EXPRESSING AND EXPLOITING PARALLELISM IN LANGUAGE MODEL DECODING

Tian Jin[*1] **Ellie Y. Cheng**[*1] **Zack Ankner**[1,2] **Jonathan Ragan-Kelley**[1] **Michael Carbin**[1]

[1]MIT CSAIL [2]MosaicML

ABSTRACT

For autoregressive language models, decoding naturally occurs sequentially, generating tokens one after another. Recent attempts to introduce parallelism require a pre-determined structure to implement parallel generation, such as generating an outline and dividing the responses into parallel sub-tasks. In this work, we explore a new technique to automate parallel generation by dynamically exploiting various parallel structures in the semantics of the language model response based on *approximate conditional independence* in generation probabilities. Specifically, we introduce a simple annotation language PASTA-LANG that allows language models to express approximate conditional independence structures in their outputs. We then develop an interpreter for PASTA-LANG that performs on-the-fly parallel generation during decoding, exploiting the parallelism expressed in the PASTA-LANG-annotated outputs. We demonstrate that our approach can improve tokens generated per second by 66-83% with high-quality generation.

1 INTRODUCTION

Recently, large language models (LLMs) rapidly grew in popularity and application, solving a wide range of problems from program synthesis to chatbot assistants. The autoregressive nature of these language models, however, means that the model must generate its outputs sequentially, creating a performance bottleneck in inference speed. This limitation restricts the practical applications of LLMs, especially in latency-sensitive applications.

Several recent efforts exploited the local statistical association between tokens to speed up autoregressive decoding by generating a few tokens at a time, such as speculative decoding (Leviathan et al., 2023; Liu et al., 2023), and lookahead decoding (Fu et al., 2023). However, they are limited to predicting a few tokens per step to prevent quality loss. Another line of work, such as Skeleton-of-Thought (SoT) (Ning et al., 2023) and APAR (Liu et al., 2024) exploited the syntactic structure of the response to generate tokens in parallel. For instance, SoT first generates a skeleton of the response as a list of points and then generates the descriptions of each point in parallel.

In this work, we present a new technique of parallel decoding grounded in the approximate conditional independence structure of the language model's generation probabilities.

Conditional Independence. We observe that many language model generations contain semantically independent structures: that is, the generation of certain groups of tokens are semantically independent from each other, when conditioned on the prefix of these groups of tokens. For example, in the response in Figure 1, the generation of "flammable" after "Newspaper is" should be the same with and without the tokens in the sentence immediately following "Coal and wood are", since the flammability of newspaper is not dependent on the flammability of coal and wood. Therefore, given the prefixes "Coal and wood are" and "Newspaper is", we can decode the tokens immediately following each prefix at the same time in parallel.

Moreover, we empirically confirm that the semantic conditional independence structure manifests as approximate conditional independence in the generation probabilities of the language model. Our

[*]Authors contribute equally. Correspondence to: tianjin@csail.mit.edu, ellieyhc@csail.mit.edu, mcarbin@csail.mit.edu..

User: Classify each of the following as flammable or non-flammable: coal, wood, newspaper, water, glass

Assistant: <scope>
Coal and wood are <async> flammable materials. They can easily ignite and burn when exposed to a source of heat or a spark. </async>
Newspaper is <async> flammable. It is made of paper, which is highly combustible. </async>
Water is <async> a non-flammable substance. It is used to extinguish fires because it removes the heat required for a fire to continue burning. </async>
Glass is <async> a non-flammable material. It does not burn or support combustion. </async>
</scope>

Figure 1: Example response using PASTA-LANG to indicate content that have parallel structure. Each `<async>` tag indicates content that can be generated in parallel with respect to another `<async>` tag. The `<scope>` tag marks synchronization and the interpreter waits for the `<async>` tags to complete generation.

findings provide a unifying framework for identifying and verifying asynchronous parallel decoding opportunities in language model generations, grounding and subsuming prior syntactically-driven parallel decoding techniques (Ning et al., 2023; Liu et al., 2024).

PASTA-LANG: We introduce an annotation language, PASTA-LANG – PArallel STructure Annotation Language, to express said conditional independence structure in the language model's generation probability in the model's output itself. We then train the language model to annotate conditional independence structure in its own outputs. In Figure 1, we present a PASTA-LANG-annotated response from Mistral 7B Instruct v0.2, that exploits parallel structure in the responses that cannot be exploited by SoT and APAR. The parallel sub-tasks of classifying each listed item are denoted by `<async>` tags, which mark contents that are logically independent of other `<async>`-wrapped contents. The `<scope>` tags denote where the independence applies. In this example, the conditional independence applies to the entire response. Using our simple annotation language, models can automatically express arbitrary parallelism during inference.

Training dataset. We create a dataset of PASTA-LANG-annotated responses to fine-tune language models to use PASTA-LANG based on our factorization framework. Our approach automates parallel generation over a diverse range of structures based on responses to those prompts. These structures include simple lists with descriptions, as well as more complicated structures including, but not limited to, nested lists, patterns with sequential text interspersed with parallel structures, and interrupting phrases and interjections.

Interpreter. Lastly, we developed an interpreter for PASTA-LANG that implements parallel generation dynamically during language model decoding. The interpreter eagerly processes tokens as they are decoded from the model, initiating asynchronous generation on-the-fly. The interpreter is general; with no restrictions on the decoded response, as long as it is valid PASTA-LANG-annotated text, the interpreter can implement any given parallel structure.

In summary, our contributions are as follows:

- PASTA-LANG, an XML-like annotation language for expressing arbitrary parallelism in language model decoding.
- Unifying framework for identifying parallel structure in generations, grounded in sound factorization of generation probabilities.
- A dataset for fine-tuning LLMs to generate PASTA-LANG-annotated outputs.
- Evaluation demonstrating our methods increases the tokens generated per second by 66-83% during inference with high-quality generation.
- An interpreter for PASTA-LANG that performs on-the-fly parallel generation.

2 LANGUAGE DESIGN

PASTA-LANG is an XML-like language designed to specify arbitrary parallelism in language model decoding. We present the syntax and semantics of the language in this section.

Syntax. PASTA-LANG introduces two kinds of tags: `<scope>` and `<async>`. They are container tags, which use both an opening and a closing tag to enclose content. These can nest arbitrarily and may have multiple sibling elements within the same level. PASTA-LANG imposes one additional syntactic constraint: every `<async>` element must be a descendant of a `<scope>` element.

Semantics. The tags `<async>` and `</async>` designate sequences of tokens that the language model can decode asynchronously. During inference, a PASTA-LANG interpreter simultaneously samples tokens for content within the `<async>` tags and beyond the `</async>` tags. In other words, decoding jumps past the `</async>` tag, generating tokens without waiting for the tokens within the `<async>` tags to be decoded.

Without synchronization, token generations outside of a pair of `<async>` tags may never be conditioned on the asynchronously generated tokens generated within it. To allow decoding to depend on the asynchronously generated content, we introduce the `<scope>` tag to mark points where asynchronous generations must synchronize. Upon sampling a `</scope>` token, the PASTA-LANG interpreter waits for asynchronously generated contents to complete, ensuring subsequent decodings can condition on them. The `<scope>` and `</scope>` tags mark the beginning and end of a *parallel scope*. The interpreter only waits for asynchronous generations initiated within the parallel scope corresponding to the `<scope>` tag to complete.

Expressivity. We design PASTA-LANG to include only simple synchronization primitives. We avoid more complex, fine-grained synchronization primitives such as producer-consumer synchronization to make the language easy to use by LLMs. PASTA-LANG is already expressive enough to capture a wide range of parallel structures in text generation. It supports the arbitrary nesting of parallel scopes and the initiation of asynchronous token generation within these scopes. This capability enables complex parallel structures, where multiple levels of parallelism can be managed and synchronized at different points in the decoding process.

Approximate Conditional Independence. PASTA-LANG encodes an approximation of the language model's generation probabilities. Given a sequence of tokens T that may include PASTA-LANG annotations, asynchronously sampled from a language model, we define the following terms:

1. We refer to the prefix of a token t that has been sequentially generated before t as its *synchronous prefix*, denoted as $S(t)$.
2. During asynchronous decoding, the generation probability of a token t can depend only on the given token's synchronous prefix: $P(t|S(t))$. In contrast, during sequential decoding, the generation probability of a token depends on the entire prefix: $P(t|\Pi(t))$ where $\Pi(t)$ refers to the entire prefix of t.
3. We use $\delta_{T,P}$ to refer to the approximation error of an asynchronously sampled language model $P(.)$ with respect to synchronously sampling from the same model $P(.)$ for each token in T. One example of an approximation error function is the 99th percentile of per-token generation probability decrease over the set of tokens: $\mathrm{P}_{99}[P(t|\Pi(t)) - P(t|S(t))]_{t\in T}$.

We proceed to define the approximation of the language model's generation probabilities encoded by PASTA-LANG tags. For simplicity, we do not consider the case where a PASTA-LANG tag has empty content or where a PASTA-LANG tag immediately follows another tag. And we assume PASTA-LANG annotations to always be syntactically valid.

To begin with, we inductively define the synchronous prefix $S(t_i)$ of a token t_i. For the base case of the first token of the generation, we define $S(t_0) = \emptyset$. There are four inductive cases of $S(t_i)$: 1). the previous token of t_i is not a PASTA-LANG tag, 2). the previous token of t_i is `<async>` or `<scope>`, 3). the previous token of t_i is `</async>`, and 4). the previous token of t_i is `</scope>`. For the first case, $S(t_i)$ is simply the synchronous prefix of the previous token plus the previous token itself: $S(t_i) = S(t_{i-1}) + t_{i-1}$.

In the second case, the previous token is either `<scope>` or `<async>`. The synchronous prefix is defined as the synchronous prefix of the token before the PASTA-LANG tag plus the token itself. For example, consider the sequence,

... t_a <scope> t_i ...

the synchronous prefix of t_i is the synchronous prefix of the last token before the <scope> tag t_a plus t_a itself: $S(t_i) = S(t_a) + t_a$.

In the third case, the previous token is </async>. There must be a matching <async> in the prefix:

... t_a <async> t_b ... t_c </async> t_i ...

The synchronous prefix of the token t_i is the synchronous prefix of the final token before the matching <async> tag t_a plus t_a itself: $S(t_i) = S(t_a) + t_a$. Asynchronous decoding computes the generation probability of token t_i from only its synchronous prefix $S(t_i)$, not the entire prefix. [1]

In the fourth case, the previous token is </scope>. There must exists a matching <scope> tag:

... t_a <scope> T_{scope} </scope> t_i...

The synchronous prefix of the token t_i is the synchronous prefix of the last token before the <scope> tag t_a plus t_a itself and every single non-PASTA-LANG token generated within the <scope> tag, synchronously or asynchronously: $S(t_i) = S(t_a) + t_a + \textit{remove\_pasta}(T_{\text{scope}})$.

Altogether, PASTA-LANG-annotations encodes an approximation of the generation probability of a token t_i conditioned on its full prefix $P(t_i|\Pi(t_i))$ using only its synchronous prefix $P(t_i|S(t_i))$. This approximation is only sound if the generation probability of t_i conditioned on the full prefix is approximately equal to the generation probability of t_i conditioned on the synchronous prefix, that is $\delta_{T,P} \leq \epsilon$ for some small ϵ. If the approximation is sound, this implies that conditioned on the synchronous prefix, the generation probability of t_i is approximately independent of the tokens generated asynchronously: $\Pi(t_i) \setminus S(t_i)$.

3 IDENTIFYING APPROXIMATE CONDITIONAL INDEPENDENCE

We design PASTA-LANG to encode approximate conditional independence structures within the generation probabilities of language models. Do these structures actually arise in practice? In this section, we use a naive combinatorial search algorithm to find approximate conditional independence structures in real language model responses.

Specifically, for the following example, we tried all valid combinations of PASTA-LANG annotations to identify the approximate conditional independence structure in the generation probabilities of the Mistral 7B Instruct v0.2 model, evaluated on tokens in this example. We measure the approximation error of the asynchronous generation probabilities by comparing it with the synchronous generation probabilities and taking the 99th percentile of the decrease in the per-token generation probability over the full set of tokens in the example. Among annotations that achieve a 99th percentile approximation error of less than 5%, pick the one that has fewest number of tokens on its critical path. This particular example achieves a 99th percentile approximation error of 2.5%.

<scope> Workers are the smallest of the bee castes, but are by far the most numerous. All workers are female, and normally incapable of reproduction. They are unable to mate, but in a hopelessly queenless colony, workers may begin to lay unfertilized eggs, which develop into drones. Drones are the male honey bees. The only function of a drone is to fertilize a young queen bee. Queen bee function in the hive is one of production. Queen bee is normally the only reproductive female in the colony. Egg <async> -laying begins in early spring, initiated when the first fresh pollen is brought home by the workers. Egg production will continue until fall, or as long as pollen is available. </async> So there are three types of bees: workers, drones and a queen bee. </scope>

The question posed is, "What is the difference between worker bees and drone bees, and are there any other types of bees?". Notably, this response is very challenging to asynchronously generate: most of the sentence is logically tightly interconnected: Discussing worker bees naturally leads to mentioning drone bees, whose primary role is to fertilize the young queen bee. However, the

[1] Even though t_b and t_i share the same synchronous prefix: (i.e. $S(t_i) = S(t_b)$), in practice, the language model can still decode t_b differently from t_i, because the positional encodings for t_i and t_b are not the same.

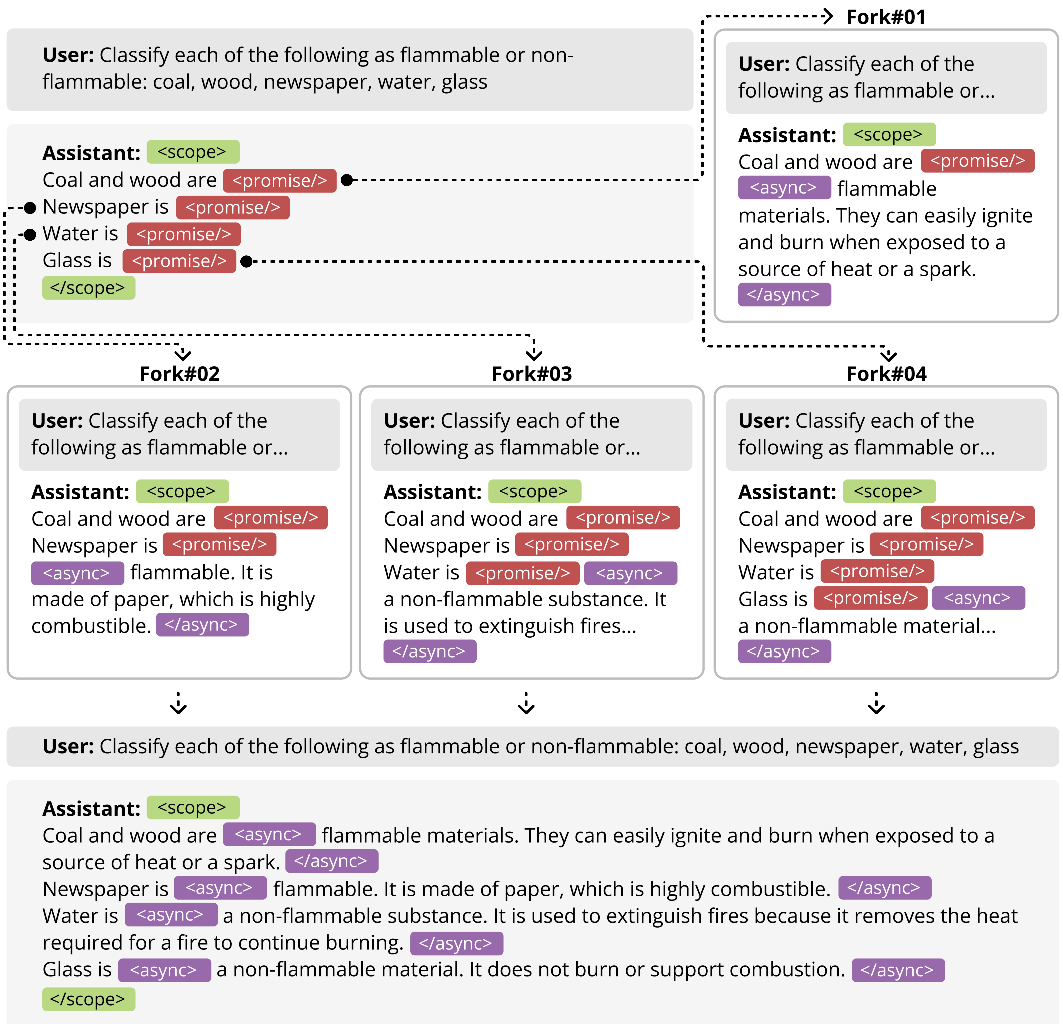


Figure 2: The interpreter invokes the INTERPRET_SCOPE operation upon decoding the `<scope>` tag. The operation continues decoding until the first `<promise/>` tag in the first line, upon which it creates Fork#01. In Fork#01, the interpreter adds an `<async>` tag and continues decoding the classification of coal and wood asynchronously until the `</async>` tag. The main thread continues decoding until the second `<promise/>`. It creates Fork#02, which contains the `<promise/>` tag of the first line in its prefix, and decodes the classification of newspaper. The main thread continues decoding in parallel to Fork#01 and Fork#02 until the next `<promise/>` tag, and so on. It stops after generating `</scope>`, where it waits until all forks are finished, and composes the entire response together by replacing `<promise/>` tags with their respective asynchronously generated tokens.

specifics about egg-laying stand out as a separate detail that could be developed independently without affecting the subsequent summary of the three bee types. Approximate conditional independence structures occur in real-world dataset and are exploitable opportunities for parallel generation.

4 Interpreter Design

Given a language model that outputs PASTA-LANG-annotated generations, the decoding algorithm can utilize the PASTA-LANG-annotations to implement parallel generation. We design a PASTA-LANG interpreter that will automatically decode PASTA-LANG-annotated outputs. The interpreter parses `<scope>` tags to initiate a parallel scope and keep track of the asynchronous generations initiated within its bounds. Upon reaching the `</scope>` tag, the interpreter waits for all asynchronous generations initiated within this parallel scope to complete and retrieves the asynchronously generated content to insert back into the appropriate location in the scope.

To implement asynchronous generation, we introduce a third PASTA-LANG tag: the `<promise/>` tag. It is a self-closing tag, and it serves as a placeholder for asynchronously generated content that has not been synchronized yet. Thus, PASTA-LANG-equipped language model will output `<promise/>` to signify where asynchronously generated contents are located.

We implemented an interpreter for PASTA-LANG using vLLM (Kwon et al., 2023) to serve model generations. We present the pseudocode for the PASTA-LANG interpreter in Appendix A. The interpreter is the operation INTERPRET_SCOPE that decodes the language model and interprets the PASTA-LANG-annotated output as tokens are generated. The interpreter is invoked during standard decoding when a `<scope>` tag is encountered in the language model's output. The interpreter execution terminates when it encounters the matching `</scope>` tag.

The interpreter continuously decodes the language model, constrained to generate syntactically valid PASTA-LANG tag. For each generated token, the interpreter matches the token against PASTA-LANG tags. If the token is a `<scope>` tag, the interpreter recursively calls the INTERPRET_SCOPE operation to interpret the nested scope. If the token is a `<promise/>` tag, the interpreter initiates asynchronous generation by appending `<async>` to the prefix and decoding a sequence of tokens that end with `</async>` in a separate fork. The interpreter tracks the forks of every asynchronous generation initiated within the parallel scope. In the main thread, the interpreter will continue generating the rest of the output within the parallel scope, with the `<promise/>` tag in the prefix representing the asynchronously generated content. When the interpreter encounters a `</scope>` tag, it synchronizes all asynchronous generations initiated within the parallel scope. It waits for all forks to finish, and then it replaces each `<promise/>` tag with its corresponding asynchronously generated content. Figure 2 shows how the interpreter produces the output from Figure 1.

5 METHOD

We teach LLMs how to use the PASTA-LANG annotation Language through fine-tuning. Here, we discuss our methodology for fine-tuning on PASTA-LANG-annotated outputs.

5.1 DATASET CURATION

Even with the approximate conditional independence defined in Section 2, it is intractable to search for sound annotations within a response. Instead, we use heuristics to annotate responses and filter for those with high quality. We create the PASTA-LANG finetuning dataset with a three-step process. We sample responses from the base language model, annotate the responses with PASTA-LANG tags, and filter the responses based on our quality and efficiency criteria.

Sampling. Typically, the instruction-finetuning data for state-of-the-art open-weight models are proprietary and not available for public use. Therefore, to minimally affect the model's performance, we sample the training data from the model itself using prompts from an open-source instruction-finetuning dataset: specifically, we greedily sample the Mistral 7B Instruct v0.2 (Jiang et al., 2023) model using prompts from the open source Dolly dataset (Conover et al., 2023).

Annotation. For responses with a list structure, we annotate the entire list with `<scope>`, and nested child of each list item with `<async>`. In an arbitrarily nested list, sibling items are not semantically independent, as the presence of one item may influence subsequent ones. However, the nested children of each item are usually semantically independent from the rest of the list. We consider descriptions or further elaborations on the item as nested children of the list item.

For examples without list structures, we use GPT4 to generate the PASTA-LANG-annotated responses. In the prompt, we explain the syntax and semantics of PASTA-LANG and ask GPT4 to rewrite a given Mistral-7B response in PASTA-LANG without changing the content of the response.

Filtering. Neither heuristic applied is inherently sound. In this final step, we evaluate the quality of PASTA-LANG-annotated responses from the previous step by comparing the quality and efficiency of predictions made by the Mistral 7B model with and without the PASTA-LANG-approximation.

To accept a PASTA-LANG-annotated response, it must meet two conditions: a) the number of next token predictions that differ from those made by the Mistral 7B model without PASTA-LANG-approximation is less than 20% of the total tokens in the response, and b) the predicted latency of the response is at least 20% faster compared to the Mistral model without PASTA-LANG-factorization.

In total, we collected 21,869 annotated responses, which is 60.8% of all the heuristic-based annotations we collected. We maintain a 1:1 ratio between responses with and without list structures.

5.2 TRAINING TECHNIQUE

We make the following adjustments to the standard training procedure to ensure that the model correctly learns to generate PASTA-LANG-annotated responses during inference.

Firstly, if we simply finetune the model on the PASTA-LANG-annotated responses, during training, the model will attend to asynchronously generated content that is not available during inference. We therefore adjust the attention mask to restrict each token's attention to its synchronous prefix. Secondly, we empirically find it necessary to adjust the positional ids of each token to be their corresponding values during inference. In other words, we configure the positional ids of each token to the number of tokens in its synchronous prefix. Thirdly, since the interpreter is responsible for inserting the `<async>` tag, we disable loss on the `<async>` tag during training. Finally, it is customary during instruction-finetuning to impose a KL divergence loss between the original model's output and the finetuned model's output to ensure that the finetuned model does not deviate significantly from the original model's output (Rafailov et al., 2023). We also adopt this technique.

5.3 HYPERPARAMETERS

We fine-tune Mistral 7B Instruct v0.2 with LoRA(Hu et al., 2022) using a LoRA rank and alpha of 16. We only fine-tune the K/Q/V/O projection matrices of the self-attention layers. We use a batch size of 8, a linearly decaying learning rate starting at 1e-4, and a warmup of 10 steps. We fine-tune for 3 epoch. The full fine-tuning run takes approximately 7 hours on a A100 GPU. We mulitiply the KL divergence loss by 0.2 and add it to the causal language modeling loss. We refer to this fine-tuned model as Mistral-PASTA.

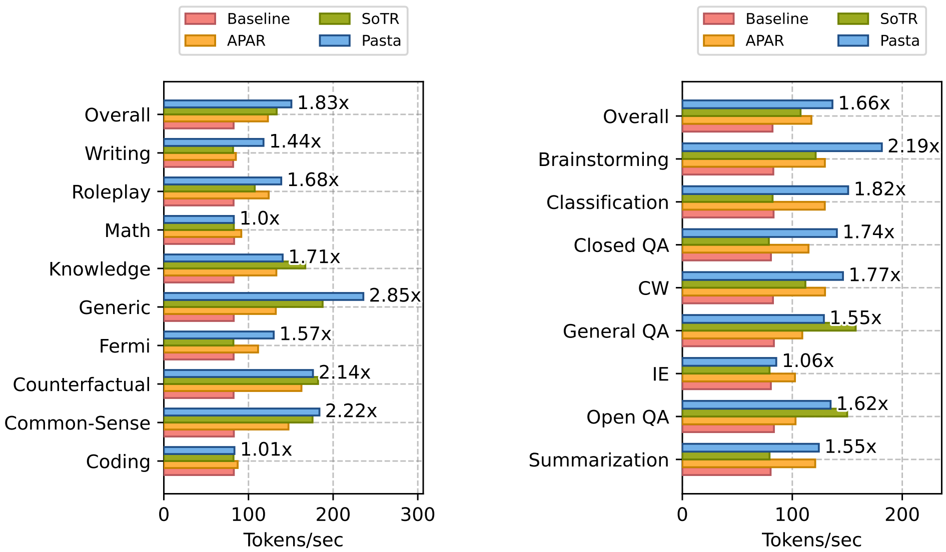
6 EVALUATION

This section evaluates the quality and efficiency of language model decoding using PASTA-LANG. We conduct all our evaluations on an NVIDIA A100 GPU with 80GB of memory.

Models and Parallel Generation Techniques. We evaluate using four different execution strategies: a). the baseline Mistral model, executed with vLLM (Kwon et al., 2023); b). Mistral-SoTR, using the SoT with router prompting strategy (Ning et al., 2023) on the baseline Mistral model; c). Mistral-APAR, the Mistral model finetuned with APAR to use APAR decoding (Liu et al., 2024); and d). Mistral-PASTA, the Mistral model fine-tuned to generate PASTA-LANG-annotated responses, which we execute using PASTA-LANG interpreter. SoT and APAR are both parallel generation techniques for increasing inference efficiency by targeting specific predefined parallel structures to exploit. To standardize the evaluation, we finetune the Mistral-APAR model using the same dataset as Mistral-PASTA, but with the APAR data pre-preprocessing instead.

Method. We evaluate each technique on Vicuna-80 (Chiang et al., 2023) and *Dolly Bench*. Dolly Bench is a set of 80 questions sampled from the Dolly dataset, 10 from each category, that are not in the training set. We use greedy decoding to generate responses. For each question and model, we obtain K completions and compute the average runtime to measure the time required for full responses accurately. We use $K = 3$.

LLM-as-a-judge. To evaluate the quality of the responses from each model we use LLM Judge (Zheng et al., 2023a) to grade the responses. The judge provides GPT-4 with the instruction and response and asks it to score the response from 1 to 10. We obtain Q sets of scores to account for stochasticity in the GPT-4 outputs. We report the mean and standard deviation of the average quality scores overall and for each category. We use $Q = 3$.



(a) Vicuna-80 (b) Dolly Bench

Figure 3: Generation speed of each method. We do not count PASTA-LANG-tags or APAR control tokens towards throughput.

Usage results. For each parallel generation method, we counted the number of questions in each benchmark for which the method was used. Mistral-SoTR uses a router model to decide whether the SoT prompting should be applied; Mistral-APAR and Mistral-PASTA only use parallel generation when the model outputs APAR control tokens or PASTA-LANG's `<async>` token, respectively. These methods fall back on sequential autoregressive decoding. Table 1 shows Mistral-PASTA is the most versatile technique, applying to more questions than APAR and SoTR.

Method	Vicuna-80	Dolly Bench
APAR	54/80	50/80
SoTR	38/80	26/80
PASTA	**61/80**	**72/80**

Table 1: Number of questions that each method applies to.

Efficiency results. We report the efficiency of decoding output by the number of tokens generated per second, excluding tokens corresponding to PASTA-LANG tags and APAR control tokens. We plot the throughput by category for each model in Figure 3. As shown, Mistral-PASTA achieves overall 1.83x the throughput of the baseline Mistral model on Vicuna-80, and 1.66x times on Dolly Bench. Mistral-PASTA achieves as high as 2.85x the throughput of the baseline. Mistral-PASTA also outperforms Mistral-APAR and Mistral-SoTR overall on Vicuna-80 and Dolly Bench.

Quality results. We also report the quality of decoding output by scoring the responses generated by each technique using LLM Judge. We present the generation quality scores for each method in Table 2 and present the breakdown by category in Appendix B.1. To account for possible quality change due to fine-tuning, we fine-tune the baseline Mistral model on the same dataset without PASTA-annotations and evaluate the quality of this model (called Mistral-Finetuned). Since the Mistral-PASTA is trained with a KL divergence loss, which can affect the generation quality, we further fine-tune the Mistral model on the dataset using KL divergence loss as well (referred to as Mistral-Fintuned-KL).

Method	Vicuna-80	Dolly Bench
Baseline	7.88 ± 0.02	$\mathbf{7.80 \pm 0.03}$
Finetuned	$\mathbf{8.02 \pm 0.04}$	7.68 ± 0.04
Finetuned-KL	7.87 ± 0.03	7.70 ± 0.09
APAR	6.21 ± 0.03	6.73 ± 0.05
SoTR	7.48 ± 0.03	7.38 ± 0.09
PASTA (Ours)	7.55 ± 0.05	7.38 ± 0.05

Table 2: Generation quality, judged by GPT-4.

All parallel generation techniques induce quality degradation from the baseline Mistral model, but Mistral-PASTA produces the least decrease in quality. The Mistral-Finetuned model increases in quality on Vicuna-80, but decreases on Dolly Bench. However, the Mistral-Finetuned-KL model does not exhibit a quality boost. As Mistral-PASTA uses the same training methodology, it is expected that it also does not produce a quality boost. Both Mistral-APAR and Mistral-PASTA are fine-tuned on the same training dataset as Mistral-Finetune and Mistral-Finetune-KL, meaning that the parallel generation techniques do induce a slight degradation to generation quality to produce speedup. Mistral-SoTR requires no fine-tuning on the base Mistral model but its quality still decreases. Mistral-PASTA maintains the quality of the baseline Model more than other parallel generation techniques.

Additional Evaluation. We further compare against fine-tuning Mistral with APAR pre-processing on the ShareGPT dataset [2] in Appendix B.2. We also evaluate prompting the baseline Mistral model to use PASTA-LANG instead of fine-tuning in Appendix B.3

Conclusion. We fine-tuned a Mistral model to generate PASTA-LANG-annotated responses and found that, using our interpreter, it generates 66-83% more tokens per second than the baseline Mistral model while maintaining generation quality better than other parallel generation techniques.

7 RELATED WORK

Efficient Decoding. Prior works have presented approaches to optimize decoding to reduce inference latency. Works on speculative decoding (Fu et al., 2023; Liu et al., 2023; Leviathan et al., 2023; Santilli et al., 2023) mitigate the one-at-a-time token generation bottle-neck from autoregressive by guessing a few future tokens during decoding. Our work extends speculative decoding beyond the local level, where we can begin generating future tokens much later in the response by exploiting semantic independence. Another similar line of work on parallel generation (Ning et al., 2023; Zheng et al., 2023b; Liu et al., 2024) also reduces inference latency by generating multiple parts of the response in parallel. These works rely on a pre-determined hierarchy of queries during inference to sketch out the top-level ideas, performing additional queries in parallel to fill out the rest of the response. Our work is not restricted to a predefined structure during inference. The fine-tuned model is allowed to deviate from the examples to apply parallelism to new structures, to arbitrarily nest parallel structures, and to intersperse parallel and sequential structures.

Agent Planning/Tool Use. Our work is related to the idea of agent planning and tool use (Yao et al., 2023; Schick et al., 2023; Shen et al., 2023; Liang et al., 2023; Lu et al., 2023). Prior studies show that LLM-based agents can solve complex tasks by planning and using tools ranging from web search to external APIs. Our work extends the repertoire of tools available to LLMs by developing the PASTA-LANG language and interpreter for improving its own decoding efficiency.

Approximate Parallelization. Our work extends the idea of approximate parallelization (Udupa et al., 2011; Misailovic et al., 2012). Udupa et al. (2011) proposed a framework that allows programmers to annotate breakable data dependencies in a program and developed a compiler and runtime that exploits these annotations to automatically parallelize otherwise sequential regions of code. Misailovic et al. (2012) opportunistically relaxes synchronization primitives in a parallel program to improve parallelism, to program outputs that are acceptably close to the original one. Similarly, we break the sequential decoding process of LLMs into approximately parallelizable and independent components and exploit the parallelism to improve decoding efficiency. Our work differs in that instead of relying on end-user annotation or compiler analysis, we teach LLMs to autonomously express parallelism in their own decoding process using the PASTA-LANG annotation language.

8 CONCLUSION

In this work, we present PASTA-LANG, an XML-like annotation language for expressions parallelism, and an accompanying interpreter that decodes and executes PASTA-LANG-annotated LLM responses. By interpreting the tokens and their corresponding PASTA-LANG annotations at inference time, the interpreter enables parallelized LLM decoding. Our method improves tokens generated per second by 66-83% with high-quality generation. For future work, additional annotation constructs may be useful to further improve decoding performance.

[2] https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered

9 ACKNOWLEDGEMENTS

We thank MosaicML, Google TPU Research Cloud for their generous support to provide the computational resources for this work. We also thank the anonymous reviewers for their valuable feedback.

REFERENCES

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/.

Mike Conover, Matt Hayes, Ankit Mathur, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. Free dolly: Introducing the world's first truly open instruction-tuned llm, 2023. URL https://www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm.

Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Breaking the sequential dependency of llm inference using lookahead decoding, November 2023. URL https://lmsys.org/blog/2023-11-21-lookahead-decoding/.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

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.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, 2023.

Yaobo Liang, Chenfei Wu, Ting Song, Wenshan Wu, Yan Xia, Yu Liu, Yang Ou, Shuai Lu, Lei Ji, Shaoguang Mao, Yun Wang, Linjun Shou, Ming Gong, and Nan Duan. Taskmatrix.ai: Completing tasks by connecting foundation models with millions of apis, 2023.

Mingdao Liu, Aohan Zeng, Bowen Wang, Peng Zhang, Jie Tang, and Yuxiao Dong. Apar: Llms can do auto-parallel auto-regressive decoding. *arXiv preprint arXiv:2401.06761*, 2024.

Xiaoxuan Liu, Lanxiang Hu, Peter Bailis, Ion Stoica, Zhijie Deng, Alvin Cheung, and Hao Zhang. Online speculative decoding. *arXiv preprint arXiv:2310.07177*, 2023.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. Chameleon: Plug-and-play compositional reasoning with large language models, 2023.

Sasa Misailovic, Stelios Sidiroglou, and Martin C. Rinard. Dancing with uncertainty. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, pp. 51–60, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316323. doi: 10.1145/2414729.2414738. URL https://doi.org/10.1145/2414729.2414738.

Xuefei Ning, Zinan Lin, Zixuan Zhou, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Large language models can do parallel decoding. *arXiv preprint arXiv:2307.15337*, 2023.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL `https://arxiv.org/abs/2305.18290`.

Andrea Santilli, Silvio Severino, Emilian Postolache, Valentino Maiorca, Michele Mancusi, Riccardo Marin, and Emanuele Rodolà. Accelerating transformer inference for translation via parallel decoding. *arXiv preprint arXiv:2305.10427*, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face, 2023.

Abhishek Udupa, Kaushik Rajan, and William Thies. Alter: exploiting breakable dependences for parallelization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pp. 480–491, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993555. URL `https://doi.org/10.1145/1993498.1993555`.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena, 2023a.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using sglang. *arXiv preprint arXiv:2312.07104*, 2023b.

A Interpreter

We present the PASTA-LANG interpreter pseudocode in Algorithm 1.

Algorithm 1 PASTA-LANG Interpreter. The function GENERATE_NEXT_TOKEN decodes the next token from the underlying language model, where invalid PASTA-LANG tags are disabled. INVALID_TAGS returns the set of PASTA-LANG tags that are syntactically invalid given the prefix. ASYNC_GENERATE initiates asynchronous generation of a sequence of tokens that ends with `</async>` and returns a promise object immediately and synchronously; it may recursively initiate a nested asynchronous generation and it may also initiate a nested scope, making it mutually recursive with INTERPRET_SCOPE. SYNC synchronizes asynchronous generations initiated within a parallel scope and replaces `<promise/>` tags with the corresponding asynchronously generated content in the input text.

```
function INTERPRET_SCOPE(text)
    promises ← []
    while True do
        ▷ Syntactically invalid tags are disabled during next token generation.
        invalid ← INVALID_TAGS(text)
        token ← GENERATE_NEXT_TOKEN(invalid)
        text ← text + token
        if token == '<scope>' then
            ▷ We permit recursively interpretation of nested scopes.
            text ← text + INTERPRET_SCOPE(text)
        else if token == '<promise/>' then
            promise ← ASYNC_GENERATE(text + '<async>')
            promises ← promises + [promise]
        else if token == '</scope>' then
            ▷ Replace promise tags with their corresponding asynchronously generated content.
            text ← SYNC(text, promises)
            break
        else
            text ← text + token
        end if
    end while
    return text
end function
```

B Evaluation

B.1 Generation Quality

We report the quality of decoding output by scoring the responses generated by each technique using LLM Judge (Zheng et al., 2023a). We present the generation quality scores for each method by category in Tables 3 and 4. We compare the baseline Mistral model, the Mistral model fine-tuned on the training dataset without any PASTA-LANG annotations, the Mistral model fine-tuned on the training dataset without any PASTA-LANG annotations using KL divergence loss, the Mistral model fine-tuned using the APAR data preprocessing and decoded with APAR, the Mistral model prompted with SoT and the SoT router, and the Mistral model fine-tuned on the PASTA-LANG-annotated training dataset and decoded with the PASTA-LANG-interpreter.

	Baseline	Finetuned	Finetuned-KL	APAR	SoTR	PASTA
Overall	7.88 ± 0.02	$\mathbf{8.02 \pm 0.04}$	7.87 ± 0.03	6.21 ± 0.03	7.48 ± 0.04	7.55 ± 0.05
Writing	8.43 ± 0.12	$\mathbf{8.60 \pm 0.17}$	8.53 ± 0.15	8.13 ± 0.06	8.40 ± 0.20	8.10 ± 0.10
Roleplay	8.57 ± 0.15	$\mathbf{8.60 \pm 0.10}$	8.43 ± 0.12	7.27 ± 0.21	8.07 ± 0.12	8.03 ± 0.06
Math	6.89 ± 0.19	$\mathbf{10.0 \pm 0.00}$	4.44 ± 0.19	4.44 ± 0.19	6.78 ± 0.19	6.89 ± 0.19
Knowledge	8.43 ± 0.12	8.37 ± 0.15	$\mathbf{8.57 \pm 0.06}$	8.20 ± 0.17	8.07 ± 0.15	8.43 ± 0.06
Generic	8.47 ± 0.21	8.50 ± 0.10	$\mathbf{8.57 \pm 0.06}$	5.57 ± 0.15	7.57 ± 0.12	8.33 ± 0.06
Fermi	5.47 ± 0.32	4.93 ± 0.06	$\mathbf{6.20 \pm 0.10}$	4.27 ± 0.21	5.60 ± 0.30	4.57 ± 0.42
CF	$\mathbf{8.40 \pm 0.10}$	8.30 ± 0.17	8.27 ± 0.06	6.90 ± 0.17	7.20 ± 0.10	$\mathbf{8.40 \pm 0.00}$
CS	$\mathbf{8.70 \pm 0.10}$	$\mathbf{8.70 \pm 0.10}$	8.67 ± 0.15	7.00 ± 0.10	8.20 ± 0.10	8.63 ± 0.06
Coding	6.48 ± 0.08	$\mathbf{7.38 \pm 0.08}$	6.29 ± 0.25	1.48 ± 0.08	6.67 ± 0.30	5.43 ± 0.25

Table 3: Comparison of generation quality, judged by GPT-4 on Vicuna-80.

	Baseline	Finetuned	Finetuned-KL	APAR	SoTR	PASTA
Overall	$\mathbf{7.80 \pm 0.03}$	7.68 ± 0.04	7.70 ± 0.09	6.73 ± 0.05	7.38 ± 0.09	7.38 ± 0.05
Brainstorming	7.70 ± 0.10	7.70 ± 0.00	$\mathbf{7.83 \pm 0.12}$	5.93 ± 0.06	6.83 ± 0.15	7.33 ± 0.15
Classification	$\mathbf{7.63 \pm 0.06}$	7.57 ± 0.06	7.50 ± 0.10	5.40 ± 0.10	7.57 ± 0.06	6.70 ± 0.17
Closed QA	7.20 ± 0.17	7.03 ± 0.06	7.27 ± 0.15	$\mathbf{7.67 \pm 0.06}$	7.13 ± 0.35	7.20 ± 0.10
CW	$\mathbf{8.23 \pm 0.15}$	$\mathbf{8.23 \pm 0.06}$	$\mathbf{8.23 \pm 0.23}$	7.07 ± 0.21	8.07 ± 0.21	$\mathbf{8.23 \pm 0.06}$
General QA	8.20 ± 0.17	8.17 ± 0.32	$\mathbf{8.23 \pm 0.06}$	7.53 ± 0.06	7.70 ± 0.17	7.13 ± 0.21
IE	7.90 ± 0.10	7.70 ± 0.10	$\mathbf{8.00 \pm 0.20}$	6.80 ± 0.20	7.83 ± 0.06	7.17 ± 0.21
Open QA	$\mathbf{8.07 \pm 0.06}$	7.23 ± 0.06	6.87 ± 0.31	6.83 ± 0.32	6.77 ± 0.12	7.37 ± 0.15
Summarization	7.50 ± 0.10	7.80 ± 0.10	7.67 ± 0.25	6.57 ± 0.15	7.17 ± 0.06	$\mathbf{7.93 \pm 0.21}$

Table 4: Comparison of generation quality, judged by GPT-4 on Dolly Bench.

B.2 APAR USING SHAREGPT

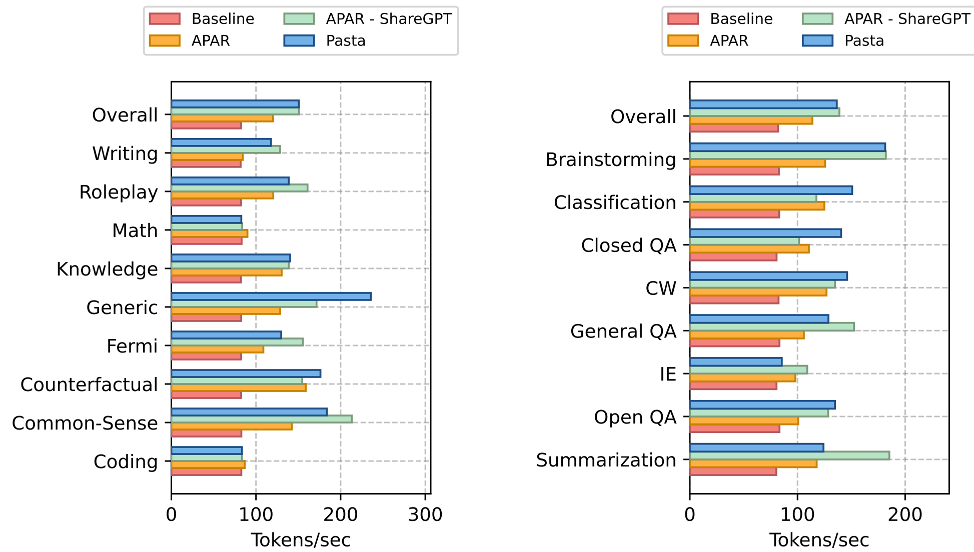
To standardize methodology during evaluation, we fine-tuned the Mistral-APAR model on the same training dataset used to fine-tune Mistral-PASTA without the PASTA-LANG tags. For completeness, we further fine-tuned the Mistral model on the ShareGPT dataset [3], following the pre-processing methodology from Liu et al. (2024), and evaluated it on Vicuna-80 and Dolly Bench.

Table 5 shows the number of questions in each benchmark that Mistral-APAR-ShareGPT applied to. Mistral-APAR-ShareGPT applies to more questions than Mistral-APAR in general and applies to about the same number of questions as Mistral-PASTA on Vicuna-80. However, it still applies to fewer questions than Mistral-PASTA on Dolly Bench. We present the throughput in Figure 4. Mistral-APAR-ShareGPT is competitive with Mistral-PASTA in throughput, an improvement from Mistral-APAR. This may be due to the APAR-preprocessed ShareGPT dataset being larger than the PASTA dataset (25k vs. 22k). It also contains more structured responses than the PASTA dataset. We also evaluated the generation quality of Mistral-APAR-ShareGPT, shown in Table 6 and Table 7. While Mistral-APAR-ShareGPT produces higher quality results than APAR on Vicuna-80, it is still inferior to Mistral-PASTA and it decreased in quality on Dolly Bench.

Method	Vicuna-80	Dolly Bench
APAR	54/80	50/80
APAR-ShareGPT	62/80	59/80
PASTA	61/80	72/80

Table 5: Number of questions that each method applies to.

[3] `https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered`



(a) Vicuna-80

(b) Dolly Bench

Figure 4: Generation speed of method compared to Mistral-APAR trained on ShareGPT. We do not count PASTA-LANG-tags or APAR control tokens towards throughput.

	Baseline	APAR	APAR-ShareGPT	PASTA
Overall	7.88 ± 0.02	6.21 ± 0.03	7.12 ± 0.08	7.55 ± 0.05
Writing	8.43 ± 0.12	8.13 ± 0.06	7.20 ± 0.26	8.10 ± 0.10
Roleplay	8.57 ± 0.15	7.27 ± 0.21	8.07 ± 0.23	8.03 ± 0.06
Math	6.89 ± 0.19	4.44 ± 0.19	6.78 ± 0.19	6.89 ± 0.19
Knowledge	8.43 ± 0.12	8.20 ± 0.17	8.03 ± 0.12	8.43 ± 0.06
Generic	8.47 ± 0.21	5.57 ± 0.15	7.80 ± 0.00	8.33 ± 0.06
Fermi	5.47 ± 0.32	4.27 ± 0.21	4.23 ± 0.31	4.57 ± 0.42
CF	8.40 ± 0.10	6.90 ± 0.17	7.13 ± 0.12	8.40 ± 0.00
CS	8.70 ± 0.10	7.00 ± 0.10	8.37 ± 0.15	8.63 ± 0.06
Coding	6.48 ± 0.08	1.48 ± 0.08	5.81 ± 0.16	5.43 ± 0.25

Table 6: Comparison of generation quality, judged by GPT-4 on Vicuna-80.

	Baseline	APAR	APAR-ShareGPT	PASTA
Overall	7.80 ± 0.03	6.73 ± 0.05	6.61 ± 0.12	7.38 ± 0.05
Brainstorming	7.70 ± 0.10	5.93 ± 0.06	5.90 ± 0.10	7.33 ± 0.15
Classification	7.63 ± 0.06	5.40 ± 0.10	7.13 ± 0.15	6.70 ± 0.17
Closed QA	7.20 ± 0.17	7.67 ± 0.06	5.87 ± 0.72	7.20 ± 0.10
CW	8.23 ± 0.15	7.07 ± 0.21	6.67 ± 0.06	8.23 ± 0.06
General QA	8.20 ± 0.17	7.53 ± 0.06	6.67 ± 0.15	7.13 ± 0.21
IE	7.90 ± 0.10	6.80 ± 0.20	7.37 ± 0.15	7.17 ± 0.21
Open QA	8.07 ± 0.06	6.83 ± 0.32	6.37 ± 0.21	7.37 ± 0.15
Summarization	7.50 ± 0.10	6.57 ± 0.15	6.93 ± 0.12	7.93 ± 0.21

Table 7: Comparison of generation quality, judged by GPT-4 on Dolly Bench.

B.3 PASTA-LANG BY PROMPTING

To justify the cost fine-tuning the Mistral-PASTA model, we evaluate the performance of simply prompting the Mistral model to use PASTA-LANG tags in its output and decoding the output with the PASTA-LANG interpreter. The prompt used is shown below.

PASTA Prompt

Introduction to PASTA
PASTA is an XML-like language designed to specify parallelism in language model decoding. We present the syntax and semantics of the language in this section.
Syntax PASTA introduces three tags: <scope>, <async>, and <promise/>. PASTA follows the standard HTML syntax rules in that <scope> tags can be nested arbitrarily within <scope> and <async> tags, and there can be any number of sibling <scope> elements within a level. PASTA imposes one additional constraint: every <async> element must be a descendant of a <scope> element.
Semantics The tags <async> and </async> mark a sequence of tokens that language models can generate asynchronously. The PASTA interpreter will continue generating tokens after an <async> tag without waiting for the tokens generated between the <async> tags to complete.
The <promise/> tags mark where asynchronous generation are located.
To define points where these asynchronous generations must synchronize, we introduce the <scope> tag. The tags <scope> and </scope> mark the beginning and end of a parallel scope. A parallel scope keep tracks of the asynchronous generations initiated within its bounds. Upon reaching the </scope> tag, the interpreter waits for all asynchronous generations initiated within this parallel scope to complete, and retrieves the asynchronously generated content to insert into the corresponding <promise/> tags that initiated the asynchronous generation.
As a result, the tokens generated after a pair of <async> tags and before its corresponding synchronization point, marked by a closing </async> tag, is not conditioned on the tokens generated between the <async> tags.
Task description Your task is to generate responses with promise and scope tags. Note that the content within promise tags should be independent from one another; therefore, think carefully what to represent with the promise tags and what to left out of it. Remember:
- Within a <scope> node, always have some synchronously generated content; this ensures the ensuing synchronous generation will have some context to maintain the cohesion between asynchronously generated blocks.
- <promise/> should not be be immediately followed by another <promise/> tag. There should be some synchronous content in between.
- You should never generate the <async> tags, only <promise/> and <scope> tags.
Here are some examples:
===============EXAMPLE=====================
Instruction: Summarize the following article: LONDON, England (Reuters) – Harry Potter star Daniel Radcliffe gains access to a reported £20 million ($41.1 million) fortune as he turns 18 on Monday, but he insists the money won't cast a spell on him. Daniel Radcliffe as Harry Potter in "Harry Potter and the Order of the Phoenix" To the disappointment of gossip columnists around the world, the young actor says he has no plans to fritter his cash away on fast cars, drink and celebrity parties. "I don't plan to be one of those people who, as soon as they turn 18, suddenly buy themselves a massive sports car collection or something similar," he told an Australian interviewer earlier this month. "I don't think I'll be particularly extravagant. "The things I like buying are things that cost about 10 pounds – books and CDs and DVDs."
Task: Generate a response using <promise/> and <scope> tags.
Full Response: <scope>Daniel Radcliffe, known for his role as Harry Potter, gained<promise/><async> access to a £20 million fortune upon turning 18.</async> Contrary to expectations,<promise/><async> he plans to spend modestly, preferring books and CDs over extravagant purchases. He has no plans to fritter his cash away on fast cars, drink, and celebrity parties.</async></scope>
Assistant Response <scope>Daniel Radcliffe, known for his role as Harry Potter, gained<promise/> Contrary to expectations,<promise/></scope>
...

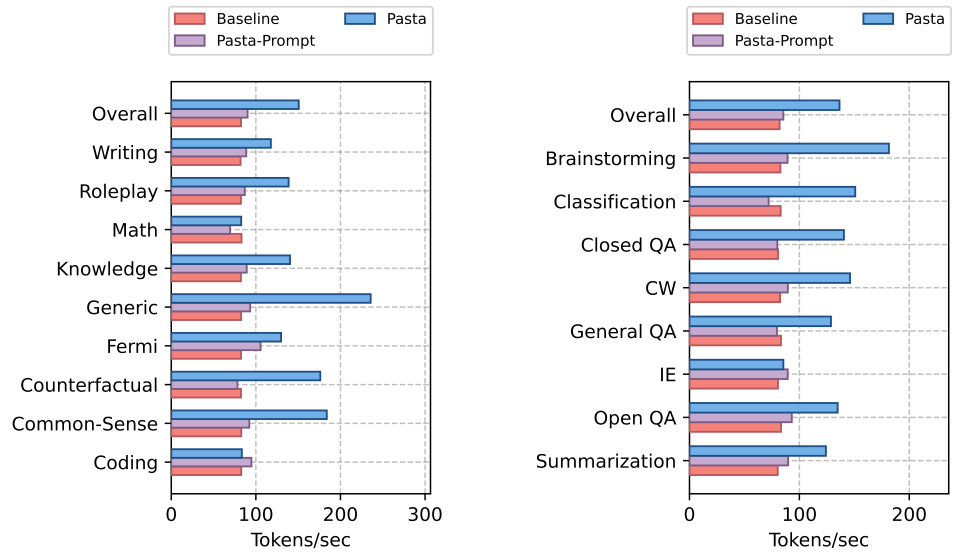
PASTA Prompt (Continued)

================EXAMPLE=======================
Instruction: Why mobile is bad for human?
Task: Generate a response using <promise/> and <scope> tags.
Full Response: Mobile devices have become an integral part of daily life for many people. However, excessive use of mobile devices can have negative effects on human health. Here are some of the ways in which mobile is bad for human:
<scope>
1. Eye Strain:<promise/><async> Constant exposure to the small screens of mobile devices can lead to eye strain, as the eyes have to focus on the screen for prolonged periods.</async>
2. Poor Posture:<promise/><async> Spending too much time hunched over a mobile device can lead to poor posture, which can cause neck and shoulder pain. This can be detrimental to human health.</async>
3. Social Isolation:<promise/><async> While mobile devices enable communication across vast distances, over-reliance on digital communication can sometimes lead to decreased face-to-face interaction and a sense of social isolation, particularly among younger generations.</async>
</scope>
These are just a few of the ways in which mobile devices can be bad for human health. It is important to find a healthy balance between using mobile devices and other aspects of daily life to maintain overall well-being.
Assistant Response: Mobile devices have become an integral part of daily life for many people. However, excessive use of mobile devices can have negative effects on human health. Here are some of the ways in which mobile is bad for human:
<scope>
1. Eye Strain:<promise/>
2. Poor Posture:<promise/>
3. Social Isolation:<promise/>
</scope>
These are just a few of the ways in which mobile devices can be bad for human health. It is important to find a healthy balance between using mobile devices and other aspects of daily life to maintain overall well-being.
================TEST=======================
Instruction: prompt
Task: Generate a response using only <promise/> and <scope> tags.
Assistant Response:

Table 8 shows that prompting the model applies the PASTA-LANG annotations more than the fine-tuned Mistral-PASTA model does. However, the application is low quality, as Figure 5 shows prompting for PASTA-LANG does not increase throughput significantly. Table 9 and Table 10 further shows this method severely decreases the generation quality. This shows that to effectively exploit parallel structure, fine-tuning the Mistral model to reliably annotate responses with PASTA-LANG tags is necessary.

Method	Vicuna-80	Dolly Bench
PASTA-Prompt	78/80	78/80
PASTA	61/80	72/80

Table 8: Number of questions that each method applies to.



(a) Vicuna-80 (b) Dolly Bench

Figure 5: Generation speed of method compared to Mistral-PASTA-Prompt. We do not count PASTA-LANG-tags towards throughput.

	Baseline	PASTA-Prompt	PASTA
Overall	7.88 ± 0.02	5.15 ± 0.06	7.55 ± 0.05
Writing	8.43 ± 0.12	4.03 ± 0.31	8.10 ± 0.10
Roleplay	8.57 ± 0.15	6.07 ± 0.15	8.03 ± 0.06
Math	6.89 ± 0.19	2.00 ± 0.00	6.89 ± 0.19
Knowledge	8.43 ± 0.12	5.50 ± 0.00	8.43 ± 0.06
Generic	8.47 ± 0.21	5.33 ± 0.21	8.33 ± 0.06
Fermi	5.47 ± 0.32	5.80 ± 0.17	4.57 ± 0.42
CF	8.40 ± 0.10	5.60 ± 0.17	8.40 ± 0.00
CS	8.70 ± 0.10	6.37 ± 0.31	8.63 ± 0.06
Coding	6.48 ± 0.08	2.76 ± 0.30	5.43 ± 0.25

Table 9: Comparison of generation quality, judged by GPT-4 on Vicuna-80.

	Baseline	PASTA-Prompt	PASTA
Overall	7.80 ± 0.03	5.60 ± 0.01	7.38 ± 0.05
Brainstorming	7.70 ± 0.10	4.87 ± 0.21	7.33 ± 0.15
Classification	7.63 ± 0.06	3.83 ± 0.06	6.70 ± 0.17
Closed QA	7.20 ± 0.17	5.60 ± 0.10	7.20 ± 0.10
CW	8.23 ± 0.15	6.40 ± 0.26	8.23 ± 0.06
General QA	8.20 ± 0.17	5.90 ± 0.17	7.13 ± 0.21
IE	7.90 ± 0.10	4.77 ± 0.15	7.17 ± 0.21
Open QA	8.07 ± 0.06	6.33 ± 0.12	7.37 ± 0.15
Summarization	7.50 ± 0.10	7.10 ± 0.10	7.93 ± 0.21

Table 10: Comparison of generation quality, judged by GPT-4 on Dolly Bench.