Evaluating and Improving Framework-based Parallel Code Completion with Large Language Models

Ke Liu[¶†‖], Qinglin Wang[¶†‖*], Xiang Chen[‡], Guang Yang[§], Yigui Feng[¶†], Gencheng Liu[¶†], Jie Liu[¶†]
[¶]Laboratory of Digitizing Software for Frontier Equipment, National University of Defense Technology, China
[†]College of Computer Science and Technology, National University of Defense Technology, China
[‡]School of Information Science and Technology, Nantong University, China
[§]College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China
Emails: {liuke23, wangqinglin, fengyigui, liugencheng, liujie}@nudt.edu.cn, xchencs@ntu.edu.cn, novelyg@outlook.com

***Abstract*—Modern computing architectures (e.g., multi-core CPUs, GPUs, distributed systems) rely on parallel code implemented via frameworks such as OpenMP, MPI, and CUDA. While large language models (LLMs) have shown strong performance in general code generation, they struggle with the structured reasoning required for parallel programming, such as handling concurrency, synchronization, and framework-specific semantics. In practical parallel code development, a common workflow begins with sequential code and incrementally introduces parallel directive codes. We formalize this process as the task of framework-based parallel code completion (FPCC), which involves three subtasks: identifying insertion points, selecting parallel frameworks, and completing parallel directive codes.**

To support this task, we construct a high-quality dataset of 16,638 framework-based parallel code pairs across six widely used frameworks, labeled with directive points, parallel frameworks, and the code of parallel directives. However, our empirical results show that six popular LLMs perform poorly on FPCC, particularly struggling with identifying insertion points and completing correct directive codes.

To address these limitations, we propose HPCL, a curriculum-based fine-tuning framework that progressively improves model capabilities in insertion point identification, parallel framework selection, and parallel directive code completion. Our approach achieves substantial improvements, yielding an 17.82% increase in EM and a 5.43% improvement in DIR scores over LLM-based baselines. Finally, expert-guided error analysis reveals common failure patterns and suggests future directions, such as in retrieval-augmented completion and consistency-aware training.

***Index Terms*—Parallel Programming Language, Large Language Model, Curriculum Learning, Code Completion**

I. Introduction

With the widespread adoption of multi-core CPUs, GPUs, and distributed computing platforms, parallel programming has emerged as a fundamental technique to achieve high performance and scalability, particularly in computationally intensive fields such as high performance computing (HPC) [1]–[8]. While modern parallel programming frameworks (e.g., OpenMP, CUDA, MPI) provide powerful abstractions for hardware concurrency, effectively parallelizing sequential code remains challenging. Developers should carefully reason about complex data dependencies, synchronization semantics, and specific architecture performance trade-offs, making manual parallelization both error-prone and expertise-intensive [9], [10].

In this work, we focus on transforming *sequential code* (i.e., conventional, single-threaded functions) into semantically equivalent *parallel directive code* augmented with explicit parallel directives. This transformation reflects a prevalent pattern in real-world software development, where a large volume of existing sequential code must be incrementally parallelized to meet modern performance demands. In practice, developers often begin with correct, functional sequential implementations and progressively insert parallel directives to exploit multicore, GPU, or distributed architectures.

Numerous efforts have been made to automate this transformation process to reduce the burden of manual parallelization. Compiler-based solutions [11], [12] and source-to-source transformation tools [13], [14] aim to automatically insert parallel directives based on static code analysis. However, these approaches often fall short when handling irregular control flows, dynamic memory behaviors, or multi-framework integration, limiting their applicability in real-world HPC codebases [15]–[19].

Recently, LLMs have demonstrated remarkable capabilities in general-purpose code generation and completion tasks [20]–[30]. However, a critical yet overlooked research question remains: *Can existing LLMs accurately insert parallel directives and select appropriate frameworks for sequential code in real-world multi-framework environments?* We refer to this task as **Framework-based Parallel Code Completion** (FPCC), focusing both on identifying insertion points, selecting frameworks, and completing parallel directives.

To systematically understand the practical challenges faced in parallel programming, we analyze the usage patterns of parallel frameworks in open-source HPC projects on GitHub. Our analysis reveals that 40.4% of repositories rely exclusively on a single parallel framework. In contrast, 59.6% of repositories integrate multiple parallel frameworks within the same project. For instance, developers frequently combine OpenMP for thread-level parallelism, MPI for distributed memory communication, and CUDA for GPU acceleration to address diverse performance goals and hardware constraints. Despite this prevalent multiframework usage in real-world development,

* Corresponding author.
‖ Both authors contributed equally to the paper.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00204

existing datasets and methods focus predominantly on single-framework scenarios, significantly limiting their applicability and practical relevance [31]–[39].

To fill this gap, we curate a large-scale dataset tailored for framework-based parallel directive code completion. Our dataset includes 16,638 labeled serial-parallel function pairs spanning six widely used frameworks (i.e., OpenMP, MPI, CUDA, SYCL, OpenCL, and TBB), with fine-grained labels for insertion points, parallel frameworks, and parallel directive codes. It captures realistic development patterns, including multiple directive insertions and mixed-framework usage within a single function.

Based on this dataset, we evaluate the current capabilities and limitations of popular LLMs (e.g., GPT-4 [40], CodeLLaMA-7B [21], and Qwen2.5-Code-7B-Instruct [41]) on the FPCC task under zero-shot and 3-shot prompting settings. The results indicate that all LLMs perform poorly on this task, struggling to correctly identify insertion points, select parallel frameworks, and complete parallel directive codes. A possible reason for this is that these LLMs may not have adequately learned knowledge related to framework-based parallel programming during their pre-training phase.

To address this limitation, we introduce **Hierarchical Progressive Curriculum Learning (HPCL)**, a fine-tuning strategy that decomposes the FPCC task into three incremental subtasks: (i) identifying insertion points, (ii) selecting frameworks, (iii) completing parallel directive code. HPCL aligns supervision with both task granularity and instance difficulty: subtasks are introduced progressively, and training is scheduled to begin with simpler examples before gradually incorporating harder ones.

Experimental results show that HPCL improves EM by 17.82% on Qwen2.5-Coder-7B-Instruct compared to standard fine-tuning, and yields consistent gains across both general-purpose and code-specific LLMs. Ablation studies further confirm that each curriculum axis independently contributes to performance, while their combination yields the strongest gains.

To further understand the limitations of LLMs, we conduct a structured expert error analysis and construct a hierarchical taxonomy of 20 fine-grained failure types. These include frequent issues such as missing or misused reduction clauses, incorrect variable scoping, and invalid synchronization patterns. This taxonomy provides a foundation for future improvements such as post-hoc repair and directive-level self-correction.

Compared to traditional compiler-based tools, which rely on static loop-level analysis and require compilable code, our approach is designed to be more flexible than compiler-based tools, which typically require fully compilable programs. This flexibility makes HPCL a promising complement to existing compiler techniques and suggests future opportunities for hybrid systems that combine compiler soundness with LLM-based adaptability.

In summary, our work makes the following key contributions:

- We are the first to formalize the *framework-based parallel code completion* (FPCC) task, which models the realistic software development process of incrementally parallelizing sequential code using parallel directives from multiple parallel frameworks.
- To support this task, we construct and release a large-scale dataset comprising 16,638 serial-parallel function pairs across six major parallel frameworks (i.e., OpenMP, MPI, CUDA, SYCL, OpenCL, and TBB).
- We evaluate popular LLMs on FPCC under zero-shot and few-shot prompting, and observe consistently low performance across identified insertion points and completed parallel directive codes.
- We propose HPCL, a hierarchical progressive curriculum learning strategy designed for FPCC. HPCL decomposes the task into three subtasks and incorporates difficulty-aware scheduling. Experiments show that HPCL outperforms standard fine-tuning and prompting baselines, improving DIR by up to 5.43% and EM by 17.82%. Ablation studies further confirm the benefits of both curriculum axes.
- We construct a hierarchical taxonomy of FPCC-specific LLM failures based on structured expert analysis. The taxonomy covers 20 fine-grained error subtypes across syntax, directive alignment, clause generation, and execution semantics, offering actionable insights for evaluation, targeted model improvement.

II. Preliminaries

A. Parallel Directive Code Completion Task Definition

Modern parallel programming frameworks provide high-level abstractions for expressing concurrency across diverse hardware platforms, including multi-core CPUs, GPUs, and distributed clusters. These frameworks enable developers to accelerate application performance by parallelizing compute-intensive regions, handling memory hierarchies, and coordinating thread or process-level execution. However, their effective use requires reasoning about data dependencies, synchronization, and hardware-specific trade-offs [9].

In practical development workflows, developers often begin with sequential code and incrementally transform performance-critical regions into parallel form. This process typically involves three steps: identifying suitable insertion points for parallelization, selecting an appropriate parallel framework based on workload characteristics, and completing correct directive code, often composed of multiple clauses such as scheduling strategies, reduction patterns, or data-sharing specifications. We define this structured, multi-step process as framework-based parallel code completion (FPCC): the task of transforming sequential code into semantically equivalent parallel code augmented with framework-specific directive statements.

Problem Statement. Given a sequential C/C++ function with one or more missing parallelization directives, the FPCC task requires a model to:

1) **Identifying Insertion Points.** Identifying one or more line numbers where parallel directive code should be inserted;
2) **Selecting Frameworks.** Select the most appropriate parallel programming framework from the set:

$$\{\text{OpenMP}, \text{CUDA}, \text{MPI}, \text{SYCL}, \text{OpenCL}, \text{TBB}\};$$

3) **Completing Parallel Directive Code.** Completing parallel directive code that adheres to framework-specific syntax, preserves the original program semantics, and supports efficient parallel execution.

This formulation supports multiple directive insertions per function, potentially involving heterogeneous frameworks. For instance, one function may benefit from OpenMP loop parallelization and MPI-based communication.

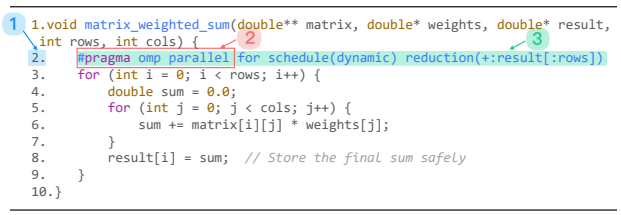

Fig. 1. Illustrative example of the FPCC task. Specifically, ① identifies insertion points (blue), ② indicates the selected parallel framework (red), and ③ completes the completed parallel directive code (green).

B. Parallel Code Completion Task Formalization

We formalize the FPCC task as a structured code editing problem that involves identifying insertion points, selecting parallel frameworks, and completing parallel directive code. Let:

- C_{input}: a C/C++ function with all parallel directives removed;
- $\mathcal{L}_{\text{insert}} \subseteq \{1, \ldots, N\}$: a subset of line numbers in C_{input} where parallel directive codes should be inserted;
- $F_{\text{framework}}$: the set of supported parallel frameworks;
- $D_{\text{directive}}$: a set of parallel directive codes, one per insertion point.

The objective is to learn a function:

$$f : C_{\text{input}} \rightarrow \{(l_i, f_i, d_i)\}_{i=1}^{k}$$

where $l_i \in \mathcal{L}_{\text{insert}}$, $f_i \in F_{\text{framework}}$, and $d_i \in D_{\text{directive}}$ represent the identified insertion points, selected parallel framework, and completed parallel directive code for the i-th parallelization site, respectively. This formulation supports multiple directives per input function and accommodates multi-framework scenarios.

Example. Fig.1 illustrates the FPCC task through a representative example. Given the sequential function `matrix_weighted_sum`, the model is required to perform three structured reasoning steps:

- **Step 1: Identifying Insertion Points.** The model identifies the loop spanning lines 3–9 as a parallelizable region and determines that line 2, which immediately precedes the loop, is the correct insertion point for a parallel directive code.
- **Step 2: Selecting Frameworks.** Given the CPU-centric loop structure, the model selects OpenMP as the appropriate framework.
- **Step 3: Completing Parallel Directive Code.** The model completes the OpenMP directive code at line 2.

The final output for this example can be formalized as a structured triplet:

```
(2, OpenMP, #pragma omp
parallel for schedule(dynamic)
reduction(+:result[:rows]))
```

III. Dataset Construction

To support the FPCC task, the trained model should learn to reason about parallelism in realistic code contexts: identifying insertion points, selecting parallel frameworks, and completing parallel directive codes. This requires training data with fine-grained supervision across diverse parallel programming paradigms. Existing datasets fall short of these requirements. General-purpose benchmarks like CrossCodeEval [42] and RepoCoder [43] cover only sequential code and lack any notion of parallelization. HPCORPUS [44] includes parallel directive code but lacks aligned serial-parallel pairs and directive-level labels. Meanwhile, prior work on parallel directive code generation [31]–[38] typically focuses on a single framework (e.g., OpenMP), limiting generalizability to multi-framework scenarios commonly seen in practice.

To fill this gap, we are the first to construct a comprehensive dataset specifically designed for FPCC. Specifically, our dataset offers three key innovations: (1) fine-grained multi-framework labels spanning six parallel frameworks, (2) aligned serial-parallel directive code pairs enabling systematic supervision, and (3) structured metadata supporting both model training and fine-grained evaluation.

A. Source Data Selection

To ensure legal compliance and practical relevance, we curated GitHub repositories based on the following criteria:

- **Legal Compliance**: We include only projects using permissive open-source licenses (e.g., MIT, Apache-2.0, BSD), ensuring legal redistribution and research reproducibility.
- **Code Quality**: We implement quality filters by selecting repositories with at least 100 stars and recent activity within 24 months, effectively excluding abandoned or low-quality codebases.
- **Parallel Framework Usage**: We target practical relevance by focusing exclusively on C/C++ projects utilizing our selected six parallel frameworks, capturing comprehensive coverage of both shared- and distributed-memory programming models.

Based on the above criteria, we finally selected 198 repositories and extracted 23,466 candidate functions containing parallel directive codes for further processing.

B. Data Preprocessing

Inspired by previous practices in constructing large-scale, high-quality datasets [21], [32], [38], [45], we adopt the following three stages.

1) **Code Normalization and Deduplication**: All code is reformatted using `clang-format` [46] and standardized to UTF-8. Duplicate functions are removed based on content hashing to eliminate redundancy and enhance the effectiveness of model supervision.
2) **Outlier Removal**: To ensure training efficiency, we apply IQR-based [47] filtering to retain functions within $[15,\ 750]$ tokens, removing both trivial snippets and overly complex samples.
3) **Serial-Parallel Pair Construction**: To support FPCC training, we construct aligned serial-parallel function pairs by following methodologies established in prior work [32], [38]. Starting from parallel directive code with framework-specific constructs, we construct serial counterparts by systematically removing directives. To ensure semantic preservation, we adopt a hybrid strategy that combines pattern-based matching with AST-guided rewriting, allowing for precise elimination across diverse frameworks.

After these preprocessing steps, we obtain 70.88% of candidates for further labeling and training. Finally, we construct semantically equivalent serial variants, averaging 5.12 directive removals per function, creating the precise supervision needed for FPCC training.

C. Data Labeling

A key innovation in our dataset is its comprehensive triple-target labeling scheme, specifically designed to support structured parallel directive code completion (Section II-B). Specifically, each labeled sample consists of a serial-parallel function pair, accompanied by one or more structured triplets of the form $(L_{\text{insert}}, F_{\text{framework}}, D_{\text{directive}})$. For each triplet, we record the following metadata:

- **Insertion Points (L_{insert})**: We precisely identify directive points through differential analysis between serial and parallel versions. By computing line-level diffs, we determine exact insertion points in the serial code.
- **Parallel Framework ($F_{\text{framework}}$)**: We identify parallel frameworks using a hybrid approach: AST traversal for parallel directive code-based frameworks (e.g., OpenMP) and specialized pattern matching for API-based frameworks (e.g., MPI). Specifically, we label each parallelization point independently, allowing accurate framework identification in multi-paradigm code where different frameworks coexist within the same function.
- **Parallel Directive Code ($D_{\text{directive}}$)**: We extract complete and executable parallel directive codes using AST-based slicing and statement-level alignment. This includes all semantic components (e.g., `#pragma omp parallel for schedule(dynamic) reduction(+:sum)`), capturing the full complexity of parallel directive codes with their clauses, dependencies, and optimization hints.

TABLE I
OVERVIEW OF DATASET SIZE AND STRUCTURED LABEL CHARACTERISTICS

Metric	Value
Frameworks/libraries	6
Total repositories collected	198
Labeled parallel-serial pairs	16,638
Average tokens per function	405.74 tokens
Parallel directive code	31.86 tokens
Average triplets per sample	5.12

D. Dataset Statistics

Table I summarizes the key properties of our dataset, which comprises 16,638 function-level serial-parallel pairs collected from 198 actively maintained open-source repositories. We partition the dataset into 13,310 training (80%), 1,664 validation (10%), and 1,664 test (10%).

The average function length is 405.74 tokens, providing sufficient context for directive-level reasoning. On average, each parallel directive code contains 31.86 tokens and often involves compound clauses such as `schedule(dynamic)`, `reduction(+:sum)`, or `shared(var)`, reflecting the syntactic and semantic complexity FPCC models must learn to handle. Our dataset covers six prevalent parallel programming frameworks: CUDA (5,501), MPI (4,444), OpenMP (3,863), OpenCL (1,214), TBB (1,047), and SYCL (218). This distribution spans diverse computational paradigms, including GPU acceleration, shared-memory parallelism, and distributed message-passing architectures.

IV. EVALUATING FPCC WITH LLMS

As FPCC remains largely unexplored in prior work, we begin by evaluating the baseline performance of large language models on this task. We conduct a comprehensive empirical study under both zero-shot and three-shot prompting settings, using a fixed test set derived from the dataset introduced in Section III.

A. Evaluation Metrics

To comprehensively assess model performance on the FPCC task, we adopt widely used code completion metrics to assess overall generation quality, while introducing task-specific metrics that independently evaluate three fundamental capabilities: insertion points identification, parallel framework selection, and directive code completion.

Exact Match (EM). EM measures the percentage of samples where the model's predictions perfectly match all reference labels (i.e., insertion points, frameworks, and directive codes). For samples with multiple directives, correctness requires matching all triplets in the ground truth.

$$\text{EM} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}(\hat{T}_i = T_i^*)$$

TABLE II
PERFORMANCE OF BASELINE LLMS ON THE PARALLEL DIRECTIVE CODE COMPLETION TASK UNDER ZERO-SHOT AND THREE-SHOT SETTINGS

Model	EM		STF		IP		FW		DIR	
	0-shot	3-shot	0-shot	3-shot	0-shot	3-shot	0-shot	3-shot	0-shot	3-shot
Llama3.1-8B	0.03	0.09	0.02	0.06	2.29	2.35	8.32	25.38	0.76	1.32
GPT-4	**0.43**	**1.81**	**0.37**	**1.36**	**6.83**	**7.66**	**32.29**	44.61	3.97	**5.87**
Qwen2.5-Coder-32B-Instruct	0.12	0.42	0.13	0.37	3.97	5.44	23.15	42.95	**4.02**	5.27
Qwen2.5-Coder-7B-Instruct	0	0.30	0	0.19	5.19	4.93	31.9	**70.32**	3.73	5.54
StarCoder2-7B	0	0	0	0	0.24	0.21	0.16	0.50	0.09	0.11
CodeLLaMA-7B	0	0	0	0	0.25	0.30	0.20	0.52	0.10	0.13

Notes. Metrics include EM (Exact Match), STF (Structured Triplet F1), IP (Insertion Point Prediction), FW (Framework Classification), and DIR (Parallel Directive Code Quality), all normalized to [0,1] with higher values indicating better performance.

where $\hat{T}_i = \{(L_{ij}, F_{ij}, D_{ij})\}_{j=1}^{k_i}$ and T_i^* are the predicted and ground-truth directive triplet sets for sample i, and k_i denotes the number of labeled insertions.

Structured Triplet F1 (STF). STF captures partial correctness by calculating the macro-averaged F1 score between predicted and reference directive triplets.

$$\text{STF} = \frac{1}{N}\sum_{i=1}^{N}\frac{2\cdot|\hat{T}_i \cap T_i^*|}{|\hat{T}_i| + |T_i^*|}$$

where $|\hat{T}_i \cap T_i^*|$ denotes the number of correctly predicted insertion-framework-directive triplets for sample i.

Component-Level Metrics. To assess model performance on the three core subcomponents of FPCC, we adopt task-specific evaluation metrics based on the prediction structure and output type of each subtask.

- **Insertion Point Prediction(IP)**: Measured as the macro-averaged F1 score between predicted and ground-truth line points where parallel directives should be inserted. Each sample may contain multiple insertion points, and prediction is treated as a set-based matching task over candidate line numbers.
- **Framework Classification (FW)**: Measured as the macro-averaged F1 score between predicted and ground-truth parallel frameworks (e.g., OpenMP, MPI, CUDA). This is evaluated as a multi-class classification task over a predefined framework label set.
- **Parallel Directive Code Quality (DIR)**: Assessed using CodeBLEU [48], a composite metric that integrates lexical, syntactic, and semantic information.

B. LLMs and Prompt Design

LLM Selection. To comprehensively evaluate the performance of LLMs on the FPCC task, we consider two representative categories of models: (i) **general-purpose LLMs**, which are trained to follow prompts across a wide range of tasks, and (ii) **code-specific LLMs**, which are pre-trained or fine-tuned explicitly for source code understanding and generation tasks.

For general-purpose LLMs, we include GPT-4 [49], and LLaMA3.1-8B [50]. For code-specific LLMs, we evaluate Qwen2.5-Coder-32B-Instruct [41], Qwen2.5-Coder-7B-Instruct [41], StarCoder2-7B [51], and CodeLLaMA-7B [21]. All models are accessed via public APIs or released checkpoints with default decoding parameters. We set the maximum input length to 2048 tokens and the generation limit to 512 tokens to maintain evaluation consistency across models.

Prompt Design. We evaluate each model under two prompting settings: *zero-shot* and *few-shot*. In the zero-shot setting, the model receives only the FPCC task instruction and the input sequential function. In the few-shot setting, three in-context demonstrations are accompanied by an input. Each demonstration contains a serial function followed by the corresponding directive completion triplet (L, F, D), formatted identically to the expected output.

To ensure fair comparison and reduce confounding effects, we manually selected three demonstrations corresponding to the three most frequent parallel frameworks in our dataset: OpenMP, CUDA, and MPI. This choice reflects common usage patterns and ensures that few-shot prompts provide representative examples of real-world directive types. All training hyperparameters, prompt templates, and implementation details are provided in our GitHub repository for reproducibility[1].

C. Evaluation Results

We evaluate six popular LLMs on FPCC under zero-shot and three-shot prompting, shown in Table II.

Prompting yields limited and model-dependent gains. FPCC performance remains poor across all models under prompting conditions. Even leading models like GPT-4 achieve minimal success (1.81% in EM, 5.87% in DIR) with three-shot prompting, while code-specialized models like StarCoder2-7B and CodeLLaMA-7B perform near zero.

Model performance on FPCC correlates more with pretraining domain than model size, with code-specialized models offering slight advantages. However, even these improvements remain marginal, suggesting inadequate parallel programming exposure during pretraining. The minimal gains from zero-shot to three-shot prompting further demonstrate that in-context learning cannot address the complex reasoning required for effective parallel code completion.

Structural sub-tasks remain the primary bottleneck. Our results reveal significant performance disparities across FPCC subtasks. Among all subtasks, parallel framework selection (FW) is the most tractable. For example, Qwen2.5-Coder-7B-Instruct achieves a top FW score of 70.32% in the three-shot setting, likely due to the presence of explicit contextual indicators.

[1]https://github.com/HPCL-Coder/HPCL

In contrast, identifying correct insertion points and completing syntactically valid and semantically appropriate directive code remain substantially more challenging. Even the best-performing model, GPT-4, achieves only 6.83% in IP and 5.87% in DIR. These results highlight the complexity of these subtasks, which demand multi-step analysis of control flow and variable scoping—capabilities beyond what current pretraining or few-shot prompting can effectively support.

Implications. These results highlight the fundamental limitations of in-context learning for FPCC, stemming from insufficient parallel programming exposure during pretraining. The significant performance variation across subtasks suggests two key requirements for effective FPCC models: (1) progressive task decomposition to address the varying complexity of different subtasks, and (2) difficulty-aware curriculum learning to accommodate the diverse complexity levels present in real-world parallel code.

V. Improving FPCC with LLMs

A. Our Proposed Approach

To alleviate the limitations identified in our empirical analysis, we introduce HPCL, a Hierarchical Progressive Curriculum framework for FPCC. As illustrated in Fig. 2, HPCL incorporates two complementary curriculum strategies: (i) sample difficulty estimation, which schedules training based on instance complexity, and (ii) multi-stage task decomposition, which incrementally introduces FPCC subtasks from simple to complex.

1) Sample Difficulty Estimation: We define a normalized difficulty score $D \in [0, 1]$ that quantifies the complexity of sequential-to-parallel directive code, incorporating three key components:

- **Cyclomatic Complexity (CC)**: Captures control-flow branching and the number of independent execution paths.
- **Lines of Code (LOC)**: Measures code size, serving as a proxy for cognitive load.
- **Number of Parallel Insertions (NPI)**: Indicates the number of parallel directive codes to be inserted, reflecting completion density.

The score is computed as:

$$D = \alpha \cdot \frac{CC}{CC_{\text{max}}} + \beta \cdot \frac{LOC}{LOC_{\text{max}}} + \gamma \cdot \frac{NPI}{NPI_{\text{max}}}, \quad \text{with } \alpha+\beta+\gamma = 1$$

We assign weights $\alpha = 0.4$, $\beta = 0.3$, and $\gamma = 0.3$ based on preliminary empirical observations that samples with high cyclomatic complexity consistently produce more model errors in both directive placement and content generation. Using the resulting difficulty score D, we stratify training instances into three equal-width difficulty tiers:

- **Easy** ($D < 0.33$): Low branching and few insertions.
- **Medium** ($0.33 \leq D < 0.66$): Moderately complex with some nested constructs or multiple directives.
- **Hard** ($D \geq 0.66$): Complex control flow, large scope, or dense completion patterns.

Table III reports the number of samples categorized into each difficulty tier across the train, validation, and test subsets.

2) Multi-Stage Task Decomposition: We decompose FPCC into three progressive subtasks, culminating in integrated end-to-end prediction, with each stage introducing incremental objectives guided by our difficulty-aware scheduling strategy:

- **Stage I: Identifying Insertion Points.** The model identifies the set of code locations where parallel directive codes should be inserted, given a sequential function C:
$$C \longrightarrow \{L_i\}_{i=1}^{k}$$
- **Stage II: Selecting Parallel Frameworks.** For each identified insertion point, the model selects the appropriate parallel framework (e.g., OpenMP, MPI, CUDA), producing location-framework pairs:
$$(C, \{L_i\}) \longrightarrow \{(L_i, F_i)\}_{i=1}^{k}$$
- **Stage III: Completing Parallel Directive Code.** Given the code and the selected framework for each location, the model completes valid parallel directive codes, completing each triplet:
$$(C, \{(L_i, F_i)\}) \longrightarrow \{(L_i, F_i, D_i)\}_{i=1}^{k}$$
- **Stage IV: End-to-End Completion.** The model jointly predicts all directive triplets directly from the input function without intermediate supervision:
$$C \longrightarrow \{(L_i, F_i, D_i)\}_{i=1}^{k}$$

3) Replay-Based Warm-Up: We implement interleaved replay across training stages, incorporating examples from previous stages to prevent catastrophic forgetting while building new capabilities. This replay mechanism aligns with our difficulty curriculum, which introduces simpler examples first before progressing to complex patterns with nested control flows or sophisticated directives.

This approach both stabilizes training by preserving earlier competencies and creates alignment between upstream tasks (identifying insertion points) and downstream objectives (generating complete directives), enabling incremental refinement of parallel reasoning abilities.

TABLE III
Distribution of samples by difficulty level across dataset splits

Difficulty Level	Train	Validation	Test	Total
Easy	3,510	441	432	4,383
Medium	5,999	786	751	7,536
Hard	3,801	437	481	4,719
Total	13,310	1,664	1,664	16,638

B. Results Analysis

We evaluate the effectiveness of HPCL strategy against two baselines: (1) three-shot prompting, and (2) standard supervised fine-tuning (SFT). The SFT baseline trains models on the transform dataset without curriculum scheduling or task decomposition. Table IV presents results across four representative LLMs.

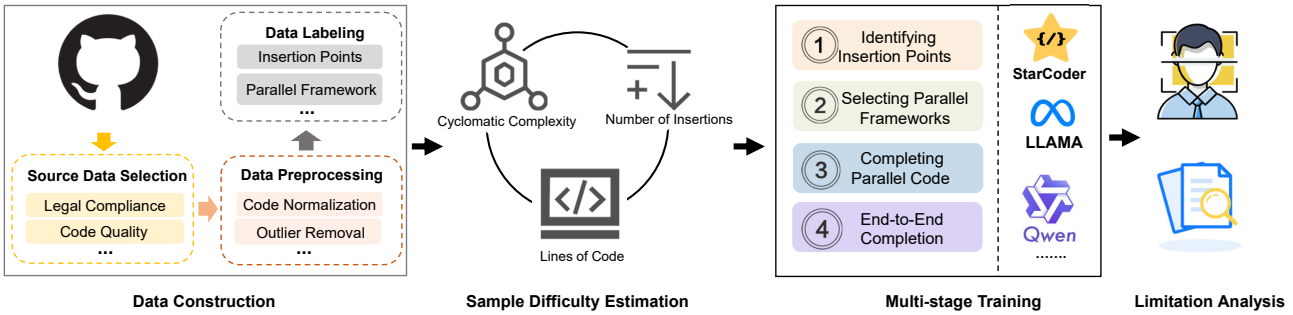

Fig. 2. Overview of our framework for structured parallel directive code completion.

TABLE IV
PERFORMANCE OF VARIOUS KNOWLEDGE INTEGRATION STRATEGIES

Model	EM	STF	IP	FW	DIR
Llama3.1-8B	0.09	0.06	2.35	25.38	1.32
+SFT	36.85	36.72	49.32	98.81	40.35
+HPCL	43.21	43.05	54.80	98.79	43.88
StarCoder2-7B	0.00	0.00	0.21	0.50	0.11
+SFT	31.05	30.95	43.19	98.68	38.08
+HPCL	42.17	42.05	53.47	98.77	43.48
CodeLlama-7B	0.00	0.00	0.30	0.52	0.13
+SFT	38.32	38.44	50.08	98.87	41.74
+HPCL	44.72	44.64	56.15	98.86	44.12
Qwen2.5-Coder-7B-Instruct	0.30	0.19	4.93	70.32	5.54
+SFT	39.16	39.08	50.09	99.05	42.14
+HPCL	**46.14**	**46.00**	**56.75**	**99.33**	**44.43**

1) Three-shot Prompting vs. SFT: Three-shot performance remains consistently poor across all models, with GPT-4 and Qwen2.5-Coder-7B-Instruct achieving less than 2% on EM. This highlights the substantial gap between general pretraining and the specialized framework understanding required for FPCC—knowledge not easily acquired through in-context learning.

SFT significantly outperforms prompting, confirming that explicit training is essential for developing directive placement and completion capabilities, thus justifying our focus on fine-tuning approaches.

2) HPCL vs. SFT: HPCL consistently outperforms SFT across all models and metrics. For instance, Qwen2.5-Coder-7B-Instruct improves EM by 17.82% and DIR by 5.43%, while LLaMA3.1-8B achieves 17.26% and 8.75% gains, respectively. These results validate our approach's core components: multi-stage task decomposition and difficulty-aware curriculum scheduling. The most significant improvements appear in structure-sensitive subtasks, with Qwen2.5-Coder-7B-Instruct gaining 13.30% in IP and LLaMA3.1-8B showing a 11.11% increase in DIR. This demonstrates how progressive subtask learning enhances compositional reasoning capabilities.

In contrast, FW remains consistently high (>98%) under both approaches, suggesting this subtask is relatively straightforward and often solvable through surface-level code patterns like loop headers or API signatures.

3) Model-wise Analysis: Architecture vs. Curriculum Benefits: HPCL benefits vary across model architectures. Smaller models show the largest gains in directive completion (e.g., StarCoder2-7B +14.18% DIR), likely due to limited structural priors from pretraining. Conversely, larger models achieve greater improvements in structural tasks like insertion point identification.

These patterns indicate HPCL effectively enhances directive generation in instruction-oriented models while improving control-flow reasoning. The consistent improvements across diverse models demonstrate HPCL's architecture-agnostic transferability, making it an effective fine-tuning strategy regardless of model scale or specialization.

TABLE V
ABLATION STUDY OF HPCL COMPONENTS

Approach	EM	STF	IP	FW	DIR
SFT	39.16	39.08	50.09	99.05	42.14
Curriculum Only	41.34	41.13	51.54	99.18	43.06
Multi-Stage Only	43.55	43.31	54.52	99.06	43.73
HPCL	**46.14**	**46.00**	**56.75**	**99.33**	**44.43**

4) Effect of Curriculum Scheduling and Task Decomposition: To quantify the individual contributions of each component, we conduct ablation experiments on Qwen2.5-Coder-7B-Instruct in Table V comparing four configurations: (i) SFT, (ii) SFT + difficulty scheduling (Curriculum-Only), (iii) SFT + task decomposition (Multi-Stage-Only), and (iv) the full HPCL framework.

Individual components each provide substantial improvements. Curriculum-Only improves EM by 5.57% (from 39.16 to 41.34) and IP by 2.89%, demonstrating how progressive difficulty improves structural generalization. Multi-Stage-Only increases EM by 11.21% and IP by 8.84%, validating the effectiveness of incremental task formulation.

Combining both strategies in HPCL leads to the highest performance: EM improves by 17.82% over SFT, IP by 13.30%, and DIR by 5.43%. In contrast, FW remains consistently high ($\geq$99%) across all variants, suggesting this subtask requires minimal structural guidance.

These results confirm the complementary nature of HPCL's dual curriculum dimensions, whose integration produces ad-

ditive gains on structure-sensitive metrics, validating our hierarchical approach to FPCC supervision.

5) Performance Analysis by Difficulty Level: To assess robustness across complexity levels, we evaluate HPCL-tuned Qwen2.5-Coder-7B-Instruct on easy, medium, and hard test subsets. As shown in Table VI, performance degrades significantly with increasing difficulty: EM drops from 59.94% to 31.24%, and DIR decreases from 62.98% to 28.76%. These patterns highlight current LLMs' limitations in handling structurally complex code regions. Notably, FW maintains high accuracy (>98%) even for difficult samples, confirming that directive generation remains the primary challenge in complex scenarios.

TABLE VI
PERFORMANCE OF HPCL ACROSS DIFFICULTY LEVELS

Difficulty Level	EM	STF	IP	FW	DIR
Easy	59.94	59.88	74.16	99.69	62.98
Medium	48.97	48.69	58.94	98.93	44.10
Hard	31.24	31.16	39.70	99.31	28.76

6) Performance by Parallel Framework: Table VII presents framework-specific results for HPCL-tuned Qwen2.5-Coder-7B-Instruct. While FW remains consistently high across all frameworks (>98%), EM and DIR metrics vary substantially.

SYCL achieves the highest EM (61.84%) and DIR (60.91%) despite its limited sample size (218 samples). This may be attributed to its concise, regular directive structure and strong syntactic alignment with OpenMP, enabling effective cross-framework knowledge transfer.

MPI and OpenMP show lower DIR scores (26.26% and 54.87%) despite larger training samples. MPI's procedural communication model with paired operations (e.g., 'MPI_Send'/'Recv') challenges static reasoning, while OpenMP's clause-level variability complicates directive completion.

CUDA and OpenCL perform competitively, benefiting from declarative APIs and broader representation in open-source repositories. TBB shows the weakest results (EM is 40.26% and DIR is 41.41%), likely due to its template-heavy, function-style approach that diverges from pragma-based patterns.

These findings suggest FPCC performance depends not only on training frequency but also on directive regularity, syntactic clarity, and structural alignment with familiar patterns.

TABLE VII
PERFORMANCE OF HPCL ACROSS PARALLEL FRAMEWORKS LEVELS

Framework	EM	STF	IP	FW	DIR
CUDA	52.66	52.67	63.81	99.72	49.75
MPI	46.93	46.69	54.97	99.25	26.26
OpenMP	36.36	36.29	49.30	99.19	54.87
OpenCL	47.30	47.41	55.03	99.10	49.59
TBB	40.26	39.67	53.80	99.15	41.41
SYCL	61.84	59.73	70.78	100.00	60.91

C. Limitation Analysis

To better understand the limitations of current LLMs on FPCC, we conducted a structured expert analysis on 200

```
/* Original code */
void SimUpdate(int iter, struct SimState state) {
    if (iter % 100 == 0) {
        LBMCollideAndPropagate<<<grid, BLOCK_SIZE, BLOCK_SIZE*6*sizeof(float)>>
>(state.dmap, state.d1, state.d2, state.drho, state.dvx, state.dvy);
        cudaMemcpy(state.vx, state.dvx, size_f, cudaMemcpyDeviceToHost);
        cudaMemcpy(state.vy, state.dvy, size_f, cudaMemcpyDeviceToHost);
        cudaMemcpy(state.rho, state.drho, size_f, cudaMemcpyDeviceToHost);
    } else {
        ... ...
    }
}
/* LLM-generated code */
void SimUpdate(int iter, struct SimState state) {
    if (iter % 100 == 0) {
        LBMCollideAndPropagate<<<grid, BLOCK_SIZE, BLOCK_SIZE*6*sizeof(float)>>
>(state.dmap, state.d1, state.d2, state.drho, state.dvx, state.dvy);
    } else {
        ... ...
    }
    cudaDeviceSynchronize();
}
```

(a) Invalid Synchronization Usage in CUDA

```
/* Original code */
static int crypt_all(int *pcount, struct db_salt *salt)
{
    ... ...
#ifdef _OPENMP
    #pragma omp parallel for default(none) private(i) shared(
ctx_salt, count, saved_key, saved_key_len, crypt_key, cp)
#endif
    for (i = 0; i < count; i++) {
        SHA256_CTX ctx;
        memcpy(&ctx, &ctx_salt, sizeof(ctx));
        SHA256_Update(&ctx, saved_key[i], saved_key_len[i]);
        SHA256_Update(&ctx, cp, FORTINET_MAGIC_LENGTH);
        SHA256_Final((unsigned char*)crypt_key[i], &ctx);
    }
    return count;
}
/* LLM-generated code */
static int crypt_all(int *pcount, struct db_salt *salt)
{
    ... ...
#ifdef _OPENMP
    #pragma omp parallel for
#endif
    for (i = 0; i < count; i++) {
        SHA256_CTX ctx;
        memcpy(&ctx, &ctx_salt, sizeof(ctx));
        SHA256_Update(&ctx, saved_key[i], saved_key_len[i]);
        SHA256_Update(&ctx, cp, FORTINET_MAGIC_LENGTH);
        SHA256_Final((unsigned char*)crypt_key[i], &ctx);
    }
    return count;
}
```

(b) Missing Clause in OpenMP

```
/* Original code */
void ADIOI_GEN_IreadStrided(ADIO_File fd, void *buf, int count,
                            MPI_Datatype datatype, int file_ptr_type,
                            ADIO_Offset offset, ADIO_Request * request, int *error_code)
{
    MPI_Count typesize;
    ... ...
    if (*error_code == MPI_SUCCESS) {
        MPI_Type_size_x(datatype, &typesize);
        nbytes = (MPI_Offset)count * (MPI_Offset)typesize;
    }
    MPIO_Completed_request_create(&fd, nbytes, error_code, request);
}
/* LLM-generated code */
void ADIOI_GEN_IreadStrided(ADIO_File fd, void *buf, int count,
                            MPI_Datatype datatype, int file_ptr_type,
                            ADIO_Offset offset, ADIO_Request * request, int *error_code)
{
    MPI_Count typesize;
    ... ...
    if (*error_code == MPI_SUCCESS) {
        MPI_Get_count(&status, datatype, &typesize);)
        nbytes = (MPI_Offset)count * (MPI_Offset)typesize;
    }
    MPIO_Completed_request_create(&fd, nbytes, error_code, request);
}
```

(c) Wrong Directive Type in MPI

Fig. 3. Error patterns in LLM-completed parallel directive code.

TABLE VIII
ERROR CATEGORIZATION IN 200 MODEL-COMPLETED PARALLEL DIRECTIVE CODE SAMPLES

High-Level Category	Fine-Grained Error Type	Explanation	Proportation	
			HPCT	SFT
Insertion Point Error	Misaligned Directive	Directive inserted at incorrect line, affects scope or logic.	30.19%	38.65%
	Scope Violation	Directive placed outside its intended loop or region.	6.73%	8.27%
	Execution Order Misunderstanding	Incorrect placement disrupts logical execution sequence.	4.04%	5.38%
Missing Completion	Missing Directive	Expected parallel construct is not completed.	8.85%	13.27%
	Missing Clause	Directive lacks required clause like num_threads, collapse.	1.73%	1.54%
	Missing Synchronization	Omitted necessary barrier or wait, risking race conditions.	0.38%	1.35%
	Missing Finalization	Memory/resource not released (e.g., MPI_Finalize).	0.38%	1.15%
Redundant Completion	Redundant Directive	Inserted directive is unnecessary given existing context.	14.42%	18.85%
	Duplicate Operation	Same memory transfer or barrier duplicated.	1.15%	3.08%
	Hallucinated Code	Model completed directive/function unrelated to context.	2.50%	5.58%
Incorrect Usage	Wrong Directive Type	Incorrect parallel directive selected (e.g., MPI_Wait vs MPI_Bcast).	9.04%	12.50%
	Wrong Framework	Used directives from the wrong parallel framework.	0.58%	2.31%
	Incorrect Variable or Buffer	Used the wrong variable in the directive or data transfer.	5.96%	10.58%
	Invalid Argument	Function call includes wrong or mismatched arguments.	2.88%	4.81%
	Incorrect Clause Specification	Clause value is invalid (e.g., collapse(-1), if()).	3.85%	5.38%
	Invalid Synchronization Usage	Synchronization APIs used in wrong order or scope.	0.77%	1.73%
Syntax Error	Malformed Directive	Directive syntax is incorrect (e.g., collapse(4,)).	4.04%	5.96%
	Invalid Clause Syntax	Clause malformed or includes illegal values.	1.73%	3.08%
	Typographical Error	Code contains undefined symbols, typos, or unclosed expressions.	0.77%	2.31%

outputs randomly sampled from our best-performing model. This evaluation focuses on persistent failure modes that remain even under favorable training conditions, revealing deeper issues in directive reasoning.

1) Methodology and Taxonomy Construction: We recruited two domain experts, each with 3–8 years of experience in parallel frameworks such as OpenMP, MPI, and CUDA. The two experts first individually labeled the samples to ensure independent judgments and then collaboratively resolved disagreements through discussion, ensuring consensus for each error analysis. To avoid biases introduced by pre-defined categories, we adopted a bottom-up analysis approach, where experts freely identified and recorded observed errors. To mitigate fatigue and maintain analysis quality, each expert worked in multiple short sessions (30 samples per session).

We aggregated the resulting error analysis and synthesized them into a hierarchical taxonomy of FPCC-specific failure types. As summarized in Table VIII, the taxonomy comprises five high-level categories and 20 fine-grained subtypes. It spans both surface-level syntax issues (e.g., malformed clauses, typographical errors) and deeper semantic challenges, such as misaligned directive placement, incomplete reductions, and invalid synchronization logic.

This taxonomy highlights key obstacles that current LLMs fail to overcome, including directive alignment, clause generation, and execution semantics. Compared to prior studies that focus primarily on token-level accuracy [31]–[33], [38], our error schema provides a more actionable lens for understanding failure modes specific to structured parallelization tasks.

2) Comparative Failure Distribution: We further compare HPCL and SFT from the perspective of failure distribution using our expert-labeled taxonomy. Key findings include:

- **Fewer structural alignment errors.** *Insertion Point Errors* remain the most dominant failure mode, but misaligned directives decrease markedly (38.65% $\rightarrow$ 30.19%), indicating stronger control-flow awareness.
- **Reduced incompleteness and redundancy.** HPCL cuts *Missing Completion* errors nearly in half (13.27% $\rightarrow$ 8.85%), while *Redundant Completion* also drops (18.85% $\rightarrow$ 14.42%), reflecting better balance between under- and over-generation.
- **Improved semantic and API usage.** Within *Incorrect Usage*, HPCL lowers buffer or variable misuse (10.58% $\rightarrow$ 5.96%) and wrong directive type selection (12.50% $\rightarrow$ 9.04%), showing stronger understanding of framework-specific semantics.
- **Suppressed syntax-level failures.** *Syntax Errors* decline consistently (5.96% $\rightarrow$ 4.04%), suggesting improved stability in clause formation and directive syntax.

Overall, compared to SFT, HPCL not only improves aggregate accuracy but also reduces critical structural, semantic, and syntactic errors. These findings demonstrate that curriculum-guided training can foster more reliable directive reasoning.

3) Representative Failure Cases: We further highlight three representative cases that illustrate common failure patterns beyond surface-level directive formatting:

a) Case 1: Invalid Synchronization Usage (Fig. 3a): In this example, the model replaces a series of `cudaMemcpy` calls with a single `cudaDeviceSynchronize()` command. While `cudaDeviceSynchronize()` guarantees kernel completion, it critically fails to perform the necessary data transfer from the GPU back to the host. The model's substitution of a synchronization primitive for a memory transfer directive demonstrates a lack of semantic understanding regarding the host-device memory model. The generated code is functionally incorrect, as the host never receives the computed results, fundamentally compromising program integrity.

b) Case 2: Missing Clause (Fig. 3b): This case reveals that the model identifies a loop suitable for OpenMP parallelization but emits only the base directive `#pragma omp parallel for`, omitting clauses like `private(i)` and `default(none)`. These clauses are essential for data scoping and thread safety. The failure indicates an incomplete understanding of OpenMP's clause semantics.

c) Case 3: Wrong Directive Type (Fig. 3c): In this example, the model misuses a semantically incompatible MPI function. The original code correctly calls `MPI_Type_size_x` to determine the size (in bytes) of a datatype for calculating the number of bytes transferred. However, the model erroneously replaces this with `MPI_Get_count`, which is only valid when applied to a completed communication. It highlights the model's limitation in distinguishing between semantically similar MPI functions based on their intended contexts.

VI. Discussion

A. Evaluation on Dynamic Metrics

Static metrics such as EM, DIR, and F1 primarily assess the similarity between generated outputs and reference labels. While they provide useful fine-grained signals, they cannot fully capture the executability and reliability of generated code in real development environments. To address this limitation, we introduce dynamic metrics that directly evaluate whether generated programs (i) compile successfully and (ii) produce correct outputs when executed against project-provided test suites.

However, running dynamic evaluation on the entire corpus is infeasible, as many HPC functions rely on complex build systems and external dependencies. We therefore curate a representative subset that balances framework diversity and code complexity, while ensuring that each case is equipped with reliable test harnesses. The final subset consists of 356 functions (21.38% of the test set) drawn from diverse open-source projects. To ensure reproducibility and prevent cross-project interference, each repository is encapsulated in an independent Docker image with the required toolchains.

Results. Table IX summarizes compilation success and test correctness across four representative models. HPCL consistently outperforms SFT on both metrics. For example, on Qwen2.5-Coder-7B-Instruct, compilation success improves from 60.96% to 66.57%, while correctness rises from 36.24% to 45.22%. Similar gains are observed for Llama3.1-8B, StarCoder2-7B, and CodeLlama-7B. Building on the above results, we find that HPCL can still achieve superior performance on dynamic metrics, demonstrating the effectiveness of our proposed approach. Our findings also indicate that HPCL poses greater challenges compared to traditional code generation tasks. Therefore, future research on dataset construction should take into account repository-level information and more comprehensive test cases.

TABLE IX
LLM Performance under SFT vs. HPCL in terms of Dynamic Metrics

Model	Approach	Compilation Success (%)	Test Correctness (%)
Llama3.1-8B	+SFT	56.74%	34.83%
	+HPCL	58.71%	41.01%
StarCoder2-7B	+SFT	53.09%	28.93%
	+HPCL	56.46%	37.92%
CodeLlama-7B	+SFT	57.02%	35.39%
	+HPCL	62.92%	41.85%
Qwen2.5-Coder-7B-Instruct	+SFT	60.96%	36.24%
	+HPCL	66.57%	45.22%

TABLE X
Performance of Qwen2.5-Coder-7B-Instruct and HPCL on single-framework and multi-framework functions

Approach	Sample Type	EM	SFT	IP	FW	DIR
Qwen2.5-Coder-7B-Instruct	Single	0.30	0.19	4.93	70.32	5.54
	Multi	0	0	0.34	38.56	0.76
+SFT	Single	39.16	39.08	50.09	99.05	42.14
	Multi	7.32	7.56	15.21	95.32	4.23
+HPCL	Single	46.14	46.00	56.75	99.33	44.43
	Multi	9.94	9.68	19.42	96.81	7.79

B. Evaluation on Multi-Framework Scenarios

In our dataset, although only about 1% of functions involve mixing multiple frameworks (e.g., MPI with CUDA), these cases represent realistic and particularly challenging scenarios in the HPCL task.

Experimental results. As shown in Table X, multi-framework scenarios are consistently more challenging than single-framework ones. The baseline model almost fails to generate correct code (IP 0.34%, DIR 0.76%). While SFT brings certain improvements (DIR 4.23%), HPCL achieves further gains (DIR 7.79%). Nevertheless, its performance remains far below that of single-framework cases (DIR 44.43%), further highlighting the research challenges posed by this scenario.

Case study. Figure 4 illustrates a mixed MPI–CUDA function where omitting `MPI_Comm_size` leaves `num_processes` undefined, triggering invalid CUDA device binding, divide-by-zero partitioning, and MPI deadlocks. This indicates that a coding error in one framework can affect the correct generation of code in other frameworks

Challenges and implications. Multi-framework scenarios pose three main challenges: (i) *semantic alignment* across processes, devices, and threads; (ii) *execution-model conflicts* from heterogeneous synchronization semantics; and (iii) *error propagation* across frameworks. While HPCL improves robustness, reliable cross-framework reasoning remains an open challenge, motivating future research on expanding hybrid-function coverage and integrating cross-framework semantic constraints into training.

```
/* Original code */
MultiGpuConfig multi_gpu_config_init(int *argc, char ***argv) {
#ifdef MULTI_GPU
    // Initialize MPI
    MultiGpuConfig result;

    mpiCheck(MPI_Init(argc, argv));
    mpiCheck(MPI_Comm_rank(MPI_COMM_WORLD, &result.process_rank));
    mpiCheck(MPI_Comm_size(MPI_COMM_WORLD, &result.num_processes));

    // Allocate local GPU device and set the current device
    result.local_device_idx = multi_gpu_get_local_device_idx(result(result.proc
ess_rank, result.num_processes);
    cudaCheck(cudaSetDevice(result.local_device_idx));
    … …
}
/* LLM-generated code */
MultiGpuConfig multi_gpu_config_init(int *argc, char ***argv) {
#ifdef MULTI_GPU
    // Initialize MPI
    MultiGpuConfig result;

    mpiCheck(MPI_Init(argc, argv));
    mpiCheck(MPI_Comm_rank(MPI_COMM_WORLD, &result.process_rank));

    // Allocate local GPU device and set the current device
    result.local_device_idx = multi_gpu_get_local_device_idx(result(result.proc
ess_rank, result.num_processes);
    cudaCheck(cudaSetDevice(result.local_device_idx));
    … …
}
```

Fig. 4. Multi-framework failure case (MPI + CUDA) caused by missing MPI_Comm_size

VII. Threats to Validity

We identify and discuss several potential threats to the validity of our study as follows:

Internal Validity. A primary threat to validity lies in the accuracy of dataset labels, including insertion points, framework labels, and parallel directive codes. Although we apply AST-based static analysis and structured diffing to ensure alignment, edge cases may persist. Future work may incorporate automated consistency checks (e.g., AST round-tripping), ensemble validation across static analyzers, and noise-robust training objectives to further improve label quality.

External Validity. The generalizability of our findings may be influenced by dataset construction. For non-OpenMP frameworks such as CUDA and MPI, building functionally aligned serial–parallel pairs is particularly difficult, since publicly available codebases rarely include both versions. Consistent with prior work (e.g., OMPGPT [33], MPIrigen [31]), we adopt a practical and widely accepted compromise: systematically removing framework-specific statements to derive serial code. We acknowledge that it may underestimate the challenges of real-world development; however, it ensures reproducibility and scalability across large corpora.

Construct Validity. Our evaluation metrics are designed to assess the model's ability to identify insertion points, select parallel frameworks, and complete parallel directive codes. However, these metrics focus primarily on static code fidelity. They do not fully capture deeper correctness properties of parallel programs, such as race freedom, synchronization safety, or execution efficiency. To mitigate this limitation, we supplement quantitative metrics with a structured human analysis that qualitatively examines directive code semantics and alignment.

Conclusion Validity. Our limitation analysis qualitatively validates model failure modes using a structured error taxonomy. Two experts independently annotated outputs and reconciled disagreements, achieving high internal agreement. However, the scope is limited to 200 samples and two evaluators, we acknowledge that broader validation across more samples and evaluator backgrounds could further support generalizability.

VIII. Related Work

In this section, we review related research on parallel programs, focusing on two main aspects: testing and automatic code generation.

Testing and debugging of parallel programs. Detecting concurrency-related defects such as data races, deadlocks, and atomicity violations has been a longstanding challenge. Dynamic tools like Eraser [52] detect race conditions via lockset-based monitoring, while static analyzers such as RacerX [53] and LockSmith [54] reason about synchronization correctness using control-flow and alias analysis. ISP [55] analyzes MPI protocol violations in message-passing programs. Debugging support via deterministic replay [56] further enables reproducible error diagnosis in multi-threaded settings.

Automatic parallel directive code generation. Traditional compiler-based tools such as PLUTO [57] and Cetus [14] rely on static rule-based analysis, primarily targeting loop-level parallelization in fully compilable code. While effective for structured programs, they often fail to generalize to unstructured control flow and hybrid-parallel scenarios. Recently, LLM-based approaches have shown that models can learn to synthesize parallel directives from data, but they typically focus on single frameworks and isolated loops [31], [32], [38], [58]. MonoCoder [59] explores HPC-oriented pretraining, evaluating models through code completion tasks. However, its focus is on syntactic completion of partially parallel code, rather than transforming purely sequential code into parallel forms.

In contrast, we introduce FPCC as a structured completion task that inserts parallel directives into fully sequential functions. FPCC demands deep reasoning over control flow, data dependencies, and framework semantics, reflecting the incremental nature of real-world parallelization. Our work is the first to support multi-framework directive completion using a dedicated dataset and a hierarchical curriculum-based training strategy. Unlike traditional compilers that require compilable input and offer static guarantees, our approach supports directive synthesis in unstructured or hybrid code scenarios frequently encountered in real-world development. We advocate for future hybrid systems that combine LLM flexibility with compiler soundness to advance automated parallelization.

IX. Conclusion

This paper investigates large language models for framework-based parallel code completion (FPCC), which involves identifying insertion points, selecting frameworks, and completing directive code. We build a dataset across six mainstream frameworks and propose HPCL, a curriculum-based training strategy that progressively enhances FPCC sub-skills. Experiments show consistent performance gains, while case analysis highlights challenges in directive alignment and semantic reasoning. Our work lays a foundation for more robust LLMs in parallel code generation, with future directions including partial-code scenarios and larger, manually curated serial–parallel datasets for realistic HPC evaluation.

Acknowledgment

We would like to thank anonymous reviewers for their insightful and constructive feedback. This work is supported in part by the National Key Research and Development Program of China (Grant Nos. 2023YFA1011704, 2021YFB0300101).

References

[1] Y. Su, J. Zhou, J. Ying, M. Zhou, and B. Zhou, "Computing infrastructure construction and optimization for high-performance computing and artificial intelligence," *CCF Transactions on High Performance Computing*, pp. 1–13, 2021.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *ACM SigPlan Notices*, vol. 30, no. 8, pp. 207–216, 1995.

[3] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[4] N. M. Josuttis, "The c++ standard library: a tutorial and reference," 2012.

[5] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in intel threading building blocks." *Intel Technology Journal*, vol. 11, no. 4, 2007.

[6] R. Xu, L. Chen, R. Zhang, Y. Zhang, W. Xiao, H. Zhou, and X. Mao, "Accelerating static null pointer dereference detection with parallel computing," in *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, 2024, pp. 135–144.

[7] J. Fang, C. Huang, T. Tang, and Z. Wang, "Parallel programming models for heterogeneous many-cores: A survey," *arXiv preprint arXiv:2005.04094*, 2020.

[8] T. Tang, Y. Lin, and X. Ren, "Mapping openmp concepts to the stream programming model," in *2010 5th International Conference on Computer Science & Education*. IEEE, 2010, pp. 1900–1905.

[9] A. Marowka, "Pitfalls and issues of manycore programming," in *Advances in Computers*. Elsevier, 2010, vol. 79, pp. 71–117.

[10] J. Chen, J. Impagliazzo, and L. Shen, "High-performance computing and engineering educational development and practice," in *2020 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2020, pp. 1–8.

[11] GNU, "Gcc, the gnu compiler collection," https://gcc.gnu.org.

[12] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[13] M. Dever, "Autopar: automating the parallelization of functional programs," Ph.D. dissertation, Dublin City University, 2015.

[14] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[15] S. Prema, R. Nasre, R. Jehadeesan, and B. Panigrahi, "A study on popular auto-parallelization frameworks," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 17, p. e5168, 2019.

[16] S. Prema, R. Jehadeesan, and B. Panigrahi, "Identifying pitfalls in automatic parallelization of nas parallel benchmarks," in *2017 National Conference on Parallel Computing Technologies (PARCOMPTECH)*. IEEE, 2017, pp. 1–6.

[17] R. Harel, I. Mosseri, H. Levin, L.-o. Alon, M. Rusanovsky, and G. Oren, "Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: analysis, pitfalls, enhancement and potential," *International Journal of Parallel Programming*, vol. 48, pp. 1–31, 2020.

[18] R. Milewicz, P. Pirkelbauer, P. Soundararajan, H. Ahmed, and T. Skjellum, "Negative perceptions about the applicability of source-to-source compilers in hpc: A literature review," in *High Performance Computing: ISC High Performance Digital 2021 International Workshops, Frankfurt am Main, Germany, June 24–July 2, 2021, Revised Selected Papers 36*. Springer, 2021, pp. 233–246.

[19] S. P. Johnson, C. S. Ierotheou, and M. Cross, "Automatic parallel code generation for message passing on distributed memory systems," *Parallel Computing*, vol. 22, no. 2, pp. 227–258, 1996.

[20] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[21] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[22] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2332–2354, 2024.

[23] D. Huang, J. M. Zhang, Q. Bu, X. Xie, J. Chen, and H. Cui, "Bias testing and mitigation in llm-based code generation," *ACM Transactions on Software Engineering and Methodology*, 2024.

[24] L. Zhong and Z. Wang, "Can llm replace stack overflow? a study on robustness and reliability of large language model code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 19, 2024, pp. 21 841–21 849.

[25] G. Yang, Y. Zhou, X. Zhang, X. Chen, T. Han, and T. Chen, "Assessing and improving syntactic adversarial robustness of pre-trained models for code translation," *Information and Software Technology*, vol. 181, p. 107699, 2025.

[26] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Y. Zhuo, and T. Chen, "Chain-of-thought in neural code generation: From and for lightweight language models," *IEEE Transactions on Software Engineering*, 2024.

[27] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.

[28] J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 2023, pp. 284–289.

[29] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–38, 2024.

[30] J. A. Baktash and M. Dawodi, "Gpt-4: A review on advancements and opportunities in natural language processing," *arXiv preprint arXiv:2305.03195*, 2023.

[31] N. Schneider, N. Hasabnis, V. A. Vo, T. Kadosh, N. Krien, M. Capota, G. Tamir, T. L. Willke, N. Ahmed, Y. Pinter *et al.*, "Mpirigen: Mpi code generation through domain-specific language models," in *Proceedings of the 2024 Workshop on AI For Systems*, 2024, pp. 1–6.

[32] T. Kadosh, N. Hasabnis, P. Soundararajan, V. A. Vo, M. Capota, N. Ahmed, Y. Pinter, and G. Oren, "Ompar: Automatic parallelization with ai-driven source-to-source compilation," *arXiv preprint arXiv:2409.14771*, 2024.

[33] L. Chen, A. Bhattacharjee, N. Ahmed, N. Hasabnis, G. Oren, V. Vo, and A. Jannesari, "Ompgpt: A generative pre-trained transformer model for openmp," in *European Conference on Parallel Processing*. Springer, 2024, pp. 121–134.

[34] L. Chen, Q. I. Mahmud, H. Phan, N. Ahmed, and A. Jannesari, "Learning to parallelize with openmp by augmented heterogeneous ast representation," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 442–456, 2023.

[35] R. Harel, Y. Pinter, and G. Oren, "Learning to parallelize in a shared-memory environment with transformers," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 450–452.

[36] T. Kadosh, N. Schneider, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, "Advising openmp parallelization via a graph-based approach with transformers," in *International Workshop on OpenMP*. Springer, 2023, pp. 3–17.

[37] R. Harel, T. Kadosh, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, "Pragformer: Data-driven parallel source code classification with transformers," 2023.

[38] N. Schneider, T. Kadosh, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, "Mpi-rical: Data-driven mpi distributed parallelism assistance with transformers," in *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 2–10.

[39] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–23, 2013.

[40] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[41] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[42] Y. Ding, Z. Wang, W. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth *et al.*, "Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion," *Advances in Neural Information Processing Systems*, vol. 36, pp. 46 701–46 723, 2023.

[43] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "Repocoder: Repository-level code completion through iterative retrieval and generation," *arXiv preprint arXiv:2303.12570*, 2023.

[44] T. Kadosh, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, "Quantifying openmp: Statistical insights into usage and adoption," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2023, pp. 1–7.

[45] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[46] Clang, "Clang compiler user's manual," https://clang.llvm.org/docs/UsersManual.html.

[47] X. Wan, W. Wang, J. Liu, and T. Tong, "Estimating the sample mean and standard deviation from the sample size, median, range and/or interquartile range," *BMC medical research methodology*, vol. 14, pp. 1–13, 2014.
[48] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.
[49] "gpt-4.0-turbo," https://platform.openai.com/docs/models/gpt-4o, 2024.
[50] Meta, "Llama-3.1-8b-instruct," https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct, 2024.
[51] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.
[52] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.
[53] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 237–252, 2003.
[54] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: context-sensitive correlation analysis for race detection," *Acm Sigplan Notices*, vol. 41, no. 6, pp. 320–331, 2006.
[55] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "Isp: a tool for model checking mpi programs," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 285–286.
[56] G. Altekar and I. Stoica, "Odr: Output-deterministic replay for multicore debugging," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 193–206.
[57] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "Pluto: A practical and fully automatic polyhedral program optimization system," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.
[58] L. Chen, P.-H. Lin, T. Vanderbruggen, C. Liao, M. Emani, and B. De Supinski, "Lm4hpc: Towards effective language model application in high-performance computing," in *International Workshop on OpenMP*. Springer, 2023, pp. 18–33.
[59] T. Kadosh, N. Hasabnis, V. A. Vo, N. Schneider, N. Krien, M. Capotă, A. Wasay, G. Tamir, T. Willke, N. Ahmed *et al.*, "Monocoder: Domain-specific code language model for hpc codes and tasks," in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2024, pp. 1–7.