	97%	95%	94%	93%	91%	93%	90%	91%	91%	88%	90%	85%	86%
Serial	100%	95%	98%	93%	94%	✗	✗	✗	✗	✗	✗	✗	✗

Table 3: Accuracy on SAT problems per variable count (8–20). A red cross (✗) denotes infeasible evaluation due to context length constraints.

E Sample Sudoku Chain-of-Thoughts (MPLM)

This section provides representative traces from the master thread and a worker thread. We display the raw directive syntax used by the runtime, including <spawn>, <recv>, <send>, and <stop>. For readability, we truncate long candidate maps.

E.1 Master Trace (Initialization)

Master prompt.

```
Solve this Sudoku:
{ (0,0): {6}, (0,1): {8}, (0,2): {9}, (0,3): {4}, (0,4): {3}, (0,5): {1},
  (0,6): {2}, (0,8): {7}, (1,0): {1}, (1,1): {4}, (1,2): {3}, ...,
  (8,8): {4} }
```

Master response.

```
<spawn [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
        27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
        50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,
        73,74,75,76,77,78,79,80,81]>
{ (0,0): {6}, (0,1): {8}, (0,2): {9}, (0,3): {4}, (0,4): {3}, (0,5): {1},
  (0,6): {2}, (0,8): {7}, (1,0): {1}, (1,1): {4}, (1,2): {3}, ...,
  (8,8): {4} }
</spawn>
<recv [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
       27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
       50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,
       73,74,75,76,77,78,79,80,81]>
```

E.2 Worker Trace (Example: Worker 42)

Worker prompt (first turn).

```
Your id is: 42.
{ (0,0): {6}, (0,1): {8}, (0,2): {9}, (0,3): {4}, (0,4): {3}, (0,5): {1},
  (0,6): {2}, (0,8): {7}, (1,0): {1}, (1,1): {4}, (1,2): {3}, ...,
  (8,8): {4} }
```

Worker response (first turn).

```
My id is 42 which corresponds to cell (4,5). The cell is empty.
The remaining candidates are: {1, 2, 3, 4, 5, 6, 7, 8, 9}.
<recv>
```

Worker prompt (after receiving neighbor messages).

```
Your id is: 42.
{ (0,0): {6}, (0,1): {8}, (0,2): {9}, (0,3): {4}, (0,4): {3}, (0,5): {1},
  (0,6): {2}, (0,8): {7}, (1,0): {1}, (1,1): {4}, (1,2): {3}, ...,
  (8,8): {4} }
My id is 42 which corresponds to cell (4,5). The cell is empty.
The remaining candidates are: {1, 2, 3, 4, 5, 6, 7, 8, 9}.
<recv>
{ From 6: 1 }
{ From 31: 7 }
{ From 32: 2 }
...
{ From 78: 8 }
</recv>
```

Worker response (computes its value and reports).

```
The neighboring cells have sent the following values: {1, 2, 3, 4, 6, 7, 8, 9}.
The remaining candidates are: {5}.
<send [0,6,15,24,31,32,33,37,38,39,40,41,43,44,45,49,50,51,60,69,78]>
{ From 42: 5 }
</send>
<stop>
```

E.3 Master Trace (Termination)

Master prompt (after collecting worker outputs).

```
Solve this Sudoku:
{ (0,0): {6}, (0,1): {8}, (0,2): {9}, (0,3): {4}, (0,4): {3}, (0,5): {1},
  (0,6): {2}, (0,8): {7}, (1,0): {1}, (1,1): {4}, (1,2): {3}, ...,
  (8,8): {4} }
<spawn [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
        27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
        50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,
        73,74,75,76,77,78,79,80,81]>{ ... }</spawn>
<recv [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
       27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,
       50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,
       73,74,75,76,77,78,79,80,81]>
{ From 1: 6 }
{ From 2: 8 }
{ From 3: 9 }
...
{ From 81: 4 }
</recv>
```

Master response (final).

```
<stop>
[[6, 8, 9, 4, 3, 1, 2, 5, 7],
 [1, 4, 3, 2, 5, 7, 8, 9, 6],
 [7, 2, 5, 8, 9, 6, 4, 3, 1],
 [5, 1, 4, 7, 2, 9, 6, 8, 3],
 [3, 6, 8, 1, 4, 5, 7, 2, 9],
 [9, 7, 2, 6, 8, 3, 1, 4, 5],
 [4, 3, 6, 5, 1, 2, 9, 7, 8],
```

```
 [8, 9, 7, 3, 6, 4, 5, 1, 2],
 [2, 5, 1, 9, 7, 8, 3, 6, 4]]
</stop>
```

F Sample Sudoku Chain-of-Thoughts (Fork-Join)

This section provides representative traces from the master thread and a worker thread under the Fork/Join (FJ) scheduling setting. We display the raw directive syntax used by the runtime, including <spawn>, <recv>, <send>, and <stop>. For readability, we truncate long candidate maps.

F.1 Master Trace (Initialization)

Master prompt.

```
Solve this Sudoku:
{ (0,0): {6}, (0,5): {1}, (0,7): {5}, (1,2): {3}, (1,3): {2},
  (2,0): {7}, (2,1): {2}, (2,2): {5}, (2,6): {4}, (3,2): {4},
  (3,6): {6}, (3,8): {3}, (4,7): {2}, (5,0): {9}, (5,1): {7},
  (5,3): {6}, (5,8): {5}, (6,1): {3}, (7,0): {8}, (7,3): {3},
  (7,4): {6}, (8,2): {1}, (8,3): {9}, (8,7): {6}, (8,8): {4} }
```

Master response.

```
<spawn [2,3,4,5,7,9,10,11,14,15,16,17,18,22,23,24,26,27,28,29,
        31,32,33,35,37,38,39,40,41,42,43,45,48,50,51,52,53,55,
        57,58,59,60,61,62,63,65,66,69,70,71,72,73,74,77,78,79]>
{ (0,0): {6}, (0,5): {1}, (0,7): {5}, (1,2): {3}, (1,3): {2},
  (2,0): {7}, (2,1): {2}, (2,2): {5}, (2,6): {4}, (3,2): {4},
  (3,6): {6}, (3,8): {3}, (4,7): {2}, (5,0): {9}, (5,1): {7},
  (5,3): {6}, (5,8): {5}, (6,1): {3}, (7,0): {8}, (7,3): {3},
  (7,4): {6}, (8,2): {1}, (8,3): {9}, (8,7): {6}, (8,8): {4} }
</spawn>
<recv [2,3,4,5,7,9,10,11,14,15,16,17,18,22,23,24,26,27,28,29,
       31,32,33,35,37,38,39,40,41,42,43,45,48,50,51,52,53,55,
       57,58,59,60,61,62,63,65,66,69,70,71,72,73,74,77,78,79]>
```

Master prompt (after receiving worker reports).

```
Solve this Sudoku:
{ (0,0): {6}, (0,5): {1}, (0,7): {5}, (1,2): {3}, (1,3): {2},
  (2,0): {7}, (2,1): {2}, (2,2): {5}, (2,6): {4}, (3,2): {4},
  (3,6): {6}, (3,8): {3}, (4,7): {2}, (5,0): {9}, (5,1): {7},
  (5,3): {6}, (5,8): {5}, (6,1): {3}, (7,0): {8}, (7,3): {3},
  (7,4): {6}, (8,2): {1}, (8,3): {9}, (8,7): {6}, (8,8): {4} }
<spawn [2,3,4,5,7,9,10,11,14,15,16,17,18,22,23,24,26,27,28,29,
        31,32,33,35,37,38,39,40,41,42,43,45,48,50,51,52,53,55,
        57,58,59,60,61,62,63,65,66,69,70,71,72,73,74,77,78,79]>{ ... }</spawn>
<recv [2,3,4,5,7,9,10,11,14,15,16,17,18,22,23,24,26,27,28,29,
       31,32,33,35,37,38,39,40,41,42,43,45,48,50,51,52,53,55,
       57,58,59,60,61,62,63,65,66,69,70,71,72,73,74,77,78,79]>
{ From 2: None }
{ From 3: None }
{ From 4: None }
{ From 5: None }
...
{ From 22: 8 }
...
{ From 74: 5 }
...
{ From 79: None }
</recv>
```

Master response (next fork/join step).

```
<spawn [2,3,4,5,7,9,10,11,14,15,16,17,18,23,24,26,27,28,29,31,
        32,33,35,37,38,39,40,41,42,43,45,48,50,51,52,53,55,57,
        58,59,60,61,62,63,65,66,69,70,71,72,73,77,78,79]>
{ (0,0): {6}, (0,5): {1}, (0,7): {5}, (1,2): {3}, (1,3): {2},
  (2,0): {7}, (2,1): {2}, (2,2): {5}, (2,3): {8}, (2,6): {4},
  (3,2): {4}, (3,6): {6}, (3,8): {3}, (4,7): {2}, (5,0): {9},
  (5,1): {7}, (5,3): {6}, (5,8): {5}, (6,1): {3}, (7,0): {8},
  (7,3): {3}, (7,4): {6}, (8,1): {5}, (8,2): {1}, (8,3): {9},
  (8,7): {6}, (8,8): {4} }
</spawn>
<recv [2,3,4,5,7,9,10,11,14,15,16,17,18,23,24,26,27,28,29,31,
       32,33,35,37,38,39,40,41,42,43,45,48,50,51,52,53,55,57,
       58,59,60,61,62,63,65,66,69,70,71,72,73,77,78,79]>
```

F.2 Worker Trace (Example: Worker 42)

Worker prompt (first turn).

```
Your id is: 42.
{ (0,0): {6}, (0,5): {1}, (0,7): {5}, (1,2): {3}, (1,3): {2},
  (2,0): {7}, (2,1): {2}, (2,2): {5}, (2,6): {4}, (3,2): {4},
  (3,6): {6}, (3,8): {3}, (4,7): {2}, (5,0): {9}, (5,1): {7},
  (5,3): {6}, (5,8): {5}, (6,1): {3}, (7,0): {8}, (7,3): {3},
  (7,4): {6}, (8,2): {1}, (8,3): {9}, (8,7): {6}, (8,8): {4} }
```

Worker response (first turn).

```
My id is 42 which corresponds to cell (4,5).
The neighboring cells already have the following values: {1, 2, 6}.
The remaining candidates are: {3, 4, 5, 7, 8, 9}.
<send [0]>{ From 42: None }</send><stop>
```

Worker prompt (later fork; after more iterations).

```
Your id is: 42.
{ (0,0): {6}, (0,2): {9}, (0,3): {4}, (0,4): {3}, (0,5): {1},
  (0,6): {2}, (0,7): {5}, (0,8): {7}, (1,0): {1}, (1,1): {4},
  (1,2): {3}, (1,3): {2}, (2,0): {7}, (2,1): {2}, (2,2): {5},
  (2,3): {8}, (2,4): {9}, (2,5): {6}, (2,6): {4}, (2,7): {3},
  (2,8): {1}, (3,0): {5}, (3,1): {1}, (3,2): {4}, (3,3): {7},
  (3,6): {6}, (3,8): {3}, (4,0): {3}, (4,1): {6}, (4,2): {8},
  (4,7): {2}, (4,8): {9}, ...,
  (8,7): {6}, (8,8): {4} }
```

Worker response (computes its value and reports).

```
My id is 42 which corresponds to cell (4,5).
The neighboring cells already have the following values: {1, 2, 3, 4, 6, 7, 8, 9}.
The remaining candidates are: {5}.
<send [0]>{ From 42: 5 }</send><stop>
```

F.3 Master Trace (Termination)

Master response (final stop).

```
<stop>
[[6, 8, 9, 4, 3, 1, 2, 5, 7],
 [1, 4, 3, 2, 5, 7, 8, 9, 6],
 [7, 2, 5, 8, 9, 6, 4, 3, 1],
 [5, 1, 4, 7, 2, 9, 6, 8, 3],
```

```
 [3, 6, 8, 1, 4, 5, 7, 2, 9],
 [9, 7, 2, 6, 8, 3, 1, 4, 5],
 [4, 3, 6, 5, 1, 2, 9, 7, 8],
 [8, 9, 7, 3, 6, 4, 5, 1, 2],
 [2, 5, 1, 9, 7, 8, 3, 6, 4]]
</stop>
```

G Sample Sudoku Chain-of-Thoughts (Serial)

This section provides a representative trace for the Serial baseline. Unlike the message-passing settings, Serial performs iterative constraint propagation in a single thread (round by round) and terminates with <stop>. For readability, we truncate long candidate maps.

G.1 Serial Trace (Rounds)

Serial prompt.

```
Solve this Sudoku:
{ (0,0): {6}, (0,5): {1}, (0,7): {5}, (1,2): {3}, (1,3): {2},
  (2,0): {7}, (2,1): {2}, (2,2): {5}, (2,6): {4}, (3,2): {4},
  (3,6): {6}, (3,8): {3}, (4,7): {2}, (5,0): {9}, (5,1): {7},
  (5,3): {6}, (5,8): {5}, (6,1): {3}, (7,0): {8}, (7,3): {3},
  (7,4): {6}, (8,2): {1}, (8,3): {9}, (8,7): {6}, (8,8): {4} }
```

Serial response

```
Round 1:
For cell (0,1), the neighboring cells have the values {1,2,3,5,6,7}.
The remaining candidates are: {4,8,9}
For cell (0,2), the neighboring cells have the values {1,2,3,4,5,6,7}.
The remaining candidates are: {8,9}
For cell (0,3), the neighboring cells have the values {1,2,3,5,6,9}.
The remaining candidates are: {4,7,8}
...
For cell (8,1), the neighboring cells have the values {1,2,3,4,6,7,8,9}.
The remaining candidates are: {5}
...
The current remaining candidates of the sudoku is :
{ (0,0): {6}, (0,1): {4,8,9}, (0,2): {8,9}, (0,3): {4,7,8}, ... }

...

Round 21:
For cell (5,5), the neighboring cells have the values {1,2,4,5,6,7,8,9}.
The remaining candidates are: {3}
For cell (6,4), the neighboring cells have the values {2,3,4,5,6,7,8,9}.
The remaining candidates are: {1}
For cell (6,6), the neighboring cells have the values {1,2,3,4,5,6,7,8}.
The remaining candidates are: {9}
For cell (8,4), the neighboring cells have the values {1,2,3,4,5,6,8,9}.
The remaining candidates are: {7}
For cell (8,6), the neighboring cells have the values {1,2,4,5,6,7,8,9}.
The remaining candidates are: {3}
The current remaining candidates of the sudoku is :
{ (0,0): {6}, (0,1): {8}, (0,2): {9}, (0,3): {4}, (0,4): {3}, ...,
  (8,6): {3}, (8,7): {6}, (8,8): {4} }
<stop>
[[6, 8, 9, 4, 3, 1, 2, 5, 7],
 [1, 4, 3, 2, 5, 7, 8, 9, 6],
 [7, 2, 5, 8, 9, 6, 4, 3, 1],
 [5, 1, 4, 7, 2, 9, 6, 8, 3],
 [3, 6, 8, 1, 4, 5, 7, 2, 9],
 [9, 7, 2, 6, 8, 3, 1, 4, 5],
 [4, 3, 6, 5, 1, 2, 9, 7, 8],
 [8, 9, 7, 3, 6, 4, 5, 1, 2],
 [2, 5, 1, 9, 7, 8, 3, 6, 4]]
</stop>
```

H Sample 3-SAT Chain-of-Thoughts (MPLM)

This section provides representative traces from the root agent and a worker agent on a 5-variable 3-SAT instance (answer: SAT). The root branches on x_1; each child branches on x_2; each grandchild is a leaf solved by unit propagation. In MPLM, each parent performs a *asynchronous receive*: it reads child results one at a time and stops as soon as the subproblem is determined. We display the raw directive syntax including `<spawn>`, `<recv>`, `<send>`, and `<stop>`.

H.1 Root Agent Trace

Root prompt.

```
( 2 v -1 v 4 ) ^ ( 4 v 5 v -2 ) ^ ( -2 v 3 v 1 ) ^ ( -5 v -2 v 4 ) ^
( 4 v 3 v -1 ) ^ ( -5 v 1 v -4 ) ^ ( 4 v -2 v 3 ) ^ ( -5 v 2 v -3 ) ^
( -5 v 2 v 1 ) ^ ( -5 v -1 v -3 ) ^ ( 4 v -2 v -1 ) ^ ( 4 v 5 v -2 ) ^
( 1 v 3 v 2 ) ^ ( -2 v 4 v 1 ) ^ ( 1 v -5 v -4 ) ^ ( 2 v -1 v -3 ) ^
( -1 v -5 v -4 ) ^ ( -1 v 4 v -5 ) ^ ( -4 v -5 v 3 ) ^ ( -1 v 4 v -2 ) ^
( -2 v 4 v 3 )
```

Root response (spawns two workers, waits for first result).

```
Try unit propagation.
No unit clauses.
Branch on x1.
<spawn [0.0]>
( 4 v 5 v -2 ) ^ ( -2 v 3 ) ^ ( -5 v -2 v 4 ) ^ ( -5 v -4 ) ^
( 4 v -2 v 3 ) ^ ( -5 v 2 v -3 ) ^ ( -5 v 2 ) ^ ( 4 v 5 v -2 ) ^
( 3 v 2 ) ^ ( -2 v 4 ) ^ ( -5 v -4 ) ^ ( -4 v -5 v 3 ) ^ ( -2 v 4 v 3 )
</spawn>
<spawn [0.1]>
( 2 v 4 ) ^ ( 4 v 5 v -2 ) ^ ( -5 v -2 v 4 ) ^ ( 4 v 3 ) ^
( 4 v -2 v 3 ) ^ ( -5 v 2 v -3 ) ^ ( -5 v -3 ) ^ ( 4 v -2 ) ^
( 4 v 5 v -2 ) ^ ( 2 v -3 ) ^ ( -5 v -4 ) ^ ( 4 v -5 ) ^
( -4 v -5 v 3 ) ^ ( 4 v -2 ) ^ ( -2 v 4 v 3 )
</spawn>
<recv>
```

Root prompt (after receiving SAT from worker 0.0; worker 0.1 is preempted).

```
( 2 v -1 v 4 ) ^ ... ^ ( -2 v 4 v 3 )
Try unit propagation.
No unit clauses.
Branch on x1.
<spawn [0.0]>
...
</spawn>
<spawn [0.1]>
...
</spawn>
<recv>
{SAT.}     % only 1 of 2 child results received; worker 0.1 is preempted
</recv>
```

Root response.

```
<stop>SAT.</stop>
```

H.2 Worker Agent Trace (Example: Worker 0.1)

Worker 0.1 prompt (first turn).

```
Your id is: 0.1.
( 2 v 4 ) ^ ( 4 v 5 v -2 ) ^ ( -5 v -2 v 4 ) ^ ( 4 v 3 ) ^
( 4 v -2 v 3 ) ^ ( -5 v 2 v -3 ) ^ ( -5 v -3 ) ^ ( 4 v -2 ) ^
( 4 v 5 v -2 ) ^ ( 2 v -3 ) ^ ( -5 v -4 ) ^ ( 4 v -5 ) ^
( -4 v -5 v 3 ) ^ ( 4 v -2 ) ^ ( -2 v 4 v 3 )
```

Worker 0.1 response (spawns subworkers, waits for first result).

```
Try unit propagation.
No unit clauses.
Branch on x2.
<spawn [0.1.0]>
( 4 ) ^ ( 4 v 3 ) ^ ( -5 v -3 ) ^ ( -5 v -3 ) ^ ( -3 ) ^ ( -5 v -4 ) ^
( 4 v -5 ) ^ ( -4 v -5 v 3 )
</spawn>
<spawn [0.1.1]>
( 4 v 5 ) ^ ( -5 v 4 ) ^ ( 4 v 3 ) ^ ( 4 v 3 ) ^ ( -5 v -3 ) ^ ( 4 ) ^
( 4 v 5 ) ^ ( -5 v -4 ) ^ ( 4 v -5 ) ^ ( -4 v -5 v 3 ) ^ ( 4 ) ^ ( 4 v 3 )
</spawn>
<recv>
```

Worker 0.1 prompt (after receiving SAT from worker 0.1.0; worker 0.1.1 is preempted).

```
Your id is: 0.1.
( 2 v 4 ) ^ ... ^ ( -2 v 4 v 3 )
Try unit propagation.
No unit clauses.
Branch on x2.
<spawn [0.1.0]>
...
</spawn>
<spawn [0.1.1]>
...
</spawn>
<recv>
{SAT.}     % only 1 of 2 child results received; worker 0.1.1 is preempted
</recv>
```

Worker 0.1 response (concludes and reports to root).

```
Conclusion for this subproblem: SAT.
<send [0]>{SAT.}</send> <stop>
```

I Sample 3-SAT Chain-of-Thoughts (FJ)

This section shows the same 5-variable instance solved by Fork-Join. In FJ, the `<recv [...]>` tag lists *all* expected senders; the parent blocks until every child result has arrived before aggregating. For readability we truncate the CNF clauses in the continuation prompts.

I.1 Root Agent Trace

Root response (spawns two workers, waits for *both*).

```
Try unit propagation.
No unit clauses.
Branch on x1.
<spawn [0.0]>
( 4 v 5 v -2 ) ^ ( -2 v 3 ) ^ ( -5 v -2 v 4 ) ^ ( -5 v -4 ) ^
( 4 v -2 v 3 ) ^ ( -5 v 2 v -3 ) ^ ( -5 v 2 ) ^ ( 4 v 5 v -2 ) ^
( 3 v 2 ) ^ ( -2 v 4 ) ^ ( -5 v -4 ) ^ ( -4 v -5 v 3 ) ^ ( -2 v 4 v 3 )
</spawn>
<spawn [0.1]>
```

```
( 2 v 4 ) ^ ( 4 v 5 v -2 ) ^ ( -5 v -2 v 4 ) ^ ( 4 v 3 ) ^
( 4 v -2 v 3 ) ^ ( -5 v 2 v -3 ) ^ ( -5 v -3 ) ^ ( 4 v -2 ) ^
( 4 v 5 v -2 ) ^ ( 2 v -3 ) ^ ( -5 v -4 ) ^ ( 4 v -5 ) ^
( -4 v -5 v 3 ) ^ ( 4 v -2 ) ^ ( -2 v 4 v 3 )
</spawn>
<recv [0.0, 0.1]>          % waits for ALL children before proceeding
```

Root prompt (after receiving SAT from *both* workers 0.0 and 0.1).

```
( 2 v -1 v 4 ) ^ ... ^ ( -2 v 4 v 3 )
Try unit propagation.
No unit clauses.
Branch on x1.
<spawn [0.0]>
...
</spawn>
<spawn [0.1]>
...
</spawn>
<recv [0.0, 0.1]>
{SAT.}
{SAT.}     % both child results received before aggregating
</recv>
```

Root response.

```
<stop>SAT.</stop>
```

I.2 Worker Agent Trace (Example: Worker 0.1)

Worker 0.1 response (spawns subworkers, waits for *both*).

```
Try unit propagation.
No unit clauses.
Branch on x2.
<spawn [0.1.0]>
( 4 ) ^ ( 4 v 3 ) ^ ( -5 v -3 ) ^ ( -5 v -3 ) ^ ( -3 ) ^ ( -5 v -4 ) ^
( 4 v -5 ) ^ ( -4 v -5 v 3 )
</spawn>
<spawn [0.1.1]>
( 4 v 5 ) ^ ( -5 v 4 ) ^ ( 4 v 3 ) ^ ( 4 v 3 ) ^ ( -5 v -3 ) ^ ( 4 ) ^
( 4 v 5 ) ^ ( -5 v -4 ) ^ ( 4 v -5 ) ^ ( -4 v -5 v 3 ) ^ ( 4 ) ^ ( 4 v 3 )
</spawn>
<recv [0.1.0, 0.1.1]>      % waits for ALL children before proceeding
```

Worker 0.1 prompt (after receiving SAT from *both* workers 0.1.0 and 0.1.1).

```
Your id is: 0.1.
( 2 v 4 ) ^ ... ^ ( -2 v 4 v 3 )
Try unit propagation.
No unit clauses.
Branch on x2.
<spawn [0.1.0]>
...
</spawn>
<spawn [0.1.1]>
...
</spawn>
<recv [0.1.0, 0.1.1]>
{SAT.}
{SAT.}     % both child results received before aggregating
</recv>
```

Worker 0.1 response (concludes and reports to root).

```
Conclusion for this subproblem: SAT.
<send [0]>{SAT.}</send> <stop>
```

J Sample 3-SAT Chain-of-Thoughts (Serial)

The serial DPLL solver branches depth-first on a single thread with no message passing.

J.1 Solver Trace

Prompt.

```
( 2 v -1 v 4 ) ^ ( 4 v 5 v -2 ) ^ ( -2 v 3 v 1 ) ^ ( -5 v -2 v 4 ) ^
( 4 v 3 v -1 ) ^ ( -5 v 1 v -4 ) ^ ( 4 v -2 v 3 ) ^ ( -5 v 2 v -3 ) ^
( -5 v 2 v 1 ) ^ ( -5 v -1 v -3 ) ^ ( 4 v -2 v -1 ) ^ ( 4 v 5 v -2 ) ^
( 1 v 3 v 2 ) ^ ( -2 v 4 v 1 ) ^ ( 1 v -5 v -4 ) ^ ( 2 v -1 v -3 ) ^
( -1 v -5 v -4 ) ^ ( -1 v 4 v -5 ) ^ ( -4 v -5 v 3 ) ^ ( -1 v 4 v -2 ) ^
( -2 v 4 v 3 )
```

Response.

```
Branch on x1.
Branch x1=0
CNF:
( 4 v 5 v -2 ) ^ ( -2 v 3 ) ^ ( -5 v -2 v 4 ) ^ ( -5 v -4 ) ^
( 4 v -2 v 3 ) ^ ( -5 v 2 v -3 ) ^ ( -5 v 2 ) ^ ( 4 v 5 v -2 ) ^
( 3 v 2 ) ^ ( -2 v 4 ) ^ ( -5 v -4 ) ^ ( -4 v -5 v 3 ) ^ ( -2 v 4 v 3 )
  Branch on x2.
  Branch x2=0
  CNF:
  ( -5 v -4 ) ^ ( -5 v -3 ) ^ ( -5 ) ^ ( 3 ) ^ ( -5 v -4 ) ^ ( -4 v -5 v 3 )
    Unit assignments: x5=False, x3=True
    CNF:
    TRUE
    Result: SAT.
<stop>SAT.</stop>
```

K Extended Theoretical Results

To understand the benefits of our framework, we consider the maximum number of context tokens, and the total number of generated tokens as the metrics of importance, and use it to understand the asymptotic behavior of the MPLM framework and show how it outperforms the FJ framework. Assume that we are trying to solve an $N^2 \times N^2$ sudoku. We analyze the CoT of both the parent and children threads for our analysis.

K.1 Proof of Theorem 2

Proof. We bound the context length required by each type of thread under all execution models.

Serial CoT. In the serial model, a single LLM thread performs all computation iteratively in context. Therefore, in one iteration, it generates at most W tokens and then an at most $|M|$ token long message for each of the N workers. Therefore, summing over all T iterations, the context consumed is:

$$C_{\max}^{\text{Serial}} = \mathcal{O}(TN(W + |M|)).$$

Since from Assumption 1, $W = \mathcal{O}(k|M|)$, we have

$$\boxed{C_{\max}^{\text{Serial}} = \mathcal{O}(TNk|M|)}$$

Fork-join.. In the fork-join (FJ) model, worker threads are respawned at every iteration, so their contexts do not persist across iterations. In a single iteration, a worker consumes its prompt and produces local work plus its outgoing message, for a total of at most $W + |M|$ tokens. Thus, the maximum worker context is

$$C_{\text{worker}}^{\text{FJ}} = \mathcal{O}(W + |M|).$$

The master thread, however, persists across all iterations. In each iteration, it receives one message of size $|M|$ from each of the N ranks, resulting in $O(N|M|)$ new tokens per iteration. Over T iterations, the master accumulates

$$C_{\text{master}}^{\text{FJ}} = \mathcal{O}(TN|M|).$$

Taking the maximum over all threads yields

$$C_{\max}^{\text{FJ}} = \mathcal{O}(\max\{\, W + |M|,\ TN|M| \,\}).$$

Since, from Assumption 1, we know that $W = \mathcal{O}(k|M|), W + |M| = \mathcal{O}(k|M|)$. Therefore,

$$\boxed{C_{\max}^{\text{FJ}} = \mathcal{O}(TN|M|)}$$

MPLM-style persistence.. In the MPLM model, each rank maintains a persistent context across iterations. Fix any rank i. In each iteration, rank i:

1. performs local work contributing $\mathcal{O}(W)$ tokens;
2. receive messages of size $|M|$ with at most k neighbors, contributing $\mathcal{O}(k|M|)$ tokens;
3. sending a $|M|$ token long message which would require $\mathcal{O}(|M| + k)$ control tokens for the message and communication bookkeeping.

Thus, the per-iteration context growth is

$$\mathcal{O}(W + k + |M| + k|M|) = \mathcal{O}(W + k|M|),$$

where lower-order control terms are absorbed.

Since the context persists for T iterations, the total context of rank i is

$$C_i^{\text{MPLM}} = \mathcal{O}(T(W + k|M|)) = \mathcal{O}(TW + Tk|M|).$$

Taking the maximum over all ranks yields

$$C_{\max}^{\text{MPLM}} = \mathcal{O}(TW + Tk|M|),$$

From Assumption 1, we know that $W = \mathcal{O}(k|M|)$, therefore,

$$\boxed{C_{\max}^{\text{MPLM}} = \mathcal{O}(Tk|M|)},$$

□

K.2 Sudoku-specific analysis

In the following section, we derive asymptotic rates for the maximum context required (C), total tokens generated (G) and the number of sequential tokens (S) for the Fork-Join paradigm and MPLMs.

K.2.1 Analysis of FJ

Parent: The Chain-of-Thought for the parent is of the following form:

$$[\textit{Initial sudoku}][\textit{spawn}_1][\textit{message}_1][\textit{spawn}_2][\textit{message}_2]....[\textit{spawn}_T][\textit{message}_T][\textit{Final sudoku}]$$

where T is the number of iterations required to complete the Sudoku using the naked-singles strategy. Note that the text in blue refers to the tokens *generated* by the model. Now, consider the maximum context (C) of the parent:

$$C_{parent} = 2C_{sudoku} + T \cdot C_{spawn} + T \cdot C_{message}$$

where C_{sudoku}, C_{spawn} and $C_{message}$ are the number of tokens consumed by these parts of the CoT. Since the Sudoku has N^4 entries, it consumes $\mathcal{O}(N^4)$ tokens. Since spawning requires sending the partially solved Sudoku to all the workers, copying it into the CoT costs $\mathcal{O}(N^4)$ tokens. The cost of the message is proportional to the number of live workers which is atmost $\mathcal{O}(N^4)$, therefore, $C_{message}$ is also $\mathcal{O}(N^4)$.

$$\Rightarrow C_{parent} = \mathcal{O}(N^4) + \mathcal{O}(T \cdot N^4) + \mathcal{O}(T \cdot N^4)$$

$$\Rightarrow \boxed{C_{parent} = \mathcal{O}(T \cdot N^4)}$$

We also compute the number of tokens generated (G) by the model which is:

$$G_{parent} = T \cdot C_{spawn} + C_{sudoku}$$
$$\Rightarrow G_{parent} = \mathcal{O}(T \cdot N^4) + \mathcal{O}(N^4)$$
$$\Rightarrow \boxed{G_{parent} = \mathcal{O}(T \cdot N^4)}$$

Child: The CoT for the child is of the following form:

$$[\textit{Current sudoku}][\textit{Response}]$$

where the Response is if the corresponding cell can be filled or not. For the child,

$$C_{child} = C_{sudoku} + C_{response}$$

Computing the response requires listing the possible values left for the cell which is $\mathcal{O}(N^2)$. Therefore, $C_{response}$ is $\mathcal{O}(N^2)$.

$$\Rightarrow C_{child} = \mathcal{O}(N^4) + \mathcal{O}(N^2)$$

$$\Rightarrow \boxed{C_{child} = \mathcal{O}(N^4)}$$

Now, consider the total number of tokens generated by *all* the workers. Say that the i^{th} worker runs for T_i iterations, then:

$$G_{child_i} = \mathcal{O}(T_i \cdot N^2)$$

Therefore, combining the tokens from all the workers gives:

$$\boxed{G_{child} = \mathcal{O}(T_{total} \cdot N^2)}$$

where $T_{total} = \sum_{i=1}^{N^4} T_i$

Using the above analysis, its evident that in FJ, the context is dominated by the parent which is $\mathcal{O}(T \cdot N^4)$. Therefore, we obtain:

$$\boxed{C^{FJ} = \mathcal{O}(T \cdot N^4)} \tag{1}$$

$$\boxed{G^{FJ} = \mathcal{O}(T \cdot N^4 + T_{total} \cdot N^2)} \tag{2}$$

K.2.2 Analysis of MPLM

Parent: The CoT for the parent in MPLM is of the following form:

$$[Initial\ sudoku][spawn][message][Final\ sudoku]$$

Therefore,

$$C_{parent} = 2 \cdot C_{sudoku} + C_{spawn} + C_{message}$$

The cost of the Sudoku is still $\mathcal{O}(N^4)$. The cost of spawning the workers and message is also $\mathcal{O}(N^4)$.

$$\Rightarrow C_{parent} = 2 \cdot \mathcal{O}(N^4) + \mathcal{O}(N^4) + \mathcal{O}(N^4)$$

$$\Rightarrow \boxed{C_{parent} = \mathcal{O}(N^4)}$$

We also compute the total tokens generated by the master which is

$$G_{parent} = C_{spawn} + C_{sudoku}$$

$$\Rightarrow \boxed{G_{parent} = \mathcal{O}(N^4)}$$

Child: Since we are interested in the maximum context consumed, consider the child which runs for the maximum number of iterations. The CoT for the child in MPLM is of the form:

$$[Initial\ sudoku][send_1][message_2][send_2][message_2]...[send_T]$$

Therefore,

$$C_{child} = C_{sudoku} + (T-1) \cdot C_{message} + T \cdot C_{send}$$

The cost of the message in MPLM is $\mathcal{O}(N^2)$ since each worker only receives messages from its neighbors, that is, workers in its rows, columns, and grid. Similary, the cost of sending a message is also $\mathcal{O}(N^2)$. Therefore,

$$C_{child} = \mathcal{O}(N^4) + \mathcal{O}((T-1) \cdot N^2) + \mathcal{O}(T \cdot N^2)$$

$$\Rightarrow \boxed{C_{child} = \mathcal{O}(N^4 + T \cdot N^2)}$$

Now, we calculate the number of tokens generated by the i^{th} worker.

$$G_{child_i} = T_i \cdot C_{send}$$

$$\Rightarrow G_{child_i} = \mathcal{O}(T_i \cdot N^2)$$

Therefore, summming over all children,

$$\Rightarrow \boxed{G_{child} = \mathcal{O}(T_{total} \cdot N^2)}$$

where $T_{total} = \sum_{i=1}^{N^4} T_i$.

Therefore, for MPLM we have

$$\boxed{C^{MPLM} = \mathcal{O}(N^4 + T \cdot N^2)} \tag{3}$$

$$\boxed{G^{MPLM} = \mathcal{O}(N^4 + T_{total} \cdot N^2} \tag{4}$$

Using Equations Equations (1) to (4) we observe that MPLM is better both in terms of maximum context required and the total number of tokens generated.

K.2.3 Sequential tokens

Number of sequential tokens represents the maximum causally dependent tokens. We measure sequential tokens as the sum, over iterations, of the maximum number of newly generated tokens among all agents in that iteration. Let $G_{i,t}$ denote the number of generated tokens produced by agent i at iteration t (including the master and workers),

and let T be the number of iterations required to complete the Sudoku using the naked-singles strategy. We define

$$S \;=\; \sum_{t=0}^{T} \max_i G_{i,t}.$$

FJ. In FJ, the master participates every iteration, it generates a `spawn` containing the (partially solved) Sudoku and aggregates worker messages. Per iteration, the master generates $\mathcal{O}(N^4)$ tokens for `spawn`, and the workers computes the response by listing the possible values left for the cell, which is $\mathcal{O}(N^2)$ tokens. So

$$\max_i G_{i,t} = \begin{cases} \mathcal{O}(N^4 + N^2), & 0 \le t \le T-1, \\ \mathcal{O}(N^4), & t = T. \end{cases}$$

Thus,

$$S_{\text{FJ}} = \mathcal{O}(T \cdot (N^4 + N^2)) + \mathcal{O}(N^4)$$

$$\boxed{S_{\text{FJ}} = \mathcal{O}\big(TN^4\big)}. \tag{5}$$

MPLM.. In MPLM, the master generates a single initial `spawn` at $t = 0$, workers exchange only local neighbor information during intermediate iterations, and the master outputs the final solved Sudoku at termination. So,

$$\max_i G_{i,t} = \begin{cases} \mathcal{O}(N^4), & t = 0, \\ \mathcal{O}(N^2), & 1 \le t \le T-1, \\ \mathcal{O}(N^4), & t = T. \end{cases}$$

Therefore if S denotes number of sequential tokens, then:

$$S_{\text{MPLM}} = \mathcal{O}(N^4) + (T-1)\mathcal{O}(N^2) + \mathcal{O}(N^4)$$

$$\Rightarrow \boxed{S_{\text{MPLM}} = \mathcal{O}(N^4 + TN^2)} \tag{6}$$

From Equations (5) to (6), we observe that for the number of sequential tokens as well, MPLMs are asymptotically better.

L Respawning Improves Efficiency and Enables Unbounded Execution

In our Sudoku setting, a worker is responsible for a single cell and its entire logical state can be summarized by the set of remaining feasible values for that cell. Therefore, after any number of iterations, the state of the worker can be summarized concisely and the worker can be respawned with a prompt containing the set of remaining possible values.

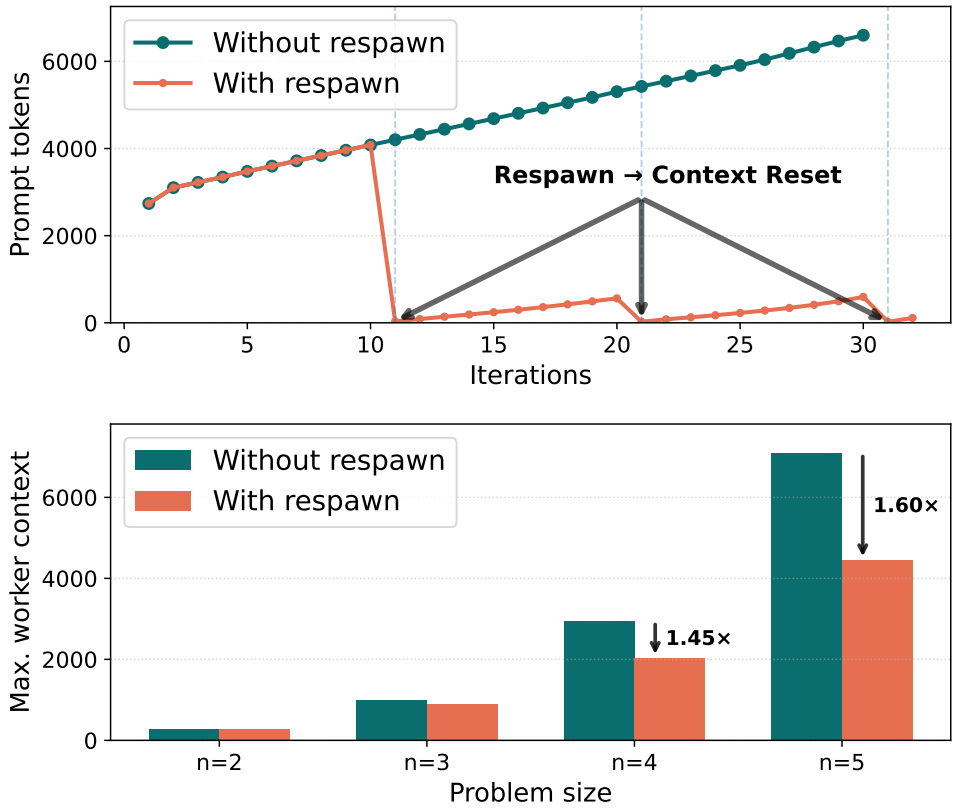

Figure 5: Respawning at a regular frequency (10 iterations in our setting) effectively reduces maximum worker context.

This mechanism has two key effects. First, it prevents unbounded growth of per-worker context: as shown in Figure 5, respawning keeps the maximum worker context substantially lower across problem sizes, with the gap widening as Sudoku size increases. Second, it reduces per-iteration input tokens by eliminating historical reasoning that is no longer necessary, leading to the characteristic 'sawtooth' pattern where input tokens reset after each respawn.

The reduced size of the prompt also has an effect on the latency of inference since per-token cost increases with the size of the prompt. To test this, we train models for $N = 4$ and $N = 5$ without and with respawning every 10 iterations and measure the effects on latency. We observe that respawning yields 11.3% and 36.0% reductions in latency for $N = 4$ and $N = 5$, respectively.

M System Profiling and Message-Token Overhead

We profile MPLM on representative Sudoku and SAT instances to understand where runtime is spent and how much communication contributes to generation cost. Across all problem sizes, the dominant cost is model execution rather than controller logic. For Sudoku, vLLM accounts for 98.2%, 99.1%, and 99.6% of wall-clock time on 9×9, 16×16, and 25×25 instances, respectively, while controller overhead remains below 2% and decreases as the problem size grows. This indicates that the message-passing controller adds negligible overhead relative to LLM inference, even as the number of rounds and requests increases substantially. GPU utilization increases with Sudoku size, rising from 68.9% on 9×9 to 90.2% on 25×25, suggesting that larger MPLM workloads better saturate the inference backend.

We observe a similar pattern on SAT. For both SAT and UNSAT instances, nearly all wall-clock time is spent inside vLLM, with controller overhead at 0.1% or lower. The SAT instances require fewer parallel requests per round than Sudoku, reflecting the smaller number of active reasoning threads in these examples, but still maintain high GPU utilization around 83.5%.

Finally, we measure the composition of generated output tokens on Sudoku to quantify message-token overhead. Master threads produce no chain-of-thought tokens; their outputs consist only of directives and message payloads, as expected from their role as coordinators. Child threads, in contrast, spend most of their output budget on local reasoning for larger puzzles. In particular, message payload tokens account for only 8.0%, 1.6%, and 0.9% of child-thread output on 9×9, 16×16, and 25×25 Sudoku, respectively.

Metric	$n = 3$ (9×9)	$n = 4$ (16×16)	$n = 5$ (25×25)
Total wall-clock time	16.99s	112.898s	999.014s
vLLM execution time	16.68s	111.832s	995.208s
vLLM execution ratio	98.2%	99.1%	99.6%
Controller overhead	0.303s	1.066s	3.806s
Controller overhead ratio	1.8%	0.9%	0.4%
Number of rounds	24	46	102
Total vLLM requests	420	1,682	5,609
Average requests per round	17.5	36.6	55.0

Table 4: Runtime profiling for MPLM on representative Sudoku instances.

Metric	$n = 3$ (9×9)	$n = 4$ (16×16)	$n = 5$ (25×25)
Mean GPU utilization	68.9%	82.5%	90.2%
Maximum GPU utilization	82%	93%	100%
Mean KV-cache usage	0.012%	0.124%	0.443%
Maximum KV-cache usage	0.030%	0.579%	1.000%
Minimum KV-cache usage	0.000%	0.000%	0.000%
Prefix-cache hit rate	75.7%	84.0%	41.7%

Table 5: GPU and KV-cache profiling for MPLM on representative Sudoku instances.

Metric	SAT	UNSAT
Total wall-clock time	41.757s	54.686s
vLLM execution time	41.726s	54.676s
vLLM execution ratio	99.9%	100.0%
Controller overhead	0.031s	0.010s
Controller overhead ratio	0.1%	0.0%
Number of rounds	18	45
Total vLLM requests	45	72
Average requests per round	2.39	1.71

Table 6: Runtime profiling for MPLM on representative SAT instances.

Metric	SAT	UNSAT
Mean GPU utilization	83.6%	83.5%
Maximum GPU utilization	90%	87%
Mean KV-cache usage	0.023%	0.015%
Maximum KV-cache usage	0.056%	0.028%
Minimum KV-cache usage	0.000%	0.000%
Prefix-cache hit rate	50.8%	50.6%

Table 7: GPU and KV-cache profiling for MPLM on representative SAT instances.

Size	Thread Type	CoT Tokens	Directive Tokens	Message Payload Tokens
$n = 3$ (9×9)	Master	0%	53.4%	46.6%
$n = 3$ (9×9)	Child	24.2%	67.8%	8.0%
$n = 4$ (16×16)	Master	0%	50.1%	49.9%
$n = 4$ (16×16)	Child	71.7%	26.7%	1.6%
$n = 5$ (25×25)	Master	0%	48.9%	51.1%
$n = 5$ (25×25)	Child	76.0%	23.1%	0.9%

Table 8: Output token composition for master and child threads on Sudoku.

Thus, while MPLM introduces explicit communication, the token overhead of messages is small for worker threads at larger problem sizes, where most generated tokens are devoted to local reasoning.

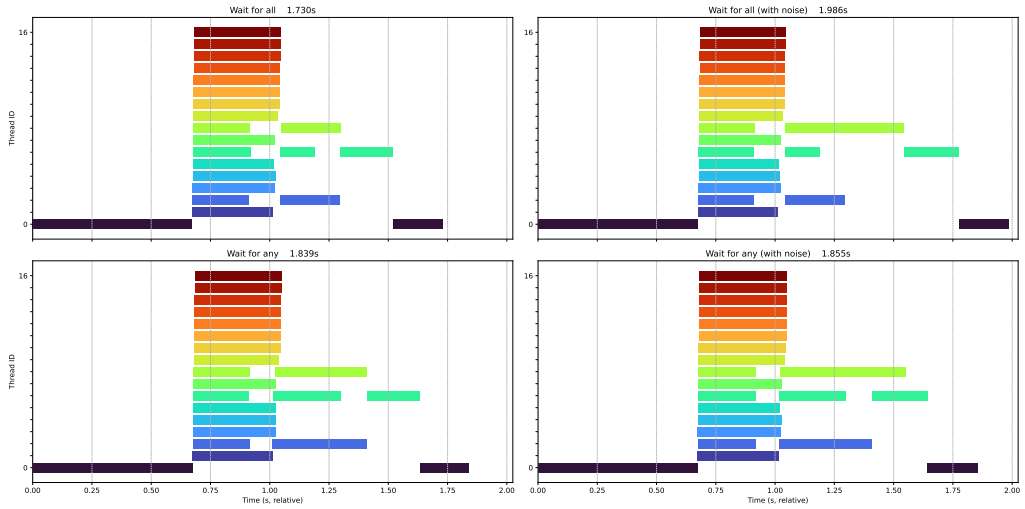

Figure 6: In the example above, we note the latency of individual workers in the settings with added noise (right) v.s. no noise (left). The y-axis denotes the ID of the worker and x-axis denotes time in seconds. The start and end of a bar denotes the time frame for which a worker was active during inference.

N Synchronization Schemes Under Variable Latencies Across Workers

As we discuss in Section B.1, there are multiple ways in which ways in which each thread is allowed to invoke the inference engine and *when* messages are delivered. We discuss two implementations: **wait for all** and **wait for any**. Technically, they can have different primitives but instead we use the same tokens and change their behavior to understand their individual behaviors without having to re-train the model.

On our Sudoku instantiation, each worker executes a highly structured and largely deterministic computation. But in practice, worker latencies can vary substantially across iterations and across threads, primarily due to heterogeneous input and output token lengths and system-level contention and queueing on shared GPU resources such as dynamic batching. To simulate this scenario while keeping the underlying reasoning behavior unchanged, we inject an additional post-run delay into each thread to amplify cross-worker and cross-iteration variance. For worker i at iteration t, we extend its runtime by

$$\Delta_{i,t} \;=\; r_{i,t} \cdot u_{i,t}, \qquad u_{i,t} \sim \mathrm{Uniform}(0,1),$$

where $r_{i,t}$ is the measured thread runtime excluding the injected delay and $u_{i,t}$ is a deterministic pseudo-random draw fixed per (i,t) for reproducibility. This construction increases the latency dispersion within and across iterations without changing the logical content of the messages. We perform an experimental study on the 4×4 Sudoku puzzles where we use the same noise (by fixing the random seed) and use the two synchronization schemes on a dataset of 100 puzzles and then compare the final latencies.

Results: We observe that out of the 100 puzzles, in 76 Sudoku puzzles, **wait-for-any** is faster because it doesn't have to wait for the slowest worker in its neighbors. However, this can be worse in cases where actually waiting for the slowest worker can be beneficial since more information is gained per iteration which would result in fewer iterations to be used for solving the overall puzzle. Below, we discuss an example where overall latency is lower because of the wait-for-all synchronization scheme.

Consider the Sudoku drawn here where there are only 3 cells to be filled denoted by W_a, W_b and W_c. Note that W_a and W_c can be resolved after 2 iterations. However, W_b can only be resolved after either W_a or W_c is resolved. Therefore, W_b is critically dependent on W_a and W_c but not both. Even if one of them finishes, W_b can finish its task. Therefore, consider that W_c is much slower.

4	W_a	1	2
1	W_b	3	W_c
3	4	2	1
2	1	4	3

Figure 7: An example Sudoku where **wait-for-any** can be faster than **wait-for-all** sychronization.

Then, in the **wait-for-all** setting, W_b will have to wait for W_c. Therefore, latency will be increased due to W_c's slowness.

However, in the **wait-for-any** setting, W_b can start when W_a is finished. Therefore, there is a possibility of a speedup.

This result is seen in right part of Figure 6 where **wait-for-any** is actually faster than **wait-for-all**.

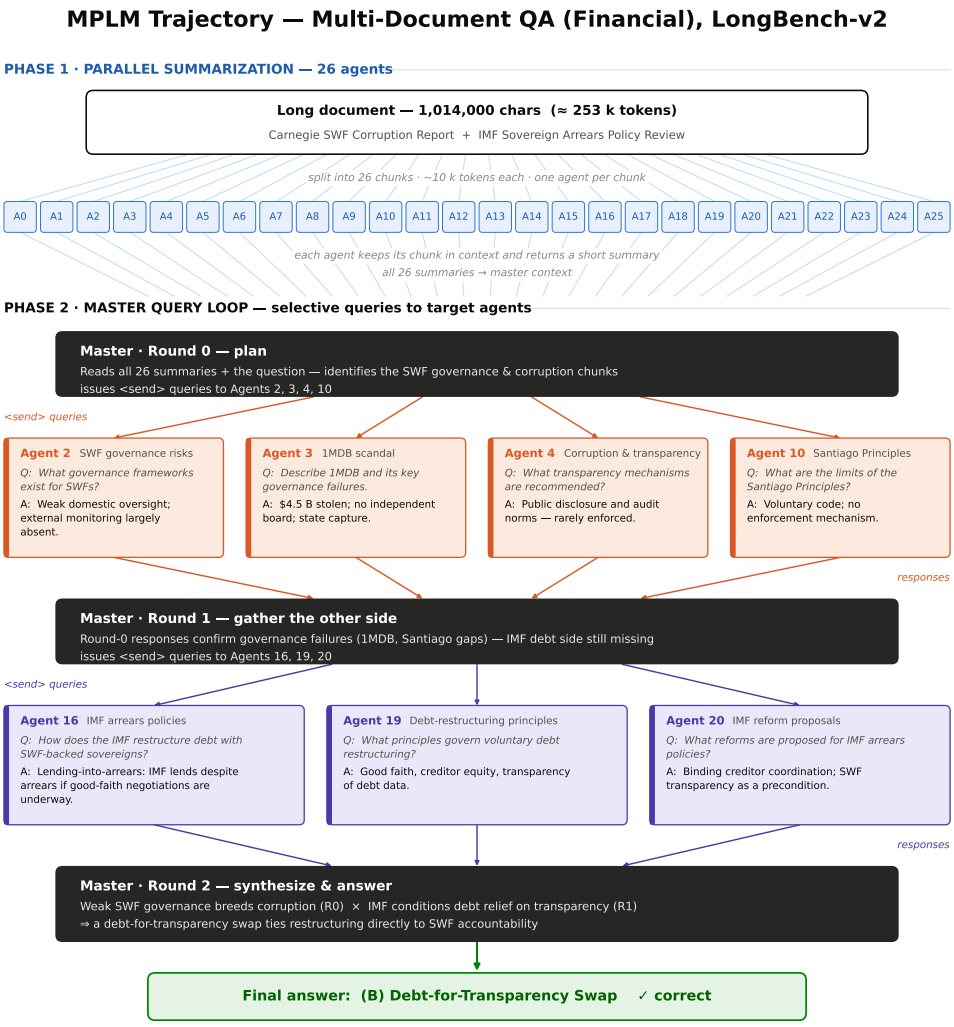

Figure 8: In the example above, we show message passing can be used to selective route information across different parallel threads. This figure is an illustrative example from the LongBench-v2 dataset using the Qwen3-30B-A3B model.

O Long Context Question Answering

MPLMs decompose long-context reasoning into two stages: parallel local processing followed by targeted message routing. Given a long input document or collection of documents, the main thread first partitions the context into chunks and spawns a set of parallel worker threads, each assigned to process one chunk. Each worker reads its local chunk, constructs a compact summary, and remains active as a persistent worker associated with that chunk. The main thread receives these summaries, which provide a global index over the document without requiring the full context to be repeatedly copied into every subsequent reasoning step.

After this initial parallel pass, the main thread performs an iterative query-routing loop. Rather than broadcasting every follow-up question to all workers, the main thread uses the chunk summaries and the current reasoning state to identify which persistent workers are relevant. It then sends targeted messages only to those workers, asking them to recover local evidence, answer subquestions, or verify intermediate claims. Since each worker retains its chunk-specific state, the main thread can route new information back to the same worker without retransmitting the original chunk. This enables the system to combine global coordination with local memory: the main thread decides what information is needed next, while persistent workers provide grounded responses from their assigned portions of the context. We report the full results on LongBench-v2 in Table 9. An example can be seen in Figure 8

Partition	Qwen3-30B-A3B				Qwen3.6-35B-A3B			
	MPLM Accuracy	MPLM Latency	RLM Accuracy	RLM Latency	MPLM Accuracy	MPLM Latency	RLM Accuracy	RLM Latency
I. Single-Document QA								
Academic	**39.8%**	**52.4s**	37.5%	110.7s	**47.7%**	**88.9s**	**47.7%**	163.0s
Literary	**40.8%**	**55.6s**	32.5%	108.2s	**40.0%**	**112.2s**	**40.0%**	210.7s
Legal	**36.8%**	**46.3s**	34.2%	89.8s	47.4%	85.9s	**68.4%**	**84.3s**
Financial	**63.6%**	**57.9s**	38.6%	141.3s	63.6%	**120.8s**	**68.2%**	232.3s
Governmental	**33.3%**	**57.3s**	**33.3%**	119.2s	**38.9%**	**118.9s**	**38.9%**	248.7s
Detective	13.6%	**62.9s**	**46.6%**	130.0s	27.3%	**122.2s**	**36.4%**	279.5s
Event Ordering	**35.0%**	**82.3s**	27.5%	129.2s	**75.0%**	**123.9s**	35.0%	497.6s
II. Multi-Document QA								
Academic	**36.0%**	**60.0s**	17.5%	107.1s	**48.0%**	**100.7s**	42.0%	259.6s
Legal	**30.4%**	**61.5s**	21.4%	75.8s	**50.0%**	**95.9s**	21.4%	180.4s
Financial	**48.3%**	**64.1s**	28.3%	114.6s	**53.3%**	**123.1s**	33.3%	279.4s
Governmental	**35.9%**	**57.0s**	13.0%	103.3s	**34.8%**	**121.0s**	**34.8%**	283.5s
Multi-News	26.1%	**53.0s**	**34.8%**	112.5s	52.2%	**88.0s**	**56.5%**	139.6s
III. Long In-context Learning								
User Guide QA	**34.4%**	**59.2s**	25.0%	105.4s	**55.0%**	**96.1s**	40.0%	258.7s
New Language Translation	**27.5%**	**78.3s**	22.5%	104.9s	**55.0%**	**115.6s**	50.0%	332.9s
Many-Shot Learning	**42.9%**	**76.1s**	33.3%	79.2s	**33.3%**	**162.2s**	23.8%	453.6s
IV. Long-dialogue History Understanding								
Agent History QA	**40.0%**	**74.4s**	28.7%	98.0s	**75.0%**	90.0s	60.0%	**70.6s**
Dialogue History QA	**63.2%**	**45.0s**	35.5%	87.0s	47.4%	98.4s	**57.9%**	**75.0s**
V. Code Repository Understanding								
Code Repo QA	**35.0%**	**65.5s**	32.5%	93.3s	28.0%	**56.7s**	**52.0%**	187.8s
VI. Long Structured Data Understanding								
Table QA	37.5%	**66.7s**	**41.7%**	99.0s	27.8%	**95.9s**	**50.0%**	96.9s
Knowledge Graph Reasoning	**48.3%**	**61.2s**	11.7%	100.8s	53.3%	**112.6s**	**86.7%**	112.9s
Average	**37.8%**	**61.3s**	29.7%	105.7s	46.5%	**102.2s**	**46.7%**	223.5s

Table 9: Full LongBench-v2 results. Qwen3 evaluations were performed on a single node of 4 GH200 GPUs and the Qwen3.6 evaluations were done on a single node of 4 L40 GPUs.