CurParCL: A Curriculum-Guided Dual-Branch Graph–Text Contrastive Learning Framework for Parallelism Detection

Boxun Li, Pinle Qin, Jianchao Zeng, Yuanyuan Shen*
School of Computer Science and Technology, North University of China, Taiyuan, China
Email: s202407005@st.nuc.edu.cn, qpl@nuc.edu.cn, zjc@nuc.edu.cn, 20230090@nuc.edu.cn

***Abstract*—With the continued advancement of high-performance computing (HPC) platforms, automatic parallelization has emerged as a key enabler for high-performance software development. In the automatic-parallelization pipeline, parallelism detection is a key step in identifying candidate regions for parallel execution. Nevertheless, existing methods are limited in their ability to model structured program semantics, accurately identify complex dependencies, and generalize across datasets. To overcome these limitations, we propose CurParCL, a curriculum-guided dual-branch graph-text contrastive learning framework. Specifically, the framework introduces lightweight graph neural network to capture edge directionality and performs adaptive structural aggregation over data-flow graphs(DFGs). It further integrates difficulty-aware strategies into contrastive learning, improving label discriminability and training stability via adaptive scheduling. Moreover, we design a dual-branch graph–text contrastive mechanism that jointly encodes DFGs and source-code sequences. This design aligns sequential semantics with structural dependencies in a unified embedding space, enabling collaborative multimodal representation learning. Experiments show that CurParCL consistently outperforms baselines on multiple parallelism detection benchmarks and remains robust under cross-dataset evaluation, suggesting good practical applicability for automated parallelization.**

***Index Terms*—parallelism detection, contrastive learning, curriculum learning, OpenMP, shared-memory parallelism**

I. Introduction

As multi-core processors and heterogeneous accelerators become ubiquitous, high-performance applications face growing demands for computational throughput and energy efficiency. However, transforming sequential programs into correct and efficient parallel implementations remains challenging. Developers must reason about data dependences, race conditions, and load balancing, while selecting suitable forms of parallelism for CPUs and GPUs. Since many programs are not inherently parallelizable, manual parallelization requires substantial expertise and is often time-consuming and error-prone. Consequently, automatically identifying candidate parallel regions while preserving correctness has become a major challenge in high-performance computing.

Traditional automatic parallelization mainly relies on static and dynamic analyses. Polyhedral frameworks and source-to-source compilers can insert OpenMP pragmas for regular loops with affine memory accesses, while dynamic profiling tools partially alleviate the conservatism of static analysis through runtime dependence detection [1]. However, these approaches depend heavily on manually designed features and heuristic rules. Their representations, often based on linearized code, ASTs, or simplified dependence graphs, have limited ability to capture long-range dependences and complex semantic interactions. As a result, they may miss parallelization opportunities or make unreliable decisions in difficult cases.

Recent advances in data-driven program analysis provide new opportunities. Graph-based representations and graph neural networks (GNNs) can better model code structure and semantic relations [2]–[5]. Meanwhile, contrastive learning improves robustness by aligning semantically related views, and curriculum learning can stabilize optimization by organizing training from easier to harder samples [6]–[9]. Nevertheless, existing learning-based parallelism detection methods still struggle with heterogeneous coding styles, sparse supervision, and complex dependence interactions.

Building on the above observations, this study proposes a curriculum-guided dual-branch graph-text contrastive learning framework for parallelism detection. We introduce a lightweight, structure-aware graph network that performs adaptive structural aggregation over data-flow-based program graphs, emphasizing critical variables and dependence paths with minimal additional inference overhead. We then extend contrastive learning from coarse-grained discrimination of parallelizability to fine-grained representation learning for evaluating candidate parallelization schemes, via task-aware construction of positive/negative pairs and curriculum-based difficulty scheduling. Finally, the dual-branch graph-text design jointly encodes code sequences and DFGs, aligning sequential semantics with structural dependences in a unified embedding space to improve parallelism detection accuracy and robustness across coding styles and datasets.

The contributions of this study are as follows:

- We design a lightweight, structure-aware graph network that performs one-pass message passing and "gate-and-hub" pooling over data-flow graphs (DFGs), while jointly encoding code sequences and DFGs in a unified representation space. This design explicitly emphasizes critical dependence paths and pivotal variables, improving sensitivity to complex parallel patterns and rich structural semantics while keeping model size and inference

*Corresponding author.

overhead manageable.

- We further propose a difficulty-adaptive contrastive learning mechanism. The mechanism introduces dynamic upper bounds on similarity and difficulty-tiered sample scheduling, transforming contrastive learning from naive similarity optimization into a staged, difficulty-graded training process. As a result, the model achieves smoother convergence and stronger discriminative capability under complex code semantics.
- We develop a dual-branch collaborative contrastive framework for code sequences and DFGs that captures sequential semantics and structural dependencies simultaneously. Cross-modal objectives align these modalities in a shared representation space. While preserving overall controllability, the framework naturally incorporates structural-consistency regularization, thereby improving generalization and robustness under complex dependence patterns and across diverse codebases.

II. Related Work

Research on automatic parallelization mainly follows two directions: dependence-analysis-based methods and learning-based methods. This section reviews representative studies with a focus on *parallelism detection* rather than full pragma generation.

A. Parallelism Detection Tools Driven by Dependence Analysis

Early automatic parallelization mainly relied on static compilation analysis and later hybrid static–dynamic methods. Static approaches, often based on polyhedral models and pointer/alias analysis, can reason formally about regular loop nests but are usually conservative for complex control flow and non-affine memory accesses. Representative systems such as Polaris [10], Pluto [11], Cetus [12], and Par4All [13] analyze program structure and automatically insert OpenMP pragmas. Although effective for regular kernels, they often miss opportunities in the presence of pointer aliasing, irregular accesses, and complex control flow.

Dynamic and hybrid approaches extend coverage by observing runtime dependences. DiscoPoP [1], for example, identifies candidate parallel regions through runtime monitoring, while Morew *et al.* [14] combine static analysis with profiling to refine decisions. However, these methods may introduce non-trivial runtime overhead and can remain sensitive to input distributions [15], [16].

B. Parallelism Detection Tools Driven by Machine Learning

Learning-based methods aim to infer parallelism-related patterns from data, reducing reliance on hand-crafted rules. Early studies explored feature-based models for program mapping on heterogeneous platforms [17], while more recent work investigates directive inference and adaptation across platforms [18].

Several studies focus more directly on *parallelism detection*. Harel *et al.* use transformer-based models for shared-memory parallelization decisions [19]. Shen *et al.* propose a GNN-based framework using contextual flow graphs for parallelism detection [5], providing a learning-based alternative to profiling-centric tools such as DiscoPoP [1]. In addition, Nichols *et al.* show that fine-tuning large language models for HPC workloads can improve OpenMP-related tasks such as pragma generation [20].

More broadly, representation learning methods provide useful foundations for code modeling. CodeBERT [21] learns joint representations of code and natural language, while UniXcoder [22] incorporates multiple modalities in a unified architecture. Contrastive pretraining with semantics-preserving transformations has also been used to improve code representations, as shown in ContraBERT [23].

A common design choice in existing pipelines is to entangle parallelism detection with downstream clause generation and/or to use coarse structural abstractions that underutilize fine-grained data-flow information during representation learning. In contrast, we treat parallelizability assessment as a distinct objective and emphasize the integration of source sequences with fine-grained data-flow structure under task-aligned representation learning, aiming to improve detection fidelity and robustness across heterogeneous code distributions.

III. METHODOLOGY

We propose CurParCL, a curriculum-guided dual-branch graph-text contrastive learning framework for parallelism detection, as illustrated in Fig. 1. The framework jointly encodes source code and its corresponding data-flow graph (DFG): the text branch captures lexical and contextual semantics, while the graph branch models variable-level data dependences. Based on GraphCodeBERT, CurParCL combines semantics-preserving data augmentation, curriculum-guided sampling, and graph-text contrastive learning to improve robustness and discriminative ability for parallelism detection.

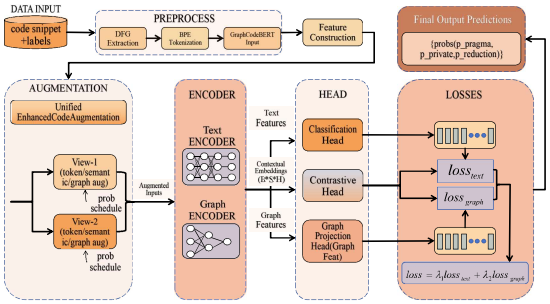
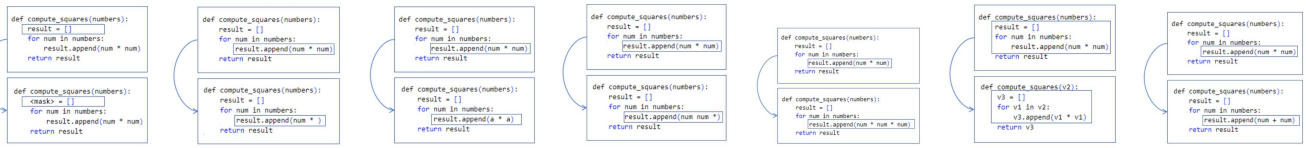


Fig. 1: Overview of the CurParCL workflow.

A. Data Augmentation

Data augmentation is a well-established technique for improving model generalization and robustness [24], yet it remains relatively underexplored in code intelligence, especially for semantics-sensitive program analysis tasks. First, source code is constrained by both syntax and semantics. Second, for fine-grained tasks such as parallelism detection, limited



(a) (b) (c) (d) (e) (f) (g)

Fig. 2: Examples of token-level (a–e) and semantic-level (f–g) data augmentation methods. For each subfigure, the upper section displays the original code, while the lower section shows the augmented code.

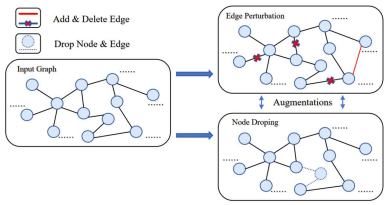


Fig. 3: Examples of graph-level data augmentation (left: original graph; right: augmented graph).

annotated data, a low proportion of parallelizable regions can cause models to fail to reliably capture dependence and structural features that are relevant to parallelism.

To address these constraints, we propose a multi-granularity code augmentation framework tailored to parallelism-related tasks. The framework systematically defines augmentation strategies along three dimensions: token level, semantic level, and graph level. While preserving program semantics, it deliberately increases the diversity of parallel training samples. This design improves robustness to semantically equivalent rewritings and mild data-flow perturbations, which is particularly beneficial in settings with sparse parallelism annotations and highly heterogeneous code distributions.

- **Token-level.** We apply surface-form edits to improve robustness to lexical variation and token-level corruption, including masking (`<mask>`), deletion, replacement, adjacent transposition, and short-span copy. To reduce syntactic breakage, destructive operations (e.g., deletion/transposition) are applied more conservatively to keywords and operators (see Fig. 2a–2e).
- **Semantic-level.** Guided by abstract syntax tree (AST) analysis, we perform semantics-preserving rewritings that maintain functional equivalence, e.g., type-compatible declaration rewriting and safe operator substitution (see Fig. 2f–2g).
- **Graph-level.** On the DFGs, we generate structure-perturbed views for the proposed structure-aware GNN by randomly dropping nodes/edges, optionally adding a small number of candidate edges, and enforcing basic connectivity of the resulting graph (see Fig. 3).

These multi-granularity augmentations provide diverse views for subsequent curriculum learning and contrastive learning. However, fixed augmentation policies alone may be insufficient to meet the differing requirements across training stages. Therefore, we further introduce a progressive curriculum learning strategy with dynamic augmentation scheduling, allowing augmentation strength and sample difficulty to evolve adaptively throughout training.

B. Progressive Curriculum Learning

Curriculum learning improves optimization by introducing training samples from easy to hard. In our setting, we use tokenized code length as a practical proxy for sample difficulty: shorter programs are generally easier to encode, whereas longer programs often contain richer structures and more complex dependence patterns.

Accordingly, we propose an adaptive curriculum learning strategy for code comprehension. By combining dynamic difficulty quantification, hierarchical sampling, and progressive enhancement mechanisms [25], the strategy helps mitigate gradient oscillations and reduces overfitting to easy samples during early training in complex parallelism detection. Concretely, we use tokenized code length as the difficulty signal and integrate time-decayed sampling with performance-adaptive adjustments to realize a smooth curriculum from simple to complex instances. Experiments show that this strategy improves convergence speed and generalization in our parallelism detection setting.

For a parallel-code program sample x_i, given a tokenizer τ, we define the token-length-based difficulty function as follows:

$$D(x_i) = \frac{\log(1 + \mathrm{Len}(\tau(x_i)))}{\max_{j=1,\ldots,N} \log(1 + \mathrm{Len}(\tau(x_j)))}, \tag{1}$$

where $\mathrm{Len}(\tau(x_i))$ denotes the token sequence after length calculation, and N is the total number of training samples. We apply a logarithmic transformation to reduce the influence of extremely long programs. We partition the overall sample difficulty into three levels. To mitigate gradient instability and training oscillations caused by hard samples in early training, we adopt a dynamic sampling scheme that progressively introduces harder samples. This strategy helps stabilize optimization in the initial stage and supports a smooth transition from easy to difficult instances. The sampling probabilities for the three levels are defined as follows:

$$P_{\text{easy}}(e) = \alpha_{\text{easy}} \cdot \exp\left(-\lambda_{\text{easy}} \cdot \frac{e}{E}\right) + \beta_{\text{easy}}, \tag{2}$$

$$\begin{aligned} P_{\text{medium}}(e) =& \alpha_{\text{medium}} \cdot \left(1 - \exp\left(-\lambda_{\text{medium}} \cdot \frac{e}{E}\right)\right) \\ &+ \beta_{\text{medium}} \cdot \exp\left(-\lambda_{\text{medium}} \cdot \frac{e}{E}\right), \end{aligned} \tag{3}$$

$$P_{\text{hard}}(e) = \alpha_{\text{hard}} \cdot \Big(1 - \exp\Big(-\lambda_{\text{hard}} \cdot \frac{e}{E}\Big)\Big) + \beta_{\text{hard}}, \quad (4)$$

where e denotes the current epoch, E is the total number of epochs, α_x and β_x control the sampling weights of each difficulty level, and λ_x regulates the rate of decay or increase. By constraining the probabilities across difficulty levels, the scheme ensures that they form a valid probability distribution throughout training.

C. Contrastive Learning

This section presents a curriculum-guided dual-branch graph-text contrastive learning framework for parallelism detection. The model jointly encodes source-code sequences and DFGs, and aligns the two modalities in a shared embedding space via contrastive objectives. This design encourages representations that capture both sequential code semantics and variable-level data dependences, improving discrimination and maintaining robustness under structurally-preserving perturbations.

Specifically, for each code sample x_i, the unmodified source code is treated as the primary view. In parallel, a semantically equivalent but perturbed view is generated via data augmentation operators, forming a positive pair.

The primary encoder f_θ encodes the original code view and is optimized by backpropagation through the detection and multi-task objectives. The momentum encoder f_{θ_m} encodes the augmented view and serves as the key encoder; it is not updated by gradients but follows f_θ via the MoCo momentum update [8]:

$$\boldsymbol{\theta}^m \leftarrow m \cdot \boldsymbol{\theta}^m + (1 - m) \cdot \boldsymbol{\theta}, \quad (5)$$

where m denotes the momentum coefficient, which suppress batch-to-batch representation drift and helps maintain continuity and stability of augmented views in the contrastive space. Building on this design, we maintain a difficulty-aware queue on the momentum encoder side to cache negative samples across mini-batches. Compared with using only in-batch negatives, this configuration increases both the number and diversity of negative samples without materially enlarging the batch size. This property is particularly advantageous for parallelism detection: under sparse parallel annotations and highly heterogeneous code distributions, it provides stable and abundant negative samples.

We design contrastive objectives to capture parallelism-relevant sequential semantics and dependence structure in a unified embedding space through two alignment tasks: text-to-text and graph-to-text. Text-to-text alignment reinforces consistency across augmented views of the same program, improving the text encoder's ability to discriminate parallelism. Graph-to-text alignment explicitly constrains cross-modal consistency between code sequences and DFGs, ensuring that sequential semantics and dependence structure remain aligned.

In the text-to-text branch, for the i-th sample, the primary encoder outputs a normalized vector q_i, and the momentum encoder outputs an normalized vector k_i. Keys from previous batches are stored in a queue and used as negative samples $\{\mathbf{k}_j^-\}$. We optimize the model using the following loss function:

$$\mathcal{L}_{\text{text}} = -\log \frac{\exp\big(\mathbf{q}_i^\top \mathbf{k}_i / T\big)}{\sum_{j=1}^{K} \exp\big(\mathbf{q}_i^\top \mathbf{k}_j^- / T\big)}, \quad (6)$$

where T is the temperature hyperparameter that controls the smoothness of the similarity distribution.

In our experiments, we observe that code structure and dependence patterns are highly complex. Without additional regularization, training often suffers from gradient oscillations or overfitting to easy samples in the early stage. To alleviate these issues, we introduce an additional constraint during representation learning. Specifically, we model the cosine similarity of positive pairs c_i and define an upper bound on its expected value based on the current training progress:

$$s_{\max}(p) = s_{\min} + (s_{\max}^{\text{final}} - s_{\min}) \cdot p, \quad (7)$$

where $s_{\min}$ and $s_{\max}^{\text{final}}$ denote the predefined lower bound and target upper bound on initial similarity, respectively. We impose a penalty term on samples whose similarity exceeds the upper bound:

$$\mathcal{R}_{\text{high}} = \frac{1}{B} \sum_{i=1}^{B} \Big(\max\big(0,\ c_i - s_{\max}(p)\big)\Big)^2, \quad (8)$$

where B represents the number of samples in the current batch. This yields the overall text-to-text contrastive loss:

$$\mathcal{L}_{\text{text\_loss}} = \mathcal{L}_{\text{text}} + \lambda_{\text{reg}}\, \mathcal{R}_{\text{high}}, \quad (9)$$

where λ_{reg} controls the penalty strength and balances the relative contribution of the constraint term. With this regularization, the model is encouraged to maintain moderate positive-pair similarity at the early stage, promoting robust alignment and reducing the risk of representational collapse. As training proceeds, the target similarity is gradually increased to sharpen decision boundaries, and mitigate premature clustering of samples that can undermine discriminative capacity. In addition, we track positive-pair similarity statistics online and use them to perform difficulty-aware updates of the negative-sample queue. This mechanism prioritizes structurally complex and semantically ambiguous instances during negative sampling, which can further smooths the optimization trajectory.

The semantics of source code extend beyond lexical tokens, and many dependence relations are more directly captured by DFGs. To address the limited structural modeling capacity of purely sequential encoders, we extend contrastive learning from a single textual view to a multimodal setting that jointly models code sequences and DFGs. For each DFG node, we keep its associated token index set and form an initial node representation by average pooling the corresponding token embeddings. This construction injects variable-level dependence structure from the DFGs into the graph branch. On top of these node features, we employ a lightweight GNN to encode structural signals, complementing the code sequence branch with structure-aware representations.

At the graph-level aggregation stage, we adopt a dual-perspective structural aggregation strategy that combines gated global pooling with hub-node pooling, capturing both overall data-flow patterns and the contributions of critical nodes and edges. Specifically, we feed each node representation $\mathbf{h}_i$ into a gating network to obtain an importance weight w_i, and then compute a global graph representation via weighted normalization:

$$\mathbf{h}_{\text{global}} = \frac{\sum_{i=1}^{\mathcal{V}} w_i \, \mathbf{h}_i}{\sum_{i=1}^{\mathcal{V}} w_i}, \tag{10}$$

where $|\mathcal{V}|$ denotes the number of nodes in the DFG. Gated pooling summarizes the overall data-flow structure while adaptively emphasizing structurally salient subcomponents through learned importance weights.

Subsequently, we capture structural salience via node degree. Let $\deg(v)$ denote the degree of node $v \in \mathcal{V}$. Using a quantile of the empirical degree distribution as a threshold, we select a hub-node set $\mathcal{H}$ consisting of nodes whose degrees exceed the threshold, and then apply average pooling over their representations:

$$\mathbf{h}_{\text{hub}} = \frac{1}{|\mathcal{H}|} \sum_{i \in \mathcal{H}} \mathbf{h}_i. \tag{11}$$

By structurally emphasizing key variables on complex dependence paths, the model captures overall data-flow patterns via the global representation $\mathbf{h}_{\text{global}}$ and further focuses on influential nodes through $\mathbf{h}_{\text{hub}}$. This dual-perspective design enables graph-level representation learning from both holistic and salient-structure views. We then apply a linear projection to map the graph representation to match the dimensionality of the text embedding space. The alignment between the two modalities is optimized using the following loss:

$$\mathcal{L}_{\text{graph\_loss}} = -\log \frac{\exp(\mathbf{z}'_t \cdot \mathbf{z}'_g / T)}{\sum_{j=1}^{K} \exp(\mathbf{z}'_t \cdot \mathbf{z}^-_{g,j} / T)}, \tag{12}$$

where $\mathbf{z}_{t'}$ denotes the projected representation from the text branch, and $\{\mathbf{z}^-_{g,j}\}$ denotes a negative sample.

In summary, we combine the text-to-text and graph-to-text contrastive losses to form the overall regularization objective, and introduce two weighting hyperparameters to balance semantic consistency and structural alignment:

$$\mathcal{L}_{\text{loss}} = \lambda_1 \, \mathcal{L}_{\text{text\_loss}} + \lambda_2 \, \mathcal{L}_{\text{graph\_loss}}. \tag{13}$$

During training, we observe that fixed augmentation policies may be insufficient to accommodate the representational needs of different stages. Excessively strong perturbations in early training can destabilize optimization, whereas overly weak augmentation after partial convergence limits the discovery of harder examples and sharp decision boundaries. To resolve this tension, we introduce a dynamic augmentation scheduling mechanism that adapts augmentation strength based on the training state. Specifically, we implement a scheduled augmentation pipeline whose online adjustments are governed by measurable statistics and explicit rules, jointly informed by training progress and contrastive learning behavior, thereby improving interpretability and controllability.

In implementation, we use the positive-pair similarity c_i as a scheduling signal. When c_i becomes overly high, the embeddings of the two views may align too rapidly and lose diversity, increasing the risk of representation collapse. We therefore increase the augmentation probability and strength to encourage greater view diversity. Conversely, when c_i is low to moderate, we reduce the perturbation strength to avoid over-separating the two views and destabilizing representation learning. This mechanism establishes a closed-loop feedback between data augmentation and representation learning, effectively mitigating representation collapse and improving the efficiency of contrastive learning across different training stages.

IV. Evaluation

In this section, we conduct a series of experiments to evaluate the effectiveness of the proposed framework for detecting potential parallelism in programs.

A. Experiment settings

All experiments were conducted on a single NVIDIA A800 GPU using the Open-OMP-Plus dataset [26], partitioned into training, validation, and test sets at an 8:1:1 ratio. We employed the Binary Cross-Entropy (BCE) loss function, combined with the AdamW optimizer and a linear warmup learning-rate schedue (initial learning rate of $3\text{e}{-5}$, weight decay of $1\text{e}{-2}$, Adam ϵ of $1\text{e}{-8}$). For contrastive learning, the momentum coefficient is set to $m = 0.95$, the temperature parameter to $T = 0.25$, and the queue size to $K = 2048$. To prevent data leakage, benchmarks overlapping with the training set were excluded. Furthermore, no pretrained weights were utilized in any experiment to assess the model without external pretraining.

TABLE I: The distribution of each class for each programming language: (left) loop-level OpenMP labels by language; (right) shared-memory attributes.

Category	Description		Clauses	
	C	C++	private	reduction
With OpenMP	14,906	8,241	6,758	3,267
Without OpenMP	17,193	14,323	–	–
Total	32,099	22,564	10,025	

B. Result

We now present empirical results to evaluate the effectiveness of the proposed framework on the target task.

To comprehensively evaluate code-model performance, we compare against several representative baselines, including both task-specific models and pretrained code models. Table II summarizes the results across models, where our approach achieves the strongest overall performance on this task. The improvement in pragma recall indicates that the model more accurately identifies samples with parallelism potential. This improvement also propagates to the remaining labels, yielding consistent gains across metrics. These results suggest that

TABLE II: Effects of different code models on parallelism detection.

Model	Metrics								
	Accuracy			Precision			Recall		
	Pragma	Private	Reduction	Pragma	Private	Reduction	Pragma	Private	Reduction
PragFormer [19]	0.841	0.924	0.953	0.793	0.716	0.632	0.847	0.663	0.598
GraphCodeBERT (Separated) [27]	0.870	0.937	0.963	0.850	0.768	0.690	0.841	0.684	0.688
OMPify [26]	0.872	0.938	**0.966**	0.849	0.755	0.690	0.848	0.689	0.700
UniXcoder [22]	0.868	0.935	0.963	0.845	0.794	0.675	0.838	0.649	0.738
Ours	**0.881**	**0.940**	**0.966**	**0.852**	**0.800**	**0.731**	**0.851**	**0.723**	**0.741**

formulating the three labeling tasks as a unified multi-label classification problem enables shared supervision and improves overall predictive accuracy. Concretely, once the model predicts concurrency, the shared representation simultaneously informs the estimation of shared-memory attributes, which are often expressed implicitly through OpenMP pragmas.

C. Ablation Study

To further validate the effectiveness of the proposed framework, we conducted ablation studies, with the results reported in Table III. As shown, introducing graph-branch contrastive learning on top of the base model improves performance, indicating that contrastive constraints from the data-flow-graph view enhance sensitivity to dependence structure and key variables. Applying text-branch contrastive learning alone also yields consistent gains, suggesting that augmented code views improve robustness to limited training data and stylistic variation. When both branches are enabled, performance improves further, highlighting their complementarity: the graph branch captures richer data-dependence information, while the curriculum-guided text branch provides more robust sequential representations. Overall, these results support the effectiveness of multimodal contrastive learning for parallelism detection from both structural and semantic perspectives.

TABLE III: Ablation study of the proposed framework on the parallelism detection task.

Setting	GNN	CL	F1 (%)	Acc (%)
Baseline	No	No	78.9	90.7
+GNN only	Yes	No	79.7	91.6
+CL only	No	Yes	80.4	91.7
Full	Yes	Yes	**81.1**	**91.9**

D. Benchmark Suite Evaluation

We evaluate the model's generalization on three HPC benchmark suites: NAS, SPEC, and HeCBench. These benchmarks cover parallel workloads, compute-intensive applications, and heterogeneous computing scenarios, respectively. From each suite, we selected C/C++ programs containing OpenMP pragmas; the dataset statistics are summarized in Table IV. We use OMPify as the baseline and conduct all evaluations under identical settings. As shown in Table V, our approach consistently outperforms OMPify across all three benchmarks.

TABLE IV: Statistics of the benchmark suites used for evaluation.

Benchmark	With OMP	Without OMP	private	reduction
NAS [28]	166	146	12	2
SPEC [29]	157	1000	1	0
HeC [30]	753	449	483	199

TABLE V: Comparison with OMPify on different benchmark suites.

Benchmark	Model	P	R	Acc
NAS [28]	OMPify [26]	0.731	0.886	0.766
	Ours	**0.768**	**0.933**	**0.914**
SPEC [29]	OMPify	0.572	**0.854**	0.894
	Ours	**0.794**	0.798	**0.949**
HeCBench [30]	OMPify	0.712	0.501	0.688
	Ours	**0.887**	**0.553**	**0.794**

E. Case study

Across the benchmark suites, we compare the model's predictions of code parallelism and related attributes against the *ground truth*. For mismatched samples, we perform targeted code-level inspection to diagnose whether discrepancies arise from model errors, conservative omissions in the ground truth, or annotation assumptions that rely on implicit compiler transformations.

Overall, the model exhibits a useful tendency: it is aggressive in recovering missed parallelism in structurally regular kernels, while remaining conservative when safe parallelization depends on non-trivial reduction transformations. In Fig. 4a, the ground truth labels the loop as non-parallel (`pragma=0`), but the model predicts parallelism (`pragma=1`). The loop satisfies data-parallel conditions—iterations write to disjoint `A[i][j]` elements, `B`/`C` are read-only, and `sum` is iteration-local—suggesting a conservative omission in the ground truth. In contrast, Fig. 4b and Fig. 4c are reduction-dependent: the ground truth treats these loops as parallelizable by assuming implicit summation/max reductions, whereas the model withholds parallelism and reduction labels because, as written, the code contains loop-carried dependences on shared scalars (`sumval` and `max_error`) that require an explicit reduction transformation for safe parallelization.

Taken together, these cases indicate that the model can recover missed parallel regions where benchmark annotations remain conservative, yet avoids overly optimistic decisions

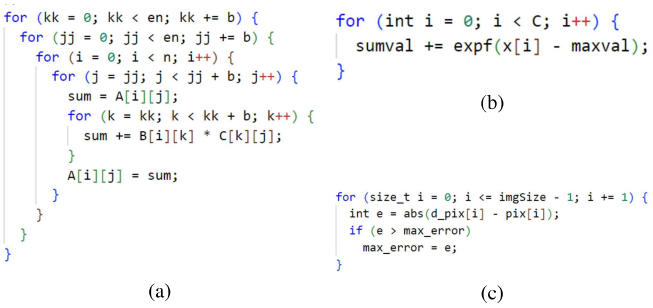


Fig. 4: Typical samples selected from the benchmark set. (a) shows data parallelism in blocked matrix multiplication missed by the benchmark. (b) reflects a conservative decision on softmax accumulation (implicit reduction). (c) illustrates dependence handling for the maximum-error pattern in image processing (max-reduction).

under complex reduction-related dependences, thereby helping reduce the risk of semantic violations in downstream automatic parallelization.

V. Conclusion and Future Work

This paper studies parallelism detection for automatic parallelization under complex data dependences. We propose CurParCL, a curriculum-guided dual-branch graph-text contrastive learning framework that jointly encodes source-code sequences and data-flow structure, strengthening representations through structure-aware dependence modeling and cross-modal alignment. Experiments show improvements over representative baselines across multiple parallelism detection tasks, indicating the value of integrating structural and semantic views in a unified training paradigm.

Future work will proceed in two directions. First, we will extend CurParCL to support finer-grained OpenMP pragmas and clause configurations. Second, we will use the detected parallelism signals to guide and normalize OpenMP pragma generation, moving toward a more complete and reliable automatic parallelization pipeline.

References

[1] T. Heller *et al.*, "Discopop: A profiling tool to identify parallelization opportunities," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2015.

[2] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2017.

[3] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning (ICML)*, ser. Proceedings of Machine Learning Research, vol. 70. PMLR, 2017, pp. 1263–1272.

[4] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations (ICLR)*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-

[5] Y. Shen, M. Peng, S. Wang *et al.*, "Towards parallelism detection of sequential programs with graph neural network," *Future Generation Computer Systems*, vol. 125, pp. 515–525, 2021.

[6] A. van den Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," *arXiv preprint arXiv:1807.03748*, 2018.

[7] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *Proceedings of the 37th International Conference on Machine Learning (ICML)*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 1597–1607.

[8] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 9729–9738.

[9] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, 2009, pp. 41–48.

[10] W. Blume, R. Doallo, R. Eigenmann *et al.*, "Polaris: The next generation in parallelizing compilers," in *Proceedings of Frontiers '96*. IEEE, 1996.

[11] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008, pp. 101–113.

[12] C. Dave, H. Bae, S.-J. Min *et al.*, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, 2009.

[13] M. Amini *et al.*, "Par4all: From convex array regions to heterogeneous computing," in *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques (HiPEAC)*, 2012.

[14] N. Morew, M. Norouzi, A. Jannesari *et al.*, "Skipping non-essential instructions makes data-dependence profiling faster," in *European Conference on Parallel Processing (Euro-Par)*. Springer, 2020, pp. 3–17.

[15] R. Eigenmann, I. Park, and M. J. Voss, "Are parallel workstations the right target for parallelizing compilers?" in *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, 1996.

[16] S. Prema, R. Jehadeesan, and B. K. 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] D. Grewe and M. F. P. O'Boyle, "A static task partitioning approach for heterogeneous systems using machine learning," in *International Symposium on Code Generation and Optimization (CGO)*, 2011.

[18] T. Kadosh *et al.*, "Ompar: Automatic parallelization with AI-driven source-to-source compilation," arXiv preprint arXiv:2409.14771, 2024.

[19] 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 (PPoPP)*, 2023.

[20] D. Nichols *et al.*, "HPC-coder: Modeling parallel programs using large language models," in *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. Prometeus GmbH, 2024.

[21] Z. Feng *et al.*, "Codebert: A pre-trained model for programming and natural languages," arXiv preprint arXiv:2002.08155, 2020.

[22] D. Guo *et al.*, "Unixcoder: Unified cross-modal pre-training for code representation," arXiv preprint arXiv:2203.03850, 2022.

[23] S. Liu, B. Wu, X. Xie *et al.*, "Contrabert: Enhancing code pre-trained models via contrastive learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2476–2487.

[24] S.-A. Rebuffi, S. Gowal, D. A. Calian *et al.*, "Data augmentation can improve robustness," *Advances in Neural Information Processing Systems*, vol. 34, pp. 29 935–29 948, 2021.

[25] S. Guo, W. Huang, H. Zhang *et al.*, "Curriculumnet: Weakly supervised learning from large-scale web images," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 135–150.

[26] T. Kadosh *et al.*, "Advising OpenMP parallelization via a graph-based approach with transformers," in *International Workshop on OpenMP (IWOMP)*. Springer Nature, 2023.

[27] D. Guo *et al.*, "Graphcodebert: Pre-training code representations with data flow," arXiv preprint arXiv:2009.08366, 2020.

[28] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks," NASA Ames Research Center, Tech. Rep. RNR-91-002, 1991. [Online]. Available: https://ntrs.nasa.gov/citations/19910071476

[29] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[30] J. Schäffeler, B. Elis, A. Raoofy, J. Weidendorfer, and M. Schulz, "Poster: Performance comparison of gpu programming models using hecbench benchmarks," in *Proceedings of the 22nd ACM International Conference on Computing Frontiers 2025 (CF '25)*. Association for Computing Machinery, 2025, pp. 226–227.