LoPT: Lossless Parallel Tokenization Acceleration for Long Context Inference of Large Language Model

Wei Shao, **Lingchao Zheng**, **Pengyu Wang**, **Peizhen Zheng**, **Jun Li**, **Yuwei Fan**

Huawei

{shaowei99, zhenglingchao, wangpengyu15, zhengpeizhen2, lijun276, fanyuwei2}@huawei.com

Correspondence: fanyuwei2@huawei.com

Abstract

Long context inference scenarios have become increasingly important for large language models, yet they introduce significant computational latency. While prior research has optimized long-sequence inference through operators, model architectures, and system frameworks, tokenization remains an overlooked bottleneck. Existing parallel tokenization methods accelerate processing through text segmentation and multi-process tokenization, but they suffer from inconsistent results due to boundary artifacts that occur after merging. To address this, we propose LoPT, a novel Lossless Parallel Tokenization framework that ensures output identical to standard sequential tokenization. Our approach employs character-position-based matching and dynamic chunk length adjustment to align and merge tokenized segments accurately. Extensive experiments across diverse long-text datasets demonstrate that LoPT achieves significant speedup while guaranteeing lossless tokenization. We also provide theoretical proof of consistency and comprehensive analytical studies to validate the robustness of our method.

1 Introduction

With the advancing capabilities of large language models (LLMs) (Team et al., 2025; Yang et al., 2025; OpenAI, 2025), long-context inference scenarios-such as document analysis (Xu and Peng, 2025), agent applications (Luo et al., 2025)-have become increasingly important. However, processing long contexts introduces significant computational latency. Previous research has addressed this challenge by optimizing operators (Dao et al., 2022), model architectures (Gu and Dao, 2024; Yang et al., 2025; DeepSeek-AI, 2025), and inference frameworks (Lee et al., 2025). Despite these efforts, tokenization time has emerged as a major contributor to inference latency as context lengths grow. Unfortunately, accelerating long-context tokenization remains underexplored.

To address these issues, we propose LoPT, a novel framework for **Lo**ssless **P**arallel **T**okenization and comparable tokenization speed with delimiter-based methods. Specifically, LoPT follows the Split $\rightarrow$ Parallel Tokenization $\rightarrow$ Merge pipeline from (OpenLLMab, 2024) but introduces key innovations. During splitting, the text is divided into overlapping chunks of equal length. Tokenization is performed in parallel, and results are merged based on character-level position information of tokens in the original text. This approach ensures accurate identification of tokens in overlapping regions without expensive token ID comparisons. Additionally, LoPT dynamically adjusts chunk length when the length of overlap sequences is insufficient, guaranteeing lossless results while maintaining speed comparable to delimiter-based methods. An example of how LoPT processes a long text is also shown in Fig. 1.

Experimental validation on three long text datasets (LongBenchV2 (Bai et al., 2025), LEval (medium sub-datasets) (An et al., 2023), and ClongEval (Qiu et al., 2024)) spanning various domains, formats, and languages demonstrates that LoPT significantly accelerates tokenization while ensuring perfect consistency with standard tokenization outputs. We also provide a theoretical proof of losslessness and conduct extensive analysis studies to provide further insights.

In summary, the contributions of this paper can be summarized as follows:

(1) We propose a novel lossless parallel tokenization framework (LoPT) that innovatively addresses the issue of inconsistency in parallel tokenization through character position-based matching and dynamic chunk length adjustment. (2) We conduct extensive experiments on multiple long-text datasets spanning diverse domains, formats, and languages to demonstrate that our framework significantly ac-

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 33107–33122

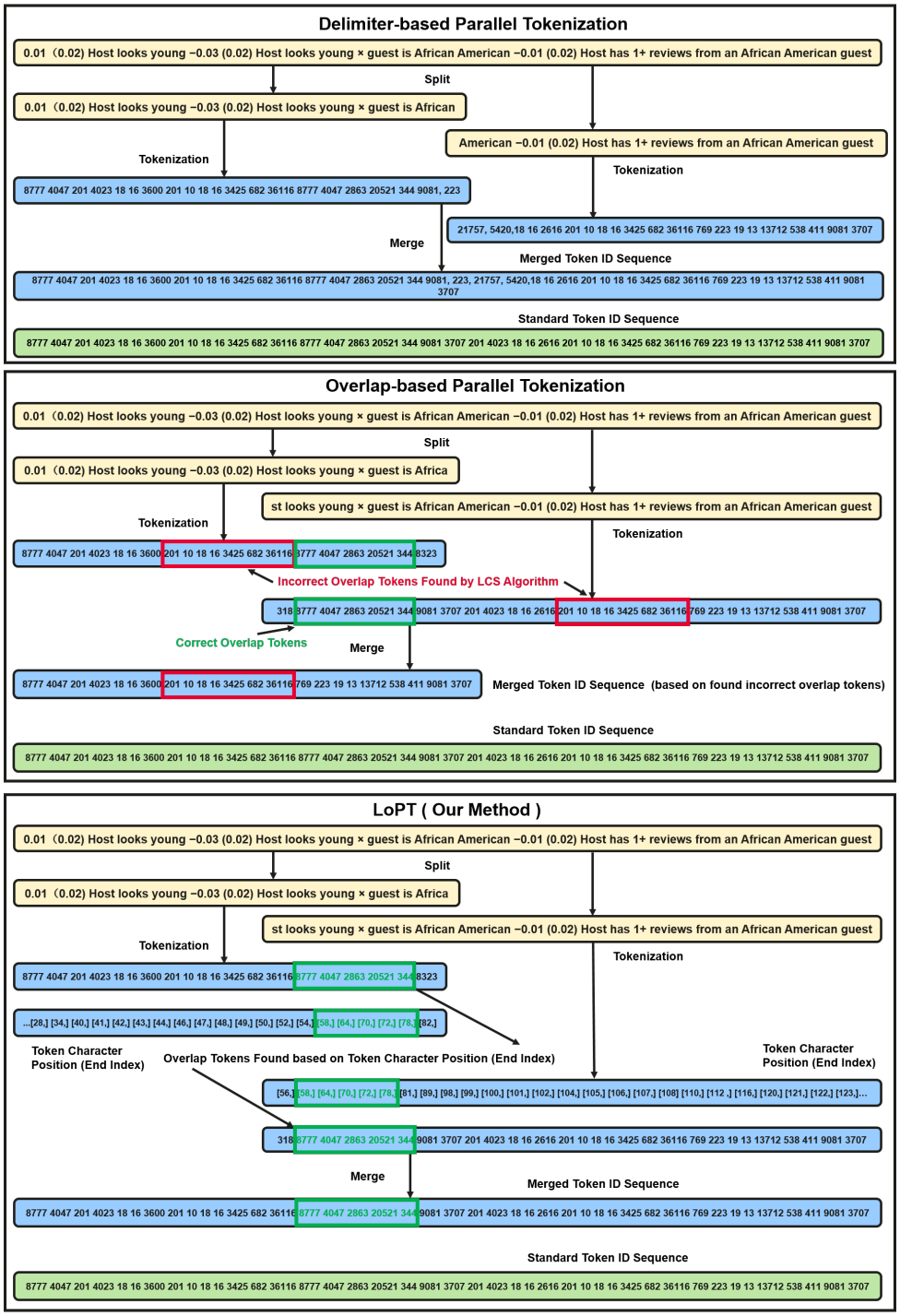

Figure 1: This figure shows the error cases of the previous chunk-based parallel tokenization methods (delimiter-based and overlap-based) and how our method LoPT obtains the correct result. Their outputs are in blue, and the standard tokenization results without splitting are in green. For delimiter-based parallel tokenization, incorrect results occur due to the neglect of token changes caused by variations in segment boundaries during the merge phase. For the overlap-based methods, although they attempted to address this issue by identifying overlap token sequences in the tokenization results of two adjacent text chunks, incorrect results still appear due to the mismatch of the overlap token sequence, which often occurs when processing certain texts, such as those containing consecutive repetitions. The LoPT identifies overlap tokens based on the token character position, which helps avoid such a mismatch.

celerates tokenization while guaranteeing lossless results. (3) We perform a comprehensive theoretical analysis and a series of analytical experiments to prove the lossless nature of our method and in-

vestigate its performance in different settings, providing both theoretical foundation and practical insight.

2 Related Work

2.1 Tokenization Algorithm and Acceleration

Tokenization is the process of converting text into input for large language models. In this paper, the term "tokenization" refers to the process of using a pre-constructed vocabulary to convert input text into token IDs for input to large language models. This process relies on a token vocabulary trained on large-scale data. To tokenize texts more efficiently, several classic algorithms have been developed, including BPE (Sennrich et al., 2016), WordPiece (Schuster and Nakajima, 2012), SentencePiece (Kudo and Richardson, 2018), and BBPE (Wang et al., 2020). Among these, BPE and WordPiece are widely employed by many large language models as tokenization algorithms due to their efficiency and universality. We mainly discuss these two algorithms in our paper. Current tokenization relies mainly on well-established software libraries such as Hugging Face Tokenizers (Hugging Face, 2018) and OpenAI TikToken (OpenAI, 2022). Both libraries have implemented numerous engineering optimizations. Note that these tokenizers work on CPUs. There are also efforts aimed at accelerating the tokenization process from the GPU side. For example, cuDF (RAPIDS, 2019) has achieved acceleration for WordPiece tokenization on GPU, and BlockBPE (You, 2025) has implemented a GPU-accelerated version of the BPE tokenization algorithm. Here, our focus is primarily on accelerating CPU-side tokenizers, as this remains the mainstream usage scenario and offers significant room for optimization. Beyond engineering implementations that accelerate the tokenization process, some efforts have focused on improving tokenizer throughput through multi-process optimization. However, this type of optimization targets batch inputs and does not reduce the latency for an individual sample. For inference scenarios, accelerating small-batch or even single-input processing is more critical, which is also the primary focus of our work.

2.2 Tokenization Acceleration for Long-Context

Although various acceleration techniques for tokenization have been mentioned earlier, some of these methods experience diminishing acceleration effects as the sequence length increases. Other techniques are less suitable for long-context scenarios—for example, multi-process acceleration for batch data processing, as batch sizes of long-context inputs are typically small. These limitations result in excessively long tokenization times for sequences that are excessively long. Currently, there are very few efforts specifically focused on accelerating tokenization for long contexts. Based on our research, only one software library, ParallelTokenizer (OpenLLMab, 2024), attempts to address this issue from a multi-process perspective. Their approach involves splitting the long context into several shorter text chunks with overlapping segments between adjacent ones, processing these shorter texts using multiple processes, and finally merging the results. However, their merging strategy is relatively simple: adjacent segments are combined based on the longest overlapping token sequences. Although this method achieves acceleration, it cannot guaranty that the merged result will be consistent with the output of direct tokenization on the original long context. Besides, due to the computation burden of finding an overlap token sequence, the acceleration performance of this method is not significant. These are the problems our work aims to solve.

3 Methodology

3.1 Overview

To achieve the harmony between lossless results and acceleration, we propose a novel lossless parallel tokenization framework (LoPT) for long context tokenization acceleration. This framework takes a long text as input and outputs its corresponding tokens. As shown in Fig. 2, the framework consists of three modules: the text split module, the parallel tokenization module, and the position-aware token merge module. The functions of each module are as follows:

Text Split Module: Receives the long text input and splits it into several overlapped text chunks based on predefined chunk length and overlap length.

Parallel Tokenization Module: Utilizes multi-processing to tokenize each text chunk in parallel.

Position-Aware Token Merge Module: Merging two adjacent text chunks' tokenization results based on their overlap token sequence. For each token in the overlap token sequence, it is essential to

ensure that its character position in the original long text, calculated based on its character positions in the two adjacent text chunks, remains consistent.

In addition, we also introduce a dynamic chunk length mechanism to ensure the existence of an overlap token sequence between two adjacent text chunks. These three modules and this mechanism work together to ensure faster tokenization and a lossless result. In the following, we will introduce them in detail.

3.2 Lossless Parallel Tokenization

3.2.1 Text Split Module

To ensure the lossless result of the parallel method, a certain overlap region is required between adjacent text chunks. Therefore, we adopt a process similar to the ParallelTokenizer, dividing the long text S into sub-segments $\{s_1, s_2, ..., s_N\}$ based on a fixed chunk size L_c and overlap length L_o. This process could be represented as:

$$s_i = S[L_c * (i-1) : L_c * i + L_o]. \quad (1)$$

3.2.2 Parallel Tokenization Module

This module serves as the primary component for acceleration. It speeds up the tokenization process by leveraging multi-process parallelization to invoke the tokenizer for processing. Moreover, due to the need for subsequent matching based on token character position information, in this step, we require the tokenizer to output the tokens along with their corresponding character positions within each text chunk. The whole process could be represented as:

$$(T_i, P_i) = Process_i(Tokenizer(s_i)), \quad (2)$$

where s_i is the i-th text chunk, T_i is the token id list of s_i and P_i is the character-level position of each token in s_i.

3.2.3 Position-Aware Token Merge Module

This module ensures lossless final results and consists of two steps: match and merge. The match step aims to find the longest overlap token sequence between adjacent text chunk tokenization results based on the token's character positions. The merge step aims to merge these tokenization results based on the overlap token sequences found.

Specifically, during the match step, we incorporate token-to-character position correspondence to ensure that matched overlapping tokens fall within the overlapping character range of both text chunks. The identified token sequences must align exactly at their starting and ending character boundaries to prevent missing or extra tokens caused by misalignment. Moreover, since position information is inherently ordered, we can leverage algorithms with lower computational complexity to identify overlapping token sequences, thereby reducing the time cost of this component compared to the previous method. During the merge step, given the overlapping region [a, b] between text chunks A and B, we retain the tokens from text chunk A before position a identified overlap token sequence, and the tokens from text chunk B after position b, then concatenate them. This process is repeated iteratively for the remaining text chunks.

According to our experiment results, a sufficient length of the overlap token sequence is necessary for lossless tokenization results. However, if the preset chunk length is not suitable, we may not find such an overlap token sequence. To this end, we evaluate whether the current chunk length is appropriate based on the length of the obtained overlap token sequence. If it is smaller than a threshold, the text chunk length will be increased, and the whole process will restart based on the new chunk length.

Formally, the match process can be represented as:

$$l_i, r_i, n_i^o = Match(P_i, P_{i+1}) \quad (3)$$

, where l_i and r_i are indexes of the start token of overlap tokens in the T_i and T_{i+1}. n_i^o is the number of valid overlap tokens. **Note that the valid overlap tokens' char-level positions in s_i and s_{i+1} must have a fixed offset: chunk length. This means that the global char-level position (in the long text) of the token is computed based on the char-level position in s_i is the same as that computed based on the char-level position in s_{i+1}.**

In the match process, if two adjacent text chunks have no sufficient overlap tokens in the long text ($n_i^o = 0$), this match will be regarded as a failed match, and the current chunk size will be doubled as the input parameter of a new long text split process. If the match is successful, we will merge token lists $T_1, T_2, ..., T_N$ as the final token list T based on the following process:

$$T = concatenate(T_1[: l_1 + n_1^o], T_2[r_1 + n_1^o : l_2 + n_2^o], ..., T_N[r_{N-1} + n_{N-1}^o :])$$

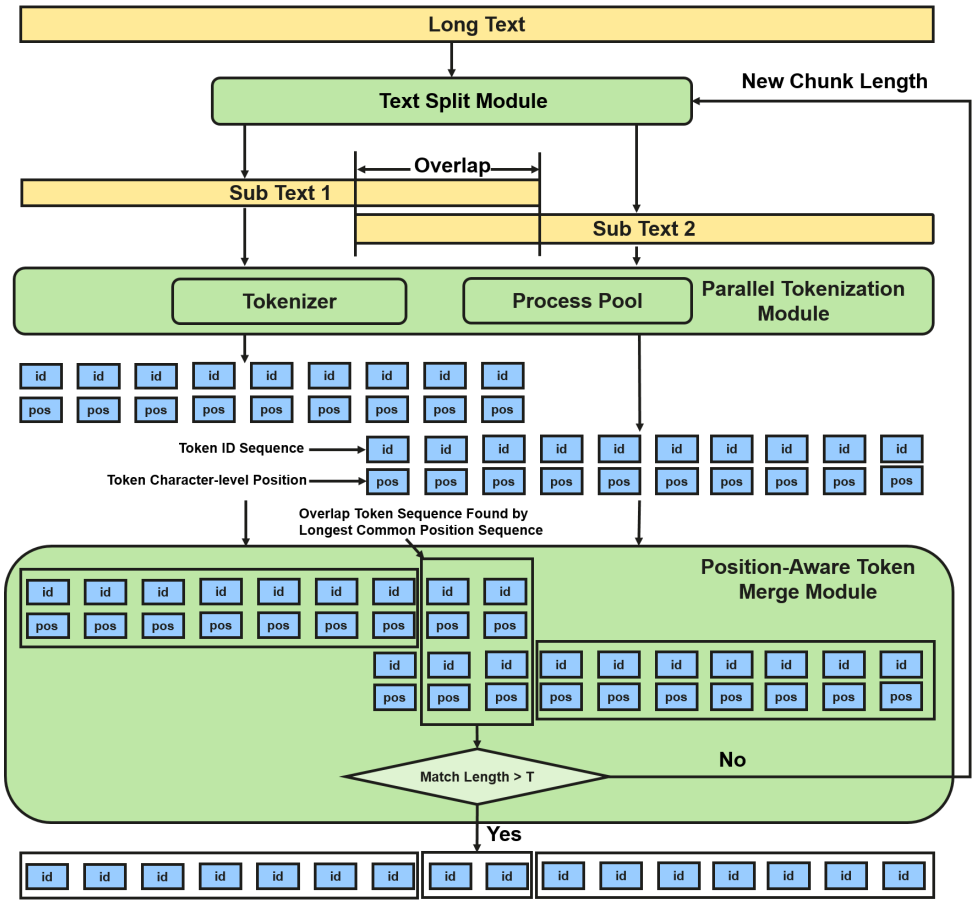

Figure 2: This figure illustrates the workflow of our method, with the green sections representing our method modules. The input long text is first split by the text split module into text chunks of chunk length, with adjacent text chunks having an overlap of overlap length. Then, each text chunk is processed by the parallel tokenization module to generate the corresponding token sequence. Unlike previous methods, at this step, we also output the character-level position of each token within the corresponding text chunk, providing the necessary positional information for merging the tokenization results. Finally, the token merge module receives the tokens and token position information of each text chunk and performs the merging operation. In this step, chunk length will be adjusted based on the matched tokens' number.

3.3 Theory Analysis

Here, we proved that what condition is met, the overlap-based parallel tokenization method can ensure consistency between the merged results and correct results. In previous overlap-based parallel tokenizer methods, the overlap tokens identified are not truly overlapped. For tokens to be considered overlapping, they must satisfy the condition that their span start positions in the original long text are consistent, which is ensured by our framework. In Appendix Section C, we proved that merging based on such overlap tokens can ensure that the final result of overlap-based parallel tokenization is consistent with the standard results of the two types of tokenization algorithms, as stated in Theorem 3.1.

Theorem 3.1. *Denote* $s_i = S[L_c * (i-1) : L_c * i + L_o]$, $(T_i, P_i) = Tokenizer(s_i)$, $(l_i, r_i, n_i^o) = Match(P_i, P_{i+1})$, *and* $T = concatenate(T_1[: l_1 + n_1^o], T_2[r_1 + n_1^o : l_2 + n_2^o], ..., T_N[r_{N-1} + n_{N-1}^o :])$.

If $len(s_i^o) > l_{max}, 1 \leq i \leq N-1$, *then*

$$T = Tokenizer(S).$$

, where s_i^o *is the character length of the overlapped tokens',* l_{max} *is the maximum character length of the tokens in the vocabulary.*

4 Experiment

To validate the effectiveness and robustness of our proposed framework, we conduct a series of comparative and analytical experiments.

4.1 Comparison Experiment

4.1.1 Experiment Settings

Datasets We use three long-text datasets: LongBenchV2, LEval (medium sub-datasets), and ClongEval as benchmark datasets. The LongBenchV2 and LEval consist of long English texts, and the ClongEval consists of long Chinese texts. The average text length of LongBenchV2 is significantly longer than that of the other two datasets.

Baselines **HuggingFace Tokenizer Fast**: It is the fast version of the tokenizer provided by HuggingFace and used to provide standard tokenization results and time baseline here. To ensure the robust and generalization of experiments, we use tokenizers from different model families, including BERT-Base-Uncased, and BERT-Base-Cased (Devlin et al., 2018), Qwen3 (Yang et al., 2025), DeepSeek-V3 (DeepSeek-AI, 2024), Llama-3.1 (Patterson et al., 2022), GPT-OSS-120B (OpenAI, 2025), Among them, the tokenizers of the first two models use the WordPiece algorithm, while the tokenizers of the latter four models use the BPE algorithm.

Delimiter-based ParallelTokenizer: Here, we employ three types of delimiters: whitespace, comma, and period. To obtain a fair comparison, we first choose the right delimiter around the length of the chunk.

Overlap-based ParallelTokenizer: Here, we use the ParallelTokenizer (OpenLLMab, 2024) as the representative method. The original version was implemented in Python, and the algorithm used to identify the sequence of overlap tokens could be improved. To ensure a fairer comparison, we reimplement the framework and optimize its token merge algorithm for more accurate results. We will list the results of the two versions in the following comparison experiment. (ParallelTokenizer-Origin, ParallelTokenizer-Ours)

Implementation Details We employ the HuggingFace fast tokenizer as the foundational tokenizer in our framework. To achieve optimal speed, the token merge module is implemented in C++, while the remaining components are developed in Python. Our framework defaults to using 32 processes for acceleration, with the chunk length set to the input text length divided by the number of processes. To obtain more accurate performance measurements for long-text processing, we set the input batch size to 1. The threshold for the minimum length of the overlap token sequence (T) is 2.

Metrics We measured the end-to-end **latency** of the standard HuggingFace TokenizerFast, our framework, and other baseline methods, as well as the **accuracy** of our framework and other parallel tokenization methods. Specifically, end-to-end latency means the time a tokenizer converts a sentence into a dictionary containing tensors like input IDs, attention masks. The accuracy refers to the proportion of results that exactly match those obtained by directly using the standard HuggingFace TokenizerFast.

Device Unless otherwise specified, all experiments were conducted in a CPU environment with a clock speed of 3.8 GHz and 112 cores.

4.1.2 Comparison Results

The experimental results are shown in Table 1. Compared to other methods, our approach is the only acceleration method that achieves an accuracy of 1 on all datasets with all tokenizers, which demonstrates the effectiveness and robustness of our framework. The Overlap-based ParallelTokenizer also achieves high accuracy on the three datasets, but its acceleration performance is significantly weaker than that of our method due to the higher complexity of its merge algorithm. While the remaining three delimiter-based methods achieve significant acceleration, their accuracy is relatively low and is influenced by the tokenizer's vocabulary and the language of the input context, which limits their generalization. Moreover, we can observe that BERT-like tokenizers achieve significantly higher accuracy when using the delimiter-based method compared to other tokenizers. This is because during preprocessing, BERT-like tokenizers typically treat punctuation and spaces as separate tokens, making the segmentation and merging results consistent for most texts. In contrast, other tokenizers tend to generate sub-word tokens where punctuation and spaces may be embedded within the tokens, leading to corruption of the token sequence after segmentation.

4.2 Analysis Experiment

4.2.1 Sequence Length's Impact

To investigate the performance differences between our method and the standard tokenization approach under varying sequence lengths (number of tokens), we conducted experiments at sequence lengths of

LLM	Tokenization Methods	LongBenchV2		LEval		ClongEval	
		Latency(ms)	Accuracy	Latency(ms)	Accuracy	Latency(ms)	Accuracy
BERT-BASE-CASED	HuggingFace TokenizerFast	509.2	-	23.6	-	63.5	-
	Delimiter based ParallelTokenizer (,)	93.7	0.970	6.7	0.966	18.1	0.834
	Delimiter based ParallelTokenizer (.)	91.4	0.956	7.0	0.953	**14.0**	0.868
	Delimiter based ParallelTokenizer (-)	**86.6**	0.996	6.5	0.994	16.6	0.715
	Overlap-based ParallelTokenizer-Origin	526.4	0.865	22.9	0.797	64.1	0.943
	Overlap-based ParallelTokenizer-Ours	479.5	0.982	7.6	0.996	29.0	0.997
	LoPT(Ours)	106.7	**1.0**	**6.3**	**1.0**	17.3	**1.0**
BERT-BASE-UNCASED	HuggingFace TokenizerFast	578.0	-	26.5	-	73.6	-
	Delimiter based ParallelTokenizer (,)	94.9	0.970	6.7	0.966	19.3	0.834
	Delimiter based ParallelTokenizer (.)	89.6	0.956	7.2	0.953	**16.3**	0.868
	Delimiter based ParallelTokenizer (-)	**83.8**	0.996	**6.6**	0.994	17.9	0.715
	Overlap-based ParallelTokenizer-Origin	571.5	0.867	31.3	0.793	75.2	0.945
	Overlap-based ParallelTokenizer-Ours	431.7	0.980	7.5	0.996	26.1	0.997
	LoPT(Ours)	107.1	**1.0**	**6.6**	**1.0**	18.5	**1.0**
Qwen3	HuggingFace TokenizerFast	618.5	-	33.2	-	69.9	-
	Delimiter based ParallelTokenizer (,)	110.9	0.247	5.7	0.315	21.1	0.576
	Delimiter based ParallelTokenizer (.)	103.7	0.328	6.4	0.559	**14.4**	0.677
	Delimiter based ParallelTokenizer (-)	**93.0**	0.708	**5.2**	0.806	15.3	0.711
	Overlap-based ParallelTokenizer-Origin	644.7	0.879	32.3	0.803	68.2	0.946
	Overlap-based ParallelTokenizer-Ours	332.4	0.982	8.5	0.998	18.6	0.999
	LoPT(Ours)	116.8	**1.0**	7.6	**1.0**	16.8	**1.0**
DeepSeek-V3	HuggingFace TokenizerFast	622.4	-	32.4	-	74.0	-
	Delimiter based ParallelTokenizer (,)	104.5	0.247	5.2	0.317	20.0	0.575
	Delimiter based ParallelTokenizer (.)	98.6	0.326	6.3	0.564	**14.1**	0.605
	Delimiter based ParallelTokenizer (-)	**91.2**	0.700	**5.1**	0.804	16.0	0.666
	Overlap-based ParallelTokenizer-Origin	648.2	0.889	40.6	0.801	80.5	0.946
	Overlap-based ParallelTokenizer-Ours	224.3	0.98	8.4	0.998	17.8	0.999
	LoPT(Ours)	108.2	**1.0**	7.2	**1.0**	15.6	**1.0**
Llama-3.1	HuggingFace TokenizerFast	513.3	-	26.1	-	60.4	-
	Delimiter based ParallelTokenizer (,)	91.2	0.247	**6.6**	0.315	18.2	0.573
	Delimiter based ParallelTokenizer (.)	87.1	0.328	7.8	0.559	**13.8**	0.665
	Delimiter based ParallelTokenizer (-)	**81.1**	0.708	7.4	0.806	15.8	0.711
	Overlap-based ParallelTokenizer-Origin	488.9	0.887	26.1	0.803	60.6	0.944
	Overlap-based ParallelTokenizer-Ours	218.7	0.980	7.9	0.998	19.4	0.998
	LoPT(Ours)	103.8	**1.0**	7.0	**1.0**	17.0	**1.0**
GPT-OSS-120B	HuggingFace TokenizerFast	500.6	-	26.9	-	66.3	-
	Delimiter based ParallelTokenizer (,)	98.3	0.245	**5.6**	0.315	19.9	0.572
	Delimiter based ParallelTokenizer (.)	92.0	0.324	5.7	0.561	**14.5**	0.610
	Delimiter based ParallelTokenizer (-)	**87.4**	0.708	6.1	0.806	15.9	0.711
	Overlap-based ParallelTokenizer-Origin	505.0	0.887	26.5	0.803	64.1	0.944
	Overlap-based ParallelTokenizer-Ours	218.5	0.982	7.8	0.998	18.9	0.998
	LoPT(Ours)	108.7	**1.0**	6.4	**1.0**	16.6	**1.0**

Table 1: The performance of different tokenization methods on three datasets. We use standard HuggingFace TokenizerFast from BERT-Base-Cased, BERT-Base-Uncased, Qwen3, DeepSeek-V3, Llama-3.1, and GPT-OSS-120B as the base tokenizers. The latency is the time (in milliseconds) required to convert a string to a dictionary consisting of a series of tensors. The accuracy is the proportion of results that exactly match those obtained by directly using the HuggingFace TokenizerFast. For the delimiter-based parallel tokenizer, ("-"), (","), (".") indicate that we use whitespace, comma, and period as the delimiters to split the long text. The best results are bolded.

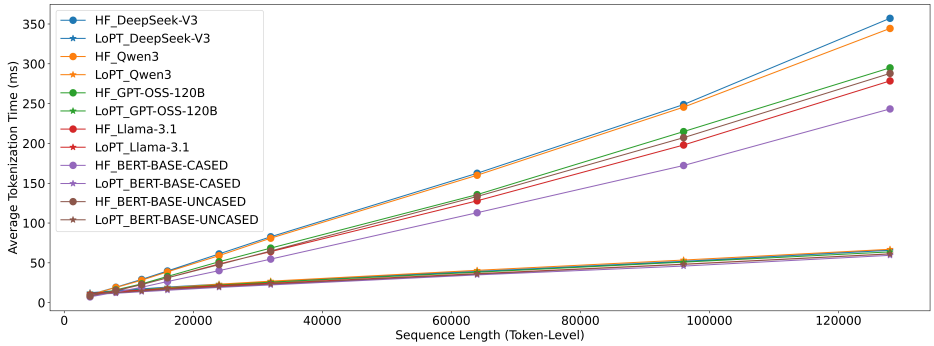

Figure 3: A performance comparison between the HuggingFace TokenizerFast ("·" points) and our framework LoPT ("*" points) on the LongBenchV2 dataset, with different sequence lengths. The horizontal axis represents the sequence length (number of tokens), and the vertical axis represents the tokenization time consumption (ms).

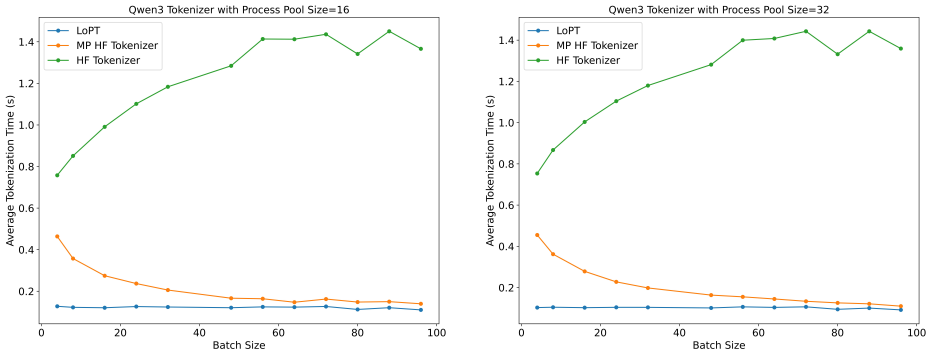

Figure 4: Performance of our framework and two baselines (we choose Qwen3 tokenizer as the base tokenizer) on the LongBenchV2 dataset with different batch sizes. The horizontal axis represents the batch size, and the vertical axis represents the time consumption for tokenization (s). HF Tokenizer and MP HF Tokenizer refer to the HuggingFace tokenizer and its sample-level, multiple-process version, respectively.

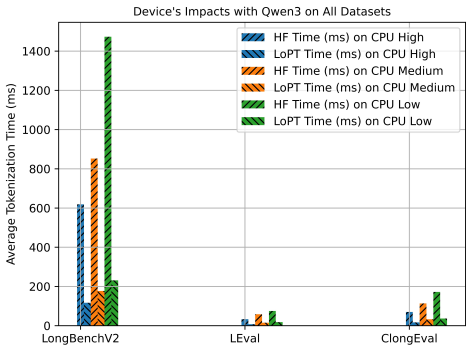

Figure 5: Performance of Qwen3 HuggingFace TokenizerFast and our framework working on devices with different computation capacity (CPU High > CPU Medium > CPU Low). The experiment is conducted on three datasets. The horizontal axis represents the dataset, and the vertical axis represents the time consumption for tokenization (ms). "/" represents the HuggingFace TokenizerFast, and "\" represents our framework.

4k, 8k, 24k, 32k, 40k, 48k, 64k, and 128k using the LongBenchV2 dataset. The performance of both the standard tokenizer and our framework is shown in Fig. 3. According to the results, we can observe that the processing time of the HuggingFace TokenizerFast increases linearly with the sequence length. Although our framework also exhibits linear growth, its slope is significantly smaller than that of the HuggingFace TokenizerFast. This indicates that the advantage of our method becomes more pronounced as the sequence length increases.

4.2.2 Batch Size's Impact

Although the batch size in long-context inference scenarios is generally small, to provide a more comprehensive comparison between our framework and HuggingFace TokenizerFast, we conduct tokenization experiments using the Qwen3 tokenizer on the LongBenchV2 dataset. The performance of our framework and HuggingFace TokenizerFast (HF tokenizer) based on different batch sizes is obtained. Additionally, considering that the HuggingFace tokenizer can be accelerated with multiprocessing at the sample level, we also implement a sample-level multi-processing version of the HuggingFace TokenizerFast (MP HF tokenizer) as a stronger baseline. The final results are shown in Fig. 4. According to the experimental results, the average tokenization time of our framework across different batch size settings remains significantly lower than that of the HF tokenizer. When the batch size is small, the processing time of our framework is less than that of the MP HF tokenizer. As the batch size increases, the processing time of our framework begins to approach that of the MP HF tokenizer. The batch size corresponding to the intersection point between our framework and the MP HF tokenizer is influenced by the process pool size.

4.2.3 CPU Device's Impact

Currently, tokenization computation is performed primarily on the CPU side. The time consumption of a tokenizer varies when working on CPUs with different computing capabilities. To investigate the performance differences of our framework on CPU with varying computing capabilities, we use the Qwen3 tokenizer as the base tokenizer and conduct experiments on three CPU devices with different computing capabilities (CPU High

Processes	Split (%)	Parallel Token. (%)	Match (%)	Merge (%)	Convert (%)	Actual Speedup	Theoretical Speedup ($s = 0.000935$)
1	0.09	99.91	0	0	0	1.00	1.00
2	0.28	98.07	0.40	1.11	0.16	1.70	1.99
4	0.53	96.58	0.73	1.83	0.30	2.84	3.97
8	0.92	94.99	1.28	2.35	0.46	4.38	7.88
12	1.23	94.29	1.49	2.60	0.45	5.55	11.70
16	1.17	94.67	1.61	2.19	0.44	6.24	15.45
20	1.51	93.89	1.82	2.30	0.48	6.78	19.14
24	1.39	94.45	1.71	1.96	0.49	6.98	22.77
28	1.44	94.17	1.77	2.11	0.42	7.23	26.34
32	1.44	94.05	1.70	2.13	0.77	7.27	29.85

Table 2: Stage proportions, actual speedup, and theoretical speedup under different numbers of parallel processes on LongBenchV2 dataset with DeepSeek-V3 tokenizer.

$>$ CPU Medium $>$ CPU Low). The time performance of both HuggingFace and our framework was obtained. Specific results are shown in Fig. 5. We can observe that the time consumption of our framework is more consistent across different computational power levels compared to the HuggingFace TokenizerFast, indicating that it is less affected by CPU capability and can achieve lower computational latency even on CPUs with lower performance. Moreover, the acceleration effect of our framework is more pronounced on CPUs with lower computational power. These results demonstrate that the performance of our framework is minimally influenced by CPU capability, exhibiting strong compatibility across various computing devices.

4.2.4 Parallel Speedup and Amdahl's Law Analysis

To evaluate the acceleration effect of the parallel tokenization stage, we conducted experiments on the LongBenchV2 dataset using the DeepSeek-V3 tokenizer, measuring the time consumption of each stage as the number of parallel processes varied from 1 to 32. In addition to reporting the overall time, we provide detailed time proportions for each stage and analyze the discrepancy between the theoretical speedup predicted by Amdahl's Law and the actual observed speedup. The processing pipeline of our method comprises five stages: **Split**: Partitioning the input text into multiple text chunks. **Parallel Tokenization**: Concurrently tokenizing the text segments using multiple processes. This is the primary parallel stage. **Match**: Performing necessary post-processing and matching operations on the tokenization results. **Merge**: Combining the tokenization outputs from different processes into a complete output. **Convert**: Transforming the merged token list into a PyTorch tensor format for subsequent model input.

Table. 2 presents the time proportions of each stage, the actual speedup, and the theoretical speedup according to Amdahl's Law under varying numbers of parallel processes. The serial fraction $s = 0.000935$ is derived from the non-parallel stage time (only the split stage, taking 0.0008 seconds) divided by the total time (0.8555 seconds) in the single-process case. As shown in this table, with an increasing number of parallel processes, the time proportion of the parallel tokenization stage gradually decreases from 99.91% in the single-process case to 94.05% at 32 processes, while the proportions of the other serial or low-parallelism stages (split, match, merge, convert) correspondingly rise. This trend aligns with the expectations of Amdahl's Law. The actual speedup improves as the number of processes grows but remains substantially lower than the theoretical speedup. For instance, at 32 processes the actual speedup is only 7.27, whereas the theoretical limit according to Amdahl's Law is 29.85 for 32 processes. The possible reason for the actual speedup being far below the theoretical value is additional time costs due to inter-process communication, synchronization, and task scheduling.

5 Conclusion

In this work, we identified tokenization as a critical yet under-optimized bottleneck in long-context LLM inference. While parallel tokenization offers a viable path to acceleration, it also leads to output inconsistency. To this end, we propose LoPT, a novel lossless parallel tokenization framework that effectively solves the inconsistency problem through character-position-based token merge and dynamic chunk length adjustment. Extensive experimental results validate that our framework achieves significant tokenization speedups across diverse datasets while fixing the output inconsistency.

Limitations

Although LoPT achieves the harmony of tokenization speed and accuracy, it also has some limitations. First, it needs the base tokenizer to provide character-level position information to ensure perfect merged results. Second, LoPT aims to accelerate the tokenization of small batch inputs in LLM inference scenarios and is not suitable for processing large batch inputs.

References

Chenxin An, Shansan Gong, Ming Zhong, Mukai Li, Jun Zhang, Lingpeng Kong, and Xipeng Qiu. 2023. L-eval: Instituting standardized evaluation for long context language models. *Preprint*, arXiv:2307.11088.

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. 2025. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *Preprint*, arXiv:2412.15204.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Preprint*, arXiv:2205.14135.

DeepSeek-AI. 2024. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

DeepSeek-AI. 2025. Deepseek-v3.2-exp: Boosting long-context efficiency with deepseek sparse attention.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Albert Gu and Tri Dao. 2024. Mamba: Linear-time sequence modeling with selective state spaces. *Preprint*, arXiv:2312.00752.

Hugging Face. 2018. Tokenizers: Fast State-of-the-Art Tokenizers optimized for Research and Production. GitHub repository. Accessed: 2026-04-18.

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

Ming-Yen Lee, Faaiq Waqar, Hanchen Yang, Muhammed Ahosan Ul Karim, Harsono Simka, and Shimeng Yu. 2025. Architecting long-context llm acceleration with packing-prefetch scheduler and ultra-large capacity on-chip memories. *Preprint*, arXiv:2508.08457.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, and 7 others. 2025. Large language model agent: A survey on methodology, applications and challenges. *Preprint*, arXiv:2503.21460.

OpenAI. 2022. tiktoken: a fast BPE tokeniser for use with OpenAI's models. GitHub repository. Accessed: 2026-04-18.

OpenAI. 2025. gpt-oss-120b & gpt-oss-20b model card. *Preprint*, arXiv:2508.10925.

OpenLLMab. 2024. ParallelTokenizer: Use the tokenizer in parallel to achieve superior acceleration. GitHub repository. Accessed: 2026-04-18.

David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2022. The carbon footprint of machine learning training will plateau, then shrink. *Preprint*, arXiv:2204.05149.

Zexuan Qiu, Jingjing Li, Shijue Huang, Wanjun Zhong, and Irwin King. 2024. Clongeval: A chinese benchmark for evaluating long-context large language models. *Preprint*, arXiv:2403.03514.

RAPIDS. 2019. cuDF - GPU DataFrame Library. GitHub repository. Accessed: 2026-04-18.

Mike Schuster and Kaisuke Nakajima. 2012. Japanese and korean voice search. *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, and 150 others. 2025. Kimi k2: Open agentic intelligence. *Preprint*, arXiv:2507.20534.

Changhan Wang, Kyunghyun Cho, and Jiatao Gu. 2020. Neural machine translation with byte-level subwords. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):9154–9160.

Renjun Xu and Jingwen Peng. 2025. A comprehensive survey of deep research: Systems, methodologies, and applications. *Preprint*, arXiv:2506.12594.

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, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Amos You. 2025. Blockbpe: Parallel bpe tokenization. *Preprint*, arXiv:2507.11941.

A BPE and WordPiece Tokenization Algorithms

We summarize the process of WordPiece and BPE tokenization algorithms in Algorithm 1 and Algorithm 2.

For BPE tokenization, assume that we have obtained vocabulary V and merge table M from BPE training, and we need to convert the input sentence S into a list of token IDs. The first step is to normalize and pre-tokenize the input sentence S. The detailed process depends on the specific tokenizer. After this step, we can obtain a list of words W. Then, for each word in W, we will split it into a basic character list C. For each adjacent pair in C, we will look up the merge table M to get its score and then determine and merge the best adjacent pair. This process will be repeated until no pairs can be merged. After obtaining a word's BPE tokens, we will map these tokens to their vocabulary IDs and append them to the final results T.

For WordPiece tokenization, the input sentence string S is split into a word list W, and each word is processed individually. For each word, the algorithm attempts to match the longest sub-word from the vocabulary V. Matched sub-words v_i are added to the output, while unmatched parts are replaced with an unknown token. This process repeats for each word in the input.

B Chunk-based Parallel Acceleration for Long-Context Tokenization

Chunk-based parallel tokenization acceleration methods can be divided into two categories. The first category involves splitting long texts based on a specific delimiter (such as periods or commas), then applying multi-process tokenization, and finally directly concatenating the tokenization results. However, this approach often alters the tokenization outcomes at the boundaries of the split text segments, leading to inconsistencies with the standard tokenization results. To alleviate this problem, the second method employs an overlapping segmentation method, where adjacent text segments are split with a certain degree of overlap. The tokenization results of these adjacent segments are then merged based on overlapping tokens. A representative method of this approach is the ParallelTokenizer (OpenLLMab, 2024).

Here, we describe how the ParallelTokenizer speeds up the tokenization of long contexts. According to Fig. 6, given the input long sequence text S, the ParallelTokenizer will split it into N shorter texts. A tokenization process will process each text chunk, and these texts are tokenized nearly simultaneously. Then, the token IDs of each adjacent text pair are merged. In this figure, we use two text chunks as an example to explain the merge process. The sub-text 1's tokens have a certain length of overlap with the sub-text 2's tokens. The ParallelTokenizer finds the overlap tokens (in yellow) with the longest common sequence algorithm based on token IDs. Then, the left tokens in sub-text 1 (in blue) and the overlap tokens are merged with the right tokens in sub-text 2 (in blue) to form the final

Algorithm 1: BPE Tokenization Algorithm

Input: Input sentence string: S;
Vocabulary: V; Merge Table: M.
Output: Token ids list of input sentence T.

```
Normalize and pre-tokenize S to get a list of words W ;
Initilize T = []
foreach i=0 to len(W)-1 do
    Split W[i] into a list of chars C;
    while True do
        (best pair, best score) = (null, 0);
        foreach j=0 to len(C)-1 do
            score = GET(M, C[j], C[j+1]);
            if score > best score then
                (best pair, best score) =
                 ((C[j], C[j+1]), score);
        if best pair is null then
            break // can no longer be merged
        foreach j=0 to len(C)-1 do
            if (C[j], C[j+1]) = best pair
             then
                MERGE(C[j], C[j+1]);
    Replace tokens in C with token id in V;
    Append C to T;
```

Algorithm 2: WordPiece Tokenization Algorithm

Input: Input sentence string: S;
Vocabulary: V;
Output: Token ids list of input sentence T.

Normalize and pre-tokenize S to get a list of words W ;
Initilize $T = []$
foreach *i=0 to len(W)-1* **do**
  while $W[i]$ **do**
    v_i = MatchLongestSubwordFromStart($W[i]$, V);
    if v_i **then**
      Append v_i's token id to the T;
      $W[i] = W[i] - v_i$;
    else
      Append unknown token's id to the T;

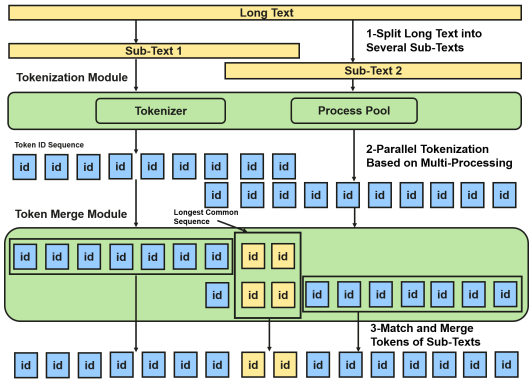

Figure 6: The ParallelTokenizer's process to parallelly tokenize a long text. Here, we use two text segments as an example.

tokens.

As mentioned earlier, although this method can achieve acceleration, it cannot guarantee that the final result will be consistent with that of standard tokenization. This is because relying solely on token ID matching does not ensure that the matched tokens correspond to the same positions in the original long context, which may result in missing or extra tokens in the merged output.

Moreover, we have observed that even if the tokens obtained from matching adjacent segments correspond to the same tokens in the original long text, there is still no guarantee that the final merged result will align with the standard tokenization output. This discrepancy arises because segmentation alters the context around overlapping tokens, potentially leading to the application of different rules during tokenization. As a result, the token IDs may differ from those generated by tokenizing the original long text directly.

C Proof of Theorem 3.1

In this section, we prove Theorem 3.1 for WordPiece-based tokenization and BPE-based tokenization. For simplicity, we denote $F(S)$ as the tokenization of sequence S, and $s(t)$ is the text span in S corresponding to t.

Without loss of generality, we only consider that S is divided into two parts, then Theorem 3.1 can be reformulated as the following lemma:

Lemma C.1. *$S = S_1 + S_o + S_2$, where S_o is the overlapped part. Then*

$$F(S_1{+}S_o) = [t_1^{(l)}, t_2^{(l)}, ..., t_m^{(l)}], \quad F(S_o{+}S_2) = [t_1^{(r)}, t_2^{(r)}, ..., t_n^{(r)}].$$

If $s(t_{i_1}^{(l)}) = s(t_{i_2}^{(r)}), s(t_{i_1+1}^{(l)}) = s(t_{i_2+1}^{(r)}), ..., s(t_{i_1+k}^{(l)}) = s(t_{i_2+k}^{(r)})$, *and*

$$len(s(t_{i_1}^{(l)}) + s(t_{i_1+1}^{(l)}) + ... + s(t_{i_1+k}^{(l)})) > T,$$

, where T = maxlen of words in the vocabulary, then we have:

$$F(S) = (t_1^{(l)}, t_2^{(l)}, ..., t_{i_1}^{(l)}, t_{i_1+1}^{(l)}, ..., t_{i_1+k}^{(l)}, t_{i_2+k+1}^{(r)}, ..., t_n^{(r)}).$$

C.1 Provement for WordPiece-based Tokenization

Lemma C.2. *If the first token generated during the tokenization of S is t (where the text span in S corresponding to t is $s(t)$) and there exists a substring S' of S satisfying $S' \subseteq S$ and $s(t) \subseteq S'$ then the first token generated during the tokenization of S' is also t.*

The validity of this lemma is self-evident.

Lemma C.3. *If $F(S) = (t_1, t_2, ..., t_n)$, then for $1 \leq l < r \leq n$,*

$$F([s_l, ..., s_r]) = (t_l, ..., t_r).$$

Proof. Considering the tokenization process of the original string S, we examine it token by token. If a token lies between t_l and t_r, it will also appear in $F([t_l, ..., t_r])$. If a token does not lie between t_l and t_r, it does not affect the result. This completes the proof of the lemma. □

Based on Lemma C.3, if the first token generated during the tokenization of S, the sub-string of S at left t is S_l, the sub-string of S at right t is S_r, which means that $S = S_l + s_t + S_r$, then

$$F(S) = F(S_l) + F(s_t) + F(S_r) \tag{4}$$

Returning to the original problem, we denote the split into left and right segments. After independent tokenization, the sub-string corresponding to overlapping tokens, i.e. $t_{i_1}^{(l)}, t_{i_1+1}^{(l)}, ..., t_{i_1+k}^{(l)}$, are denoted as M. It is not difficult to prove that M is a continuous string. Denoting the string to the left of M as L and the string to the right of M as R, we have $S = L + M + R$. Further, we have the following result:

If the length of M is greater than or equal to the maximum length of W_i, then the final result is correct, i.e.,

$$F(S) = F(L) + F(M) + F(R) \tag{5}$$

Proof. We prove this by the method of infinite descent. If the theorem does not hold, there exists a counterexample with the smallest $len(S)$. Based on the given conditions and Lemma 2, we have

$$F(L + M) = F(L) + F(M),$$
$$F(M + R) = F(M) + F(R)$$

Note that $S = L + M + R$. Consider the first token t generated during the tokenization of S. We proceed with a case-by-case analysis:

Case 1 $s(t) \subseteq L$ **or** $s(t) \subseteq R$**:** We may assume that $s(t) \subseteq L$. Denote $L = Ll + t + Lr$. Then t is also the first token of L during the tokenization process. Thus,

$$F(L) = F(Ll+s(t)+Lr) = F(Ll)+t+F(Lr),$$

$$F(S) = F(Ll) + t + F(Lr + M + R),$$
$$F(S) \neq F(L) + F(M) + F(R).$$

Thus we have

$$F(Ll)+t+F(Lr+M+R) \neq F(Ll)+t+F(Lr)+F(M)+F(R),$$

$$F(Lr + M + R) \neq F(Lr) + F(M) + F(R).$$

Considering the combination $Lr+M+R$, it forms a new and shorter counterexample, leading to a contradiction.

Case 2 $s(t) \subseteq M$**:** In this case, t is the first token in M. Thus,

$$M = Ml + s(t) + Mr,$$

$$F(S) = F(L + Ml) + t + F(Mr + R),$$
$$F(L + M) = F(L + Ml) + t + F(Mr),$$
$$F(M + R) = F(Ml) + t + F(Mr + R),$$
$$F(M) = F(Ml) + t + F(Mr).$$

On the other hand,

$$F(L+M) = F(L)+F(M) = F(L)+F(Ml)+t+F(Mr),$$

$$F(M+R) = F(M)+F(R) = F(Ml)+t+F(Mr)+F(R),$$

thus

$$F(L+Ml) = F(L)+F(Ml), \quad F(Mr+R) = F(Mr)+F(R).$$

Finally, we have

$$F(S) = F(L) + F(M) + F(R).$$

Case 3 $s(t) \subseteq L + M$ **or** $s(t) \subseteq M + R$**, but case 1 and 2 are not satisfied:** Without loss of generality, assume $s(t) \subseteq L + M$. Since t is the first token generated in S, it is also the first token generated in $L+M$. However, $s(t)$ does not belong to L or M, meaning t does not appear in $F(L)$ nor in $F(M)$, yet t is present in $F(L + M)$. This contradicts

$$F(L + M) = F(L) + F(M)$$

Case 4 $s(t)$ **does not belong to** $L + M$ **nor to** $M + R$**:** The starting point of t is before the starting point of M, and the ending point of t is after the ending point of M. Thus, $len(t) > len(M)$, which contradicts $len(M) \geq \max\ len$.

□

C.2 Proof for BPE-based Tokenization

Assume there exists a counterexample with the minimal number of fragments, where fragments refer to the unmerged parts. Then, by considering the first merge operation, the remaining steps of the proof are consistent with the proof for WordPiece.

D The LoPT Algorithm

In this subsection, to more clearly demonstrate the process of our proposed method, we summarize the processing flow of the LoPT into Algorithm 3.

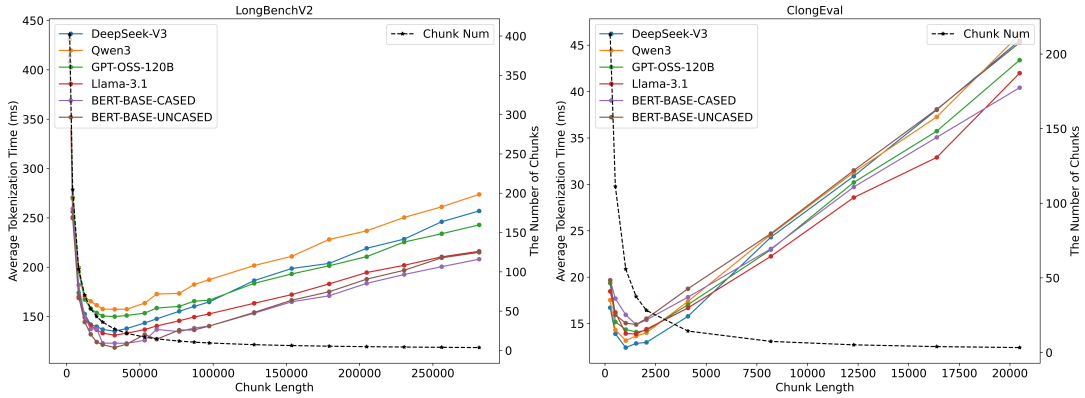

Figure 7: Performance of tokenization using our framework on LongBenchV2 and ClongEval datasets with different chunk lengths. The process pool size is 32. The horizontal axis represents the chunk length (character-level), and the vertical axis represents the tokenization time consumption. The dashed line represents the average number of chunks corresponding to different chunk lengths.

Algorithm 3: The LoPT Algorithm

Input: Long Text: S; A tokenization tool $Tokenizer$; The length of text chunk L_c, The length of overlap between two adjacent text chunks L_o, A process pool with size M

Output: Final token ID list of input sentence T.

Initialize $T = []$;
Initialize Overlap $O = []$;
Split S into N text chunks $s_i = S[L_c * (i-1) : L_c * i + L_o], i = 1, 2, .., N$;
Assign process ($Process_i$) from the process pool for each s_i's tokenization;
$(T_i, P_i) = Process_i(Tokenizer(s_i))$; // parallelly tokenize each text chunk s_i to obtain token ID list and token positions
foreach *i=1 to N-1* **do**
    $l_i, r_i, n_i^o = Match(P_i, P_{i+1})$; // get start tokens' indexes of overlap in T_i and T_{i+1} and the number of overlapped tokens
    if $n_i^o > 0$ **then**
        Append (l_i, r_i, n_i^o) to O;
    else
        L_c = L_c * 2;
        Re-split S with L_c and L_o;
$T = concatenate(T_1[: l_1 + n_1^o], T_2[r_1 + n_1^o : l_2 + n_2^o], ..., T_N[r_{N-1} + n_{N-1}^o :])$;

E Chunk Size's Impact

To investigate the performance of our framework with different chunk lengths (character-level), we conduct experiments to evaluate the performance of the LoPT across varying chunk lengths. The results are shown in Fig. 7.

According to this figure, in the initial stage, due to the relatively small chunk lengths, the number of chunks generated is large, and the process pool is insufficient. As the chunk length increases, the number of chunks decreases. Although the processing time per process becomes longer, the waiting time for fragmented processes is shortened, leading to a reduction in overall time. However, as the chunk length continues to grow and the number of chunks further decreases, idle processes emerge in the process pool, and the processing time per process increases even more. During this stage, as the chunk size increases, the overall time required also increases.

Additionally, it can be observed that the type of tokenizer relatively less influences the chunk length corresponding to the minimum tokenization time. Theoretically, this turning point is more significantly affected by the input length and the pool size. Longer prompts and larger pool sizes lead to a larger chunk size at this inflection point. Overall, the chunk length at this point falls within a range that brings the number of chunks close to the size of the pool.

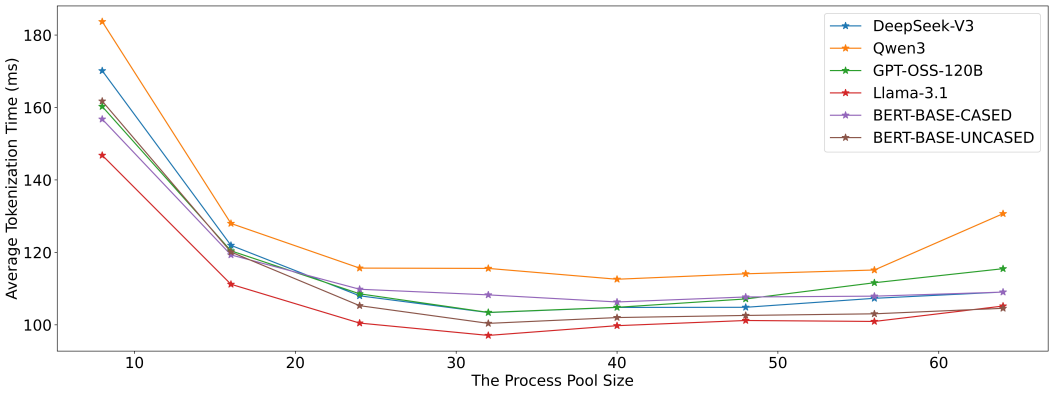

Figure 8: Performance of tokenization using our framework on the LongBenchV2 dataset with different process pool sizes. The horizontal axis represents the process pool size, and the vertical axis represents the time consumption for tokenization.

Tokenizer Model	**Dataset Type**	**Standard Time (s)**	**LoPT Time (s)**	**Avg. Retries**	**Speedup**
BERT-based-cased	repetitive-LongBenchV2	0.55	0.50	5.0	1.1
BERT-based-uncased	repetitive-LongBenchV2	0.62	0.64	5.0	1.0
DeepSeek-V3	repetitive-LongBenchV2	0.57	0.24	1.88	2.3
gpt-oss-120b	repetitive-LongBenchV2	0.54	0.24	1.88	2.3
Llama-3.1-8B	repetitive-LongBenchV2	0.53	0.24	1.88	2.3
Qwen3-8B	repetitive-LongBenchV2	0.60	0.26	1.88	2.3
BERT-based-cased	LongBenchV2	0.53	0.11	0.004	4.8
BERT-based-uncased	LongBenchV2	0.58	0.11	0.004	5.3
DeepSeek-V3	LongBenchV2	0.63	0.11	0.0	5.6
gpt-oss-120b	LongBenchV2	0.50	0.11	0.0	4.4
Llama-3.1-8B	LongBenchV2	0.49	0.10	0.0	4.7
Qwen3-8B	LongBenchV2	0.63	0.12	0.0	5.2

Table 3: Retry counts and speedup ratios of different tokenizers on repetitive and real-world datasets.

F The Process Pool Size's Impact

To investigate the impact of the number of processes on tokenization time, we conduct experiments using process pools of varying sizes on the LongBenchV2 dataset. The results are shown in the Fig. 8. Overall, as the process pool increases, the tokenization time of our framework continuously decreases. However, a larger number of processes does not necessarily lead to better acceleration performance. Among the configurations with an increasing number of processes, there exists an optimal number of processes that minimizes the time required for tokenization.

G Latency Risk Analysis of the Dynamic Retry Mechanism

This section evaluates the worst-case latency risk introduced by the dynamic retry mechanism in LoPT and provides its performance upper bound based on theoretical analysis and experimental verification.

G.1 Theoretical Analysis of Worst-Case Performance

In LoPT, each retry doubles the chunk length. The initial chunk length is set to 'prompt_length / num_processes', where 'num_processes' defaults to 32 (see the Appendix for detailed experimental settings). Consequently, a text segment will undergo at most 5 retries (i.e., up to 6 matching attempts) before falling back to the original text. Each retry re-executes the splitting, parallel tokenization, and matching operations. With 32 processes, parallel tokenization accounts for approximately 95% of the total time, splitting for 1.3%, and matching for 1%–2%.

Taking into account the worst-case scenario, we calculate the time ratio between LoPT and standard tokenization. For a given tokenizer with 32 processes, this ratio typically ranges between 1/7 and 1/5; we conservatively adopt the upper bound of 1/5. Therefore, the worst-case time (6 matching attempts) is at most $6 \times (1/5) = 1.2$ times that of standard tokenization. This upper bound ensures that the additional overhead remains limited even under extreme conditions.

G.2 Experimental Verification

To empirically validate the above upper bound, we construct a repetitive string dataset (repetitive-LongBenchV2) based on LongBenchV2. Specifically, we randomly select a character from 'a'–'z' and replace all characters in the input context with this single character. Experiments are conducted on both this synthetic dataset and the original LongBenchV2 to compare the performance of LoPT against standard tokenization (HuggingFace implementation). The results are summarized in Table. 3.

Repetitive Dataset (Worst Case): BERT-series tokenizers (based on WordPiece) exhibit poor robustness when handling repetitive strings, typically requiring the maximum number of retries (5) before falling back to the original text. Their speedup ratios are 1.1× and 1.0×, respectively, which are very close to the theoretical upper bound of $1.2\times$. This confirms that the overhead in the worst-case scenario is indeed bounded. In contrast, other tokenizers (e.g., those based on BPE) require only 1.88 retries on average to succeed and achieve a notable speedup of 2.3×. This indicates that the retry mechanism remains efficient for most tokenizers even under challenging inputs.

Real Dataset (LongBenchV2): Retry frequency is extremely low. BERT-based models trigger retries in only 0.4% of cases (average retries of 0.004), while other models exhibit no retries at all. Consequently, all tokenizers achieve substantial speedups ranging from 4.4× to 5.6×, with no performance degradation attributable to the retry mechanism.

In summary, the additional overhead introduced by the dynamic retry mechanism in the worst-case scenario does not exceed 20% of the standard tokenization time. On real-world datasets, retries are rarely triggered, and the mechanism yields significant acceleration benefits. This design ensures robustness without introducing unacceptable latency risks.