Multi-View Learning for Parallelism Discovery of Sequential Programs

Le Chen
Department of Computer Science
Iowa State University
Ames, USA
lechen@iastate.edu

Quazi Ishtiaque Mahmud
Department of Computer Science
Iowa State University
Ames, USA
mahmud@iastate.edu

Ali Jannesari
Department of Computer Science
Iowa State University
Ames, USA
jannesari@iastate.edu

***Abstract*—Identifying suitable parallelizable regions in sequential programs is a crucial task for performance optimizations. Traditional methods like static and dynamic analysis have flaws like insufficient accuracy or high overhead runtime. Recent studies are more interested in applying machine learning techniques to this topic. The crux of parallelism discovery with machine learning is to generate meaningful code representations. One promising route is to exploit the dependence graph through Graph Neural Networks (GNNS). In this paper, a novel multi-view framework is proposed to automatically detect potential parallelism opportunities. Sequential programs are first represented by program execution graphs encompassing both semantic and structural information. Then two independent views are defined: namely, a structural pattern view and a node feature view. In the structural view, local graph structural patterns are captured via random anonymous walks and then fed into a Graph Convolutional Network (GCN). The node features, both dynamic and static, are fed into another GCN in the node feature view. In addition, a multi-view model is designed to unify the node features and the structural features for parallelism detection. Our approach achieves comparable state-of-the-art performance on parallel region classification with an accuracy up to 92.6% when evaluated with popular parallel computing benchmarks.**

***Index Terms*—machine learning, artificial intelligence, parallel program language**

I. Introduction

The rapid development of data and compute-intensive applications has resulted in a significant shift toward parallelism in typical performance programming approaches. Parallel programming is generally recognized as a viable and effective approach for developers to speed up computation and match the expansion of chip architecture [1]. Designing and implementing parallel applications, on the other hand, is a difficult undertaking that takes time and necessitates a thorough understanding of both the original serial application and parallelism techniques. Parallel programming models or libraries such as OpenMP [2] and Open MPI [3] provide a solution for generating parallel versions of sequential programs, but programmers must have substantial knowledge to use them effectively. A significant barrier when using these techniques to process sequential programs for parallelization is the detection of parallelism opportunities [4].

Multiple static or dynamic attempts have been made in past studies to find parallelism automatically in sequential programs. However, the challenge of effective potential parallelism identification is still unsolved. Following the recent successes of machine learning and natural language processing (NLP), research on using similar methods to code analysis has been conducted. These methods commonly treat the code as natural language phrases, with NLP models processing the code segments directly. There are, however, some major limitations and restraints. In NLP, words are frequently embedded with context information, especially for words that are not part of the word vector vocabulary. On the other hand, instructions of code do not have to be related to their contextual instructions. As a result, directly using NLP approaches to code analysis misses out on key structural elements like function calls, branching, and interchangeable statement order [5]. Furthermore, due to differences in syntax, the performance of these approaches differs amongst computer languages.

In addition to the studies that threat the code as natural language phrases, data, and control dependence graphs have been used to represent programs for parallel optimization since the 1980s [6] as they can capture the structural pattern of the programs. Inspired by this, researchers present the code with graphs and apply Graph Neural Networks (GNNs) techniques because of their ability to interpret graph-structured data. As a powerful tool for machine learning on graphs, GNNs can recursively incorporate neighboring nodes' information and capture the graph structure simultaneously with node features [7]. Success has been achieved in many applications such as knowledge-graphs [8], [9], social networks [10], [11], and recommendation systems [12], [13].

Unlike token representations of programs, graph representations can capture structural information using a topological graph while also preserving semantic information using node and edge attributes. As a result, parallelism discovery with GNNs looks promising. However, GNNs are notoriously difficult to apply in code analysis. First, an effective approach to representing code in graphs is required to properly encode features of the code. Second, graph embeddings are required for GNNs. Ben-Nun Et Al. [5] presents a learnable representation of code semantics. The work of Jain Et Al. [14] learns contextual representations of source code by reconstructing tokens from their context. Yet, the representation of the structural information and its aggregation with node features

978-1-6654-9747-3/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPSW55747.2022.00059

is still under-researched.

To address the above challenges, we propose a multi-view Graph Neural Networks (MV-GNN) approach for parallelism discovery in sequential programs. With the help of profiling tools, we represent the sequential programs as dependency graphs where each loop can be taken as a sub-graph. The proposed MV-GNN model comprises two sub-modules that produce graph embeddings based on node and the local structural patterns, respectively. As a result, it examines the code graph from two views simultaneously. Meanwhile, to address the problem of limited data, we use several approaches to generate datasets. We define the objectives of this paper as demonstrating an MV-GNN approach that can be applied to the parallelism region discovery for sequential programs. Our work makes the following contributions:

- We propose a structural embedding method for sequential programs. With the help of anonymous random walks [15], we construct embeddings for each node after they have been turned into graphs.
- Using the proposed MV-GNN model, we present a novel technique to leverage both the structural and node features of sequential programs in graphs for parallelism discovery. Experiments are carried out to show that our framework is effective and achieves equivalent state-of-the-art performance.
- To deal with the issue of a small dataset during training, we use a variety of strategies to build a labeled dataset that not only fulfills our needs but can also be used for other code analysis applications.

II. Related Work

This section provides a quick overview of parallelism detection, GNNs, and research of parallelism detection using machine learning. Although this field is currently understudied, new research has demonstrated the capabilities of GNNs in this area.

A. Parallelism Detection

Parallelism discovery is the process of analyzing program fragments (often authored for serial execution) to identify opportunities for parallelism. Parallelism can be expressed through two fundamental concepts, task-level parallelism and loop-level parallelism. Task-level parallelism defines regions from an application that can be executed simultaneously on multiple cores or threads. Yet task-level parallelism methods require pre-defined distinct regions in the program to restrain fine-grained opportunities. Loop-level parallelism considers loop bodies as parallel regions whose iterations can be distributed across threads [16]. This paper focuses on loop-level parallelism pattern discovery.

There are two principal strategies for detecting data dependencies: static and dynamic methods. In static methods, the dependencies are inferred by analyzing the program during compilation. The static analysis can discern dependencies across all code sections of an analyzed program. Basic dependencies can be drawn from local and shared loops variables, providing a general understanding of functional and data dependencies. Nevertheless, it is impossible to capture dependencies caused by pointer aliasing or complex array indexing due to the lack of run-time information. Hence, static analysis is reckoned to be conservative for the discovery of parallelism and is prominently used in Integrated Development Environments (IDE) for real-time error checking [17]. In contrast, dynamic methods overcome the inherent problems of static methods through program instrumentation and execution. Additional analysis instructions are injected into the source code during the instrumentation stage. Then, the run-time statistics of the original program are generated upon execution of instrumented programs. Although, dynamic methods also have some disadvantages. The outcome of the analysis depends on the input to the instrumented program since some code segments may never run due to the nature of the input data. It also increases the time cost for instrumenting a program and memory overhead during execution. Depending on the application and the respective dataset, this could require significantly more computing resources than some machines have available. As an illustration, the Polybench application dataset [18] can easily exceed 32 GB of RAM during the dynamic analysis of the instrumented executables (when utilizing the LARGE input dataset).

Many static, dynamic, and hybrid (i.e., both static and dynamic) tools have been developed for automatic parallelization opportunity identification. Polly [19] is an automatic parallelism detection tool based on static analysis, LLVM [20], and the polyhedral model. Kremlin [21] determines the length of the critical path in the loops using dependency information and then calculates a metric, namely self-parallelism, for parallelism detection. Alchemist [22] discovers the candidates of parallelization by comparing the number of instructions with the read-after-write (RAW) dependencies, both of which are generated by Valgrind [23] during run-time. DiscoPoP [24]–[26] extracts dynamic profiling and instruction dependency data from instrumented sequential programs. Information like dependency type, the number of incoming and outgoing dependencies, and critical path length is extracted from a data dependency graph for parallelism detection. As a hybrid method tool, DiscoPoP provides detailed dynamic analysis statistics that can inform tooling in addition to static analysis, providing an improved understanding useful for detecting parallel opportunities. Our work uses the output of DiscoPoP as the basis for our static and dynamic analysis data.

In addition to static or dynamic data, research has shown that graph structure features can find parallelism. Techniques such as shortest pathways [27] and graphlets [28] have been studied. For some parallel patterns, such as stencil and reduction, graph structure properties are critical for parallelism detection, as shown in figure 1. However, in traditional methods, the arbitrary depth of input graphs frequently requires pattern matching, which is difficult to afford in large projects.

In conclusion, both static and dynamic information and the graph structure features of the graph representation of the code can serve for parallelism discovery. In some cases,

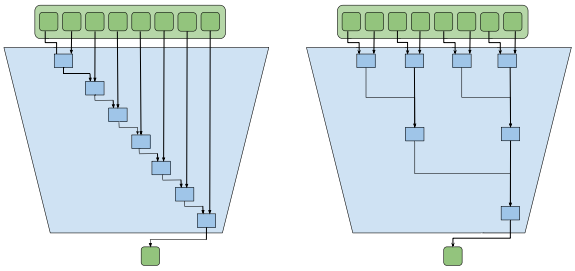

Fig. 1. Illustration of parallelization patterns: stencil (left) and reduction (right). For these parallelization patterns, graph structure patterns can be easily captured for classification.

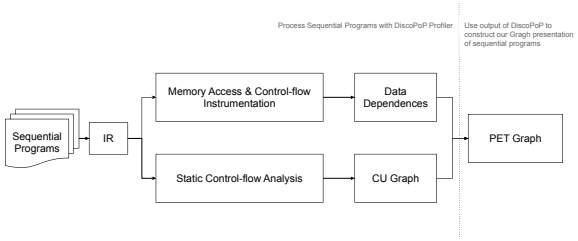

Fig. 2. Process of constructing graphs for code in this work. Sequential programs are instrumented with the DiscoPoP profiler, generating the data denpendences information and the control-flow graph called the computational unit (CU) graph. We merge the output of DiscoPoP to generate a graph representation of the sequential programs called program execution graph (PEG). This process can be done with other profiling tools.

like code with stencil and reduction patterns in figure 1, the structure features outperform static and dynamic information. In this paper, we adopt a multi-view architecture to consider both static and dynamic data and graph structure aspects for parallelism discovery.

B. Graph Representation of Code with Profilers

Graph representation of code has been used widely in the field of code analysis. To represent programs in graph, usually a profiler tool has to be utilized for instrumenting the code. DiscoPoP is an open-source project that uses the LLVM Pass framework to initially instrument serial code. It uses a hybrid analysis technique (i.e., both static and dynamic) to extract data and control dependencies as well as control region information of a program. This data is then further analyzed for the discovery of parallelization opportunities. This paper utilizes DiscoPoP to extract the dynamic information for later analysis by our model. It is worth noting that DiscoPoP does not limit our work because we can build the graph representation of code using any profiler tool.

Figure 2 shows the workflow of DiscoPoP. There are three main phases of DiscoPoP: The first phase deals with dependency analysis. The memory accesses of the input program are instrumented and control flow information is extracted statically. The instrumented program is then executed to obtain additional run-time data dependencies. The second phase comprises of parallelism discovery and pattern identification. Analysis of the extracted dependencies is performed using multiple techniques including pattern detection and priority ranking. In the final phase, parallelism opportunities discovered in the second phase are sorted according to various metrics including coverage, speed-up, and load imbalance. The post-processing phase of DiscoPoP requires fine-tuned detection applications and reconfiguration upon being presented new programs. Instead, in this paper, we utilize the data generated from phase 1 of DiscoPoP as the basis for our input data and leverage the latest machine learning techniques to overcome the disadvantages of DiscoPoP. Further implementation details are provided in section III.

C. Code Analysis with Machine Learning

Background. Machine learning (ML) has made outstanding achievements in many fields, which has empowered works in a variety of areas of code analysis. Ali et al. [29] build a deep neural network model to find data race bugs instead of creating a data race detector manually. [30] applies Machine Learning techniques on static features extracted from Android's application files for the classification between tools and games. A set of classifiers like Decision Tree, Bayesian Networks, PART, Boosted Bayesian, and others are evaluated in their tests. [31] applies Machine Learning Techniques onto Code Smell Detection to overcome the subjectiveness of developers and the difficulties in finding suitable thresholds to be used for detection in traditional methods

Thus far, machine learning techniques are under-explored and rarely utilized in parallelism detection tasks.

[32] investigates an automatic method for classifying qualified regions of sequential programs that could be parallelized. They use benchmarks with hand-annotated OpenMP directives for training. Various classification methods are evaluated in their work. However, machine learning techniques have achieved significant progress since [32]'s work, and the latest techniques have shown the ability to capture a context's pattern. Being inspired, we propose a parallelism detection approach utilizing modern the machine learning techniques in this work.

Graph Neural Networks (GNNs). Gori et al. [33] first outline the concept of GNNs. Scarselli et al. [34] develop the idea of GNNs by using neural networks to process data in graph domains. Graph neural networks typically use a neighborhood aggregation framework to do representation learning, in which node features are updated by recursively aggregating and transforming features from their neighbor nodes. There are mainly two categories for GNN [35]: spectral-based and spatial-based. Spectral-based GNNS employ convolutional neural networks in the spectral domain using the graph Laplacian [36]–[38]. Spatial-based GNNs formulate graph convolutions as aggregating feature information from neighbors, which recurrently apply neural networks to every node of the graph [12], [39], [40].

We briefly introduce the spatial-based GNNs in this section. The input of GNNs is typically constructed as a graph $G = (V, E)$, where V is the node set, and E is the edge set. The target of GNNs is to learn a state h_p for each node $p \in V$. Let Ω_p be the neighbor node set of p, h_p contains the information of neighbor nodes $q \in \Omega_p$. h_p^t is the state at time t and can be updated by:

$$m_p^t = M(\{h_p^t | q \in \Omega_p\}) \tag{1}$$

and

$$h_p^{t+1} = F(m_p^t, h_p^t), \tag{2}$$

where m_p^t is the aggregation of neighboring messages. F and M are update functions used for updating the hidden representation of all nodes in graph and they are shared among the propagation process.

Parallelism Detection with GNNs. Graphs are ubiquitous because of their ability to accurately describe relational data. Park et al. [41] shows that graph-based characterization techniques can outperform non-graph characterization methods. The recent development of GNNs [34], [42] exhibits the GNN as a powerful tool for processing data represented in tasks like graphical structures such as social networks, images, and citations. Meanwhile, sequential programs can naturally be represented in graphs, making GNNs a perfect fit for analyzing code. However, the work of applying GNNs on parallelism analysis is still understudied for the following reasons: a) the lack of generalized graph representation for this topic and b) small dataset that affects the performance of the GNN models. Ben-Nun et al. [5] define an embedding space, called inst2vec, based on the LLVM Intermediate Representation (IR) of the compiled code. They show that the inst2vec embeddings outperform specialized approaches for performance prediction. Though, dynamic features are barely covered in the proposed embedding space. To the best of our knowledge, Shen et al. [43] are the first to apply GNNs on parallelism discovery. However, their training dataset relies on Pluto, a polyhedral model tool, making it hard to generalize the framework for generalized data.

III. Multi-View GNNs for Parallelism Discovery

In this section we introduce our MV-GNN model for parallelism discovery in sequential programs. Figure 3 shows an overview of our model. We first introduce the graph representation of sequential programs in our model. Anonymous random walks are collected from each node and processed to collect the structural features of the node neighborhoods. Node features are generated with inst2vec [5] embedding and concatenated with dynamic features generated by DiscoPoP.

A. Graph Representation of Code

Computational Units. As mentioned in section II-B, the output of DiscoPoP first phase is utilized to extract static and dynamic features from serial programs. To represent the input program, multiple CUs can be stacked together to represent the input program. An example of two CUs generated from a code section is shown in Figure 4. The variable x is given a value on line 3. Lines 5 and 6 use the variable x to compute various values, while line 7 assigns a new value to it. As a result, these lines create a CU. For the variable y, the same notion applies to lines 4, 8, 9, and 11. As a result, for this example code, DiscoPoP creates two CUs.

Program Execution Graph Program Execution Graphs are the basis of our work and are constructed with CUs and the data dependence information. We define a PEG as a graph $G = (V, E)$, where V is the set of nodes, and E is the set of edges. Each node $v \in V$ is a sub-graph node or a CU. Each Edge $e \in E$ is a directed edge connecting CUs and sub-graph nodes. The attributes of nodes and edges are discussed with the embeddings. A plotted PEG is illustrated in figure 5.

B. Node Feature Embedding

For each node, we intend to contain both static and dynamic data. The inst2vec embedding space was introduced by Ben-Nun et al. [5] based on LLVM IR of the code. The inst2vec is unaffected by the source programming language and can operate with DiscoPoP output due to LLVM IR. They begin by presenting a contextual flow that is very similar to the PEG graphs we used in our model. The inst2vec embedding outperforms specialized techniques once a sufficient vocabulary has been taught. For static/semantic properties of nodes in PEGs, we employ inst2vec as the embedding in this paper. Furthermore, as seen in table I, dynamic features are carried out and integrated with the static/semantic features. These characteristics have been described by Fried et al. [32] that they play a significant influence in parallelism discovery. The feature N_Inst and exec_times represent a direct measurement of a loop's execution cost. They are, respectively, the number of IR instructions in the loop body and the number of times the loop is executed. The critical path length, or CFL, measures the length of the loop's longest sequence of dependent instructions. The estimated speedup, or ESP, is a heuristic calculated using the maximum breadth and critical path length of the dependency graph and Amdahl's Law [44]. If a loop is parallelized, this feature calculates the maximum reduction in run time for that loop.

TABLE I
DYNAMIC FEATURES USED FOR LOOP PARALLELIZATION CLASSIFICATION.

feature name	description
N_Inst	Number of instructions within the loop
exec_times	Total number of times the loop is executed
CFL	Critical path length
ESP	Estimated speedup
incoming_dep	Incoming dependency count
internal_dep	Dependency count between loop instructions
outgoing_dep	Outgoing dependency count

C. Structural Feature Embedding

As previously stated, the structural feature view is critical for detecting parallelism in code that uses specific parallelization patterns such as stencil and reduction. Jin et al. [45] and Long et al. [46] have demonstrated that reasonably long

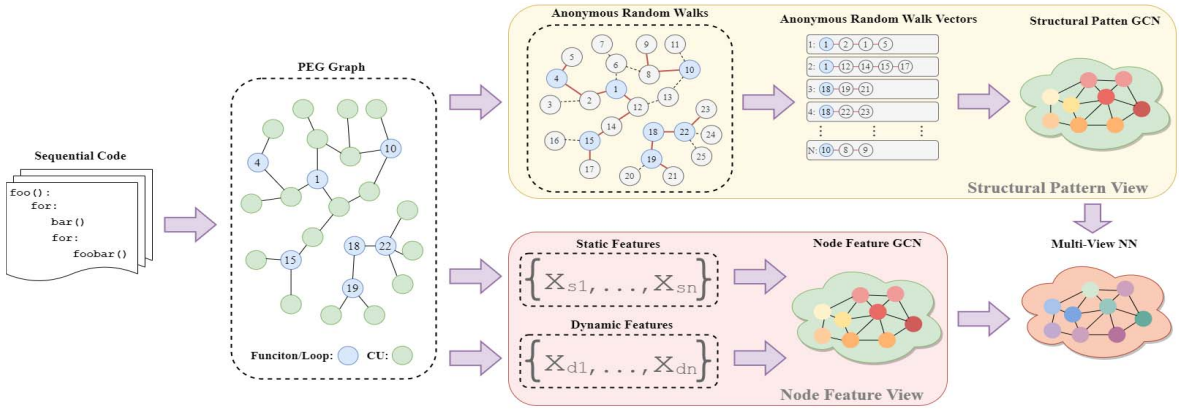

Fig. 3. The processing pipeline of our GNN-based multi-view learning model for parallelism discovery. Code is first stored in Program Execution Graphs (PEGs). Both structural and node features are embed and fed into individual GCNs. A multi-view learning neural network takes the distribution output of the two GCNs and generates a prediction.

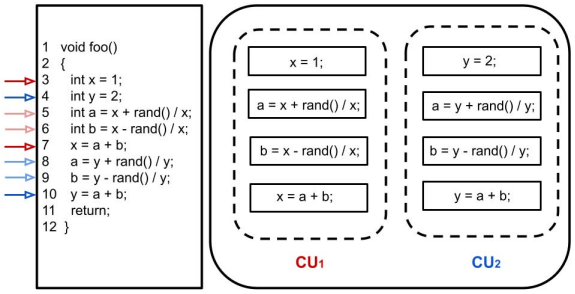

Fig. 4. Two CUs and corresponding code blocks.

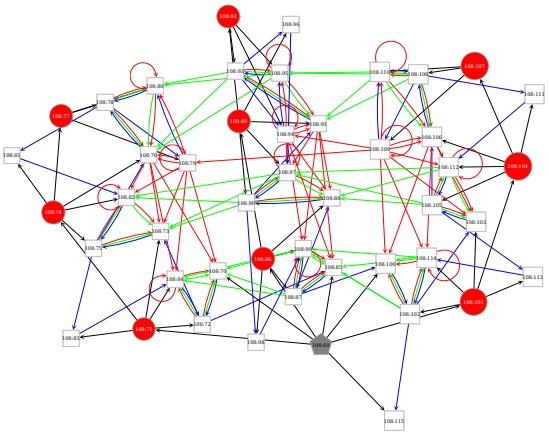

Fig. 5. Example of a PEG graph. The vertices are either CUs, loops or functions. The edges are write-after-read (WAR) or RAW data dependencies with the parent and child. We divide the PEG graph to be different sub-graphs. Each loop and the node within the loop is a sub-PEG for classification.

anonymous walks can reconstruct the graph and are capable of being used to depict general local structural properties. Please refer to their work for a detailed discussion in the rationale of anonymous walks embedding. As a result, we use Anonymous Walks embeddings to collect and embed local structural information. We begin by defining an anonymous walk, as per the definition in [15].

Definition 1: Given a graph $G = (V, E)$, where node set $V = \{v_1, v_2, \ldots, v_{|V|}\}$. $E = \{v_i, v_j\}$ is the edge set, and a random walk $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ where $\langle w_i, w_{i+1} \rangle \in E$. The anonymous walk $\mathrm{aw}(\mathbf{w})$ for $\mathbf{w}$ can be defined as a subset of $\mathbf{w}$ with the exact identities of the nodes in $\mathbf{w}$ removed.

For example, the anonymous walk of random walk $\mathbf{w}$ where $\mathbf{w} = (v_1, v_2, v_3, v_4, v_2)$ is $\mathrm{aw}(\mathbf{w}) = (v_1, v_2, v_3, v_4)$. Following the work of Jin et al. [45], we sample a set of γ random walk sequences $\mathcal{W}^i$ of length l for each node v_i. The empirical distribution $\hat{p}(\omega_j^l | v_i)$ of the underlying anonymous walks of $\mathcal{W}^i$ is a fully connected layer with 400 units and the mean empirical distribution over the graph G is computed by:

$$\hat{p}(\omega_j^l | v_i) = \frac{\sum_{\mathbf{w} \in \mathcal{W}^{(i)}} \mathbb{I}(aw(\mathbf{w}) = \omega_j^l)}{\gamma} \tag{3}$$

$$\hat{p}(\omega_j^l | G) = \frac{\sum_{i=1}^{|V|} \hat{p}(\omega_j^l | v_i)}{|V|} \tag{4}$$

The anonymous walks are represented by a vector through an embedding table lookup in our model.

D. Graph Conventional Networks

Our model employs two independent GCNs to investigate the input data from two different views, relying on suitable embeddings of structural patterns and node attributes. Zhang et al. [47] present a neural network architecture with the Sort Pooling pooling method, which pools nodes by rearranging them in a meaningful order based on their structural responsibilities within the graph. Furthermore, the proposed neural network architecture, Deep Graph Convolutional Neural Network (DGCNN), may accept graphs of any structure, making it an excellent match for our model. Some extra attributes are defined to appropriately input our PEGs into the DGCNN:

- Labels: The benchmark dataset regions are divided into two categories: parallelizable and non-parallelizable. Each loop node in the PEG graph is in just one label set. Please refer section IV for more details about datasets.
- Nodes V: Each node of the PET graph is a graph node or a CU. Other than the node features we discussed above, there is a triple $\langle$ID, START, END$\rangle$ that represents the ID of the CU and the starting and ending line numbers in the code.
- Edges E: Directed edges connecting CUs according to their dependency type. A triple $\langle$SINK, TYPE, SOURCE$\rangle$ where SINK and SOURCE represents the source code locations of the latter and the former memory access, respectively.

As shown in figure 6, The architecture of DGCNN contains steps of graph convolution layers, SortPooling, and a 1-D convolution before it reaches the fully connected layer. We take the input of the fully connected layer into the muli-view model in our model.

E. Multi-View GCN

Multi-view learning models utilize multiple views to complement one another. In our model, two views capture the latent patterns of the structure and the nodes, respectively. Following Long et al. [46], we apply a nonlinear neural network layer on the the node feature GCN kth layer output $h_{i,n}^{(k)}$ and the structural pattern GCN output $h_{i,s}^{(k)}$:

$$h_i^{(K)} = (\mathbf{W} \cdot \tanh([h_{i,n}^{(K)} \oplus h_{i,s}^{(K)}]) + b \tag{5}$$

where $h_i^{(K)}$ is the final output vector of our mode. Upon that, the unsupervised objective of GraphSAGE [40], [46] is adopt for learning and making predictions.

IV. Experiments

In this section, we describe our experimental setup, including the machine environment and the benchmarks used for tests. We then evaluate the effectiveness of our approach for the task of parallelism discovery.

A. Dataset

We check the most commonly used benchmarks for parallelism detection training and testing

NAS Parallel Benchmarks. The NAS Parallel Parallel Benchmark [49] is a well known benchmark suite for evaluating performance of multiprocessor systems. There are both parallel and serial versions of the NPB, providing us an easy way to evaluate our model performance.

PolyBench PolyBench [50] is a benchmark suite consisting of 30 numerical computations extracted from operations in various application domains. It is widely used in performance evaluation in data mining, image processing, and linear algebra computation, etc.

Barcelona OpenMP Tasks Suite (BOTS) The BOTS benchmark [51] contains nine applications using OpenMP tasks. There also exists the serial version of the BOTS.

TABLE II
STATISTICS OF EVALUATED DATASETS: THE NUMBER OF FOR-LOOPS CONTAINED IN EACH TEST BENCHMARKS

Application	Benchmark	Loops #
BT	NPB	184
SP	NPB	252
LU	NPB	173
IS	NPB	25
EP	NPB	10
CG	NPB	32
MG	NPB	74
FT	NPB	37
2mm	PolyBench	17
jacobi-2d	PolyBench	10
syr2k	PolyBench	11
trmm	PolyBench	9
fib	BOTS	2
nqueens	BOTS	4
Total		**840**

Table II lists the number of for-loops for each application from above selected benchmarks.

Transformed dataset We extract the loops from the benchmarks above into parallelizable and non-parallelizable categories. When the benchmark does not have labels, we classify it using tools like DiscoPoP and Pluto. We use transformations such as modifying the operation type and loop order to generate more data. Then we applied six different optimization options of the clang compiler to build six different LLVM-IR intermediary representations of each source code. As illustrated in figure 2, we can use LLVM IR representation of the source codes as input for our model.

B. Experiment Settings

Both benchmark and transformed data are included in the final dataset for training and testing. We utilize the method described above to construct a dataset with 3100 parallelizable loops and 3100 non-parallelizable loops to balance the number of parallelizable and non-parallelizable loops. The dataset, which contains 6200 sets of data, is split into two parts with a 75:25 ratio, with 75% of the data used for model training and 25% for model testing. There are no common objects in the training and testing sets. The model is trained to tell whether or not one loop is parallelizable using supervised learning. Our testing is performed on a computer with an Intel(R) Core(TM) i7-10700K CPU running at 3.80GHz and an NVIDIA GeForce 2080 Ti GPU. All models are written in Python 3.6 in an Ubuntu 20.04 system. Both DGCNNs are set with 200 node feature dimensions; the learning rate is 1e-5; the number of epochs is 200; the loss function is a softmax loss function; the temperature parameters is 0.5; k of the SortPooling layer is 135. Figure 7 shows the training result of the our approach.

C. Baseline Models

Hand-crafted Classifiers, GNNs with Static Information, and Neural Code Comprehension (NCC) architecture are the three baseline models we choose. The first two models have no publicly available data or code. So, in this section, we use the benchmark dataset to evaluate our model and compare it to

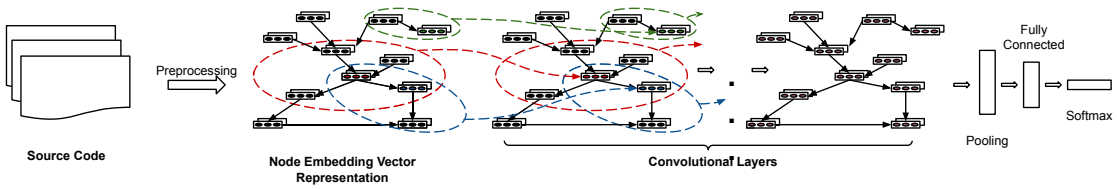

Fig. 6. The architecture of the DGCNN [48]

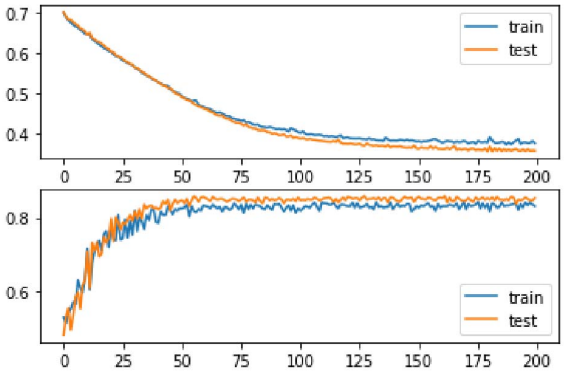

Fig. 7. Loss (above) and Accuracy (below) of the training process for the Generated Dataset.

the findings of past studies. We also compare our result with auto-parallelization tools like DiscoPoP, AutoPar and Pluto.

Hand-crafted Classifiers Fried et al. [32] evaluates traditional machine learning techniques including SVM, decision tree, and AdaBoost.

GNNs with Static Information Shen et al. [43] proposes a framework using a graph representation of the code with inst2vec. Their model is trained based on their dataset and tested with different benchmarks. Since their dataset is not public, we consider their results with the same benchmarks.

Neural Code Comprehension (NCC) architecture Ben-Nun et al. [5] propose the Neural Code Comprehension (NCC) model. NCC uses the inst2vec [5] embedding with two stacked LSTM. Each layer had 200 units and ReLU activation function. We used the NCC model with dense layer size of 16 and training batch size of 32.

D. Evaluation Results

The results in table III demonstrate that our approach outperforms the CNN+LSTM model, indicating that by taking both the structural and node analysis feature into account, our model understands the pattern and context of the code in a more comprehensive manner. The performance of our model also improves upon the results achieved by the SVM and Decision Tree methods implemented in Fried et al.'s work [32] while remaining slightly inferior to the AdaBoost method [32]. Despite this, our approach does not require hand-crafting and can be adapted to new applications rather than requiring detailed adjustment.

TABLE III
EVALUATION RESULTS WITH OUR APPROACH (SA-GNN) AND TOOLS LIKE DISCOPOP, PLUTO , AND AUTOPAR. IT SHOWS THAT OUR APPROACH OUTPERFORMS OTHER METHODS.

Benchmark	**Model/Tools**	**Acc**(%)
NPB	MV-GNN	**92.6**
	Static GNN	89.3
	SVM	85
	Decision Tree	85
	AdaBoost	92
	NCC	87.3
	Pluto	60.5
	AutoPar	74.8
	DiscoPoP	91.2
PolyBench	MV-GNN	**89.4**
	NCC	76.5
	Pluto	82.5
	AutoPar	76.7
	DiscoPoP	87.4
BOTS	MV-GNN	**82.9**
	NCC	72.4
	Pluto	60.5
	AutoPar	74.8
	DiscoPoP	78.9
Generated Dataset	MV-GNN	**88.7**
	NCC	62.9
	Pluto	60.5
	AutoPar	64.8
	DiscoPoP	80.1

We use the NPB dataset as a case study to better evaluate our approach. Table IV shows the results of our model's parallelism detection. Various factors contribute to incorrect classifications. False positive cases like loop 452 in IS.is are mostly caused by missing expert annotation in the original dataset False negative cases like loop line 53 in LU.setiv is because of the function call. Overall, our model's validity is demonstrated by the results.

TABLE IV
STATISTICS OF NPB DATASET TEST

Benchmark	Loops (#)	Identified Parallelizable Loops (#)
BT	184	176
SP	252	232
LU	173	163
IS	25	20
EP	10	9
CG	32	28
MG	74	68
FT	37	35
Total	787	731

We also assessed the performance of each view by putting

the output of each view into an LSTM layer, followed by a fully connected layer, in addition to the parallelism identification evaluation. The importance of the points of view is determined by their performance. For each benchmark, we set N_{multi}, N_{n}, N_{s} as the number of parallelism identified by our approach, the node feature view and the structural pattern view correspondingly. Then the importance of the view is represented as a normalized value $\text{IMP}_{\text{view}} = \frac{N_{\text{view}}}{N_{\text{multi}}}$ where N_{view} is either N_{n} or N_{s}. From figure 8, we can see that:

- The node feature view and the structural view in our model consensus well. The performance of the multi-view model outperforms any single view.
- For all three benchmark, the node feature view is more important.

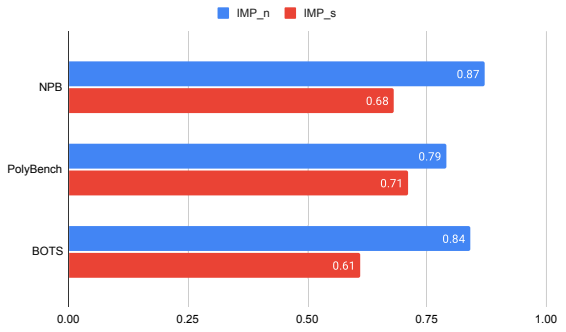

Fig. 8. Importance of views in our multi-view approach. IMP_n and IMP_s stands for the importance of the node feature view and the structure feature view, respectively.

We believe the different importance values come from the different characteristics of these benchmarks. The NPB benchmark consists largely DoALL loops and applications with simple parallelism. Meanwhile, the Polybench programs are all polyhedral loops with substantial structural patterns, while the BOTS benchmark programs all follow task parallelism.

V. Conclusion

In this paper, we propose a GNN-based multi-view learning strategy for discovering parallelism in sequential programs. Furthermore, we compare this approach to state-of-the-art techniques and demonstrate its competitiveness.

After integrating the PEG output from the DiscoPoP analysis phase, defining two views, a structural pattern view and a node feature view, the latent structural patterns extracted through anonymous random walks and the static and dynamic features can be combined. This combination of static and dynamic features expose coarse-grained parallelism opportunities static analysis alone would not detect. Compared to the CNN+LSTM network, our model realizes a 7.5% - 10.1% improvement in classification accuracy. By combining the output of two complementary yet distinct views, we achieve accuracies that rival other state-of-the-art approaches.

This novel multi-view implementation provides a promising future for automatic parallel region identification. With additional improvements in future works, we foresee this approach having significant potential to excel and exceed the performances of competitive methods. These advances would be done through the following additions:

- Modifying our resulting classification to specify distinct parallel patterns. By classifying the type of parallelism present in a region, parallelism frameworks can improve generated parallel code to further increase performance.
- Additional datasets for unsupervised model training. With more examples to expose our model to, the better our model can generalize the problem and provide improved predictions on new structures.
- We take advantage of dynamic features of the serial application since they supplement our model with additional information useful in prediction. This limits our choice of input data since it forces our input programs to link and execute to be dynamically analyzed. In future works we can depart from this assumption and decouple the dynamic and static features, allowing the model to selectively apply information from either method if it deems it beneficial. In doing so, our model would be applicable to a wider range of applications.

By addressing these areas for improvement, we expect further advancements in our model accuracies presented.

References

[1] M. J. Quinn, "Parallel programming," *TMH CSE*, vol. 526, p. 105, 2003.
[2] T. G. Mattson, "An introduction to openmp." in *ccgrid*, 2001, pp. 3–5.
[3] R. L. Graham, T. S. Woodall, and J. M. Squyres, "Open mpi: A flexible high performance mpi," in *International Conference on Parallel Processing and Applied Mathematics*. Springer, 2005, pp. 228–239.
[4] J. Alcaraz, S. Sleder, A. TehraniJamsaz, A. Sikora, A. Jannesari, J. Sorribes, and E. Cesar, "Building representative and balanced datasets of openmp parallel regions," in *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2021, pp. 67–74.
[5] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *arXiv preprint arXiv:1806.07336*, 2018.
[6] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.
[7] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, "Gnn explainer: A tool for post-hoc explanation of graph neural networks," *arXiv preprint arXiv:1903.03894*, 2019.
[8] K. Guu, J. Miller, and P. Liang, "Traversing knowledge graphs in vector space," *arXiv preprint arXiv:1506.01094*, 2015.
[9] W. Hamilton, P. Bajaj, M. Zitnik, D. Jurafsky, and J. Leskovec, "Embedding logical queries on knowledge graphs," *Advances in neural information processing systems*, vol. 31, 2018.
[10] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," *Advances in neural information processing systems*, vol. 31, 2018.
[11] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.
[12] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein, "Geometric deep learning on graphs and manifolds using mixture model cnns," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 5115–5124.
[13] M. Mao, J. Lu, G. Zhang, and J. Zhang, "Multirelational social recommendations via multigraph ranking," *IEEE transactions on cybernetics*, vol. 47, no. 12, pp. 4049–4061, 2016.
[14] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," *arXiv preprint arXiv:2007.04973*, 2020.

[15] S. Ivanov and E. Burnaev, "Anonymous walk embeddings," in *International conference on machine learning*. PMLR, 2018, pp. 2186–2195.

[16] R. Wismüller, *Loops, Parallel*. Boston, MA: Springer US, 2011, pp. 1079–1087. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_27

[17] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electronic notes in theoretical computer science*, vol. 217, pp. 5–21, 2008.

[18] T. Yuki, "Understanding polybench/c 3.2 kernels," in *International workshop on Polyhedral Compilation Techniques (IMPACT)*, 2014, pp. 1–5.

[19] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[20] 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.

[21] S. Garcia, D. Jeon, C. Louie, and M. B. Taylor, "The kremlin oracle for sequential code parallelization," *IEEE Micro*, vol. 32, no. 4, pp. 42–53, 2012.

[22] X. Zhang, A. Navabi, and S. Jagannathan, "Alchemist: A transparent dependence distance profiling infrastructure," in *2009 International Symposium on Code Generation and Optimization*. IEEE, 2009, pp. 47–58.

[23] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[24] Z. Li, R. Atre, Z. Ul-Huda, A. Jannesari, and F. Wolf, "Discopop: A profiling tool to identify parallelization opportunities," in *Tools for High Performance Computing 2014*. Springer, 2015, pp. 37–54.

[25] M. Norouzi, F. Wolf, and A. Jannesari, "Automatic construct selection and variable classification in OpenMP," in *Proc. of the International Conference on Supercomputing (ICS), Phoenix, AZ, USA*. ACM, Jun. 2019, pp. 330–341.

[26] Z. U. Huda, R. Atre, A. Jannesari, and F. Wolf, "Automatic parallel pattern detection in the algorithm structure design space," in *Proc. of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago, USA*. IEEE, May 2016, pp. 43–52.

[27] K. M. Borgwardt and H.-P. Kriegel, "Shortest-path kernels on graphs," in *Fifth IEEE international conference on data mining (ICDM'05)*. IEEE, 2005, pp. 8–pp.

[28] N. Shervashidze, S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt, "Efficient graphlet kernels for large graph comparison," in *Artificial intelligence and statistics*. PMLR, 2009, pp. 488–495.

[29] A. TehraniJamsaz, M. Khaleel, R. Akbari, and A. Jannesari, "Deeprace: A learning-based data race detector," in *International Workshop on Artificial Intelligence in Software Testing (AIST), Virtual Co-located with IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 1–8.

[30] A. Shabtai, Y. Fledel, and Y. Elovici, "Automated static code analysis for classifying android applications using machine learning," in *2010 international conference on computational intelligence and security*. IEEE, 2010, pp. 329–333.

[31] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[32] D. Fried, Z. Li, A. Jannesari, and F. Wolf, "Predicting parallelization of sequential programs using supervised learning," in *2013 12th International Conference on Machine Learning and Applications*, vol. 2. IEEE, 2013, pp. 72–77.

[33] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE international joint conference on neural networks*, vol. 2, no. 2005, 2005, pp. 729–734.

[34] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[35] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[36] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

[37] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in neural information processing systems*, vol. 29, 2016.

[38] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[39] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[40] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *arXiv preprint arXiv:1706.02216*, 2017.

[41] E. Park, J. Cavazos, and M. A. Alvarez, "Using graph-based program characterization for predictive modeling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 196–206.

[42] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *arXiv preprint arXiv:1812.08434*, 2018.

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

[44] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[45] Y. Jin, G. Song, and C. Shi, "Gralsp: Graph neural networks with local structural patterns," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 4361–4368.

[46] Q. Long, Y. Jin, G. Song, Y. Li, and W. Lin, "Graph structural-topic neural network," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 1065–1073.

[47] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[48] A. V. Phan, M. Le Nguyen, Y. L. H. Nguyen, and L. T. Bui, "Dgcnn: A convolutional neural network over large-scale labeled graphs," *Neural Networks*, vol. 108, pp. 533–543, 2018.

[49] H. Jin, M. Frumkin, and J. Yan, "The openmp implementation of nas parallel benchmarks and its performance," Citeseer, Tech. Rep., 1999.

[50] L.-N. Pouchet and T. Yuki, "Polybench: The polyhedral benchmark suite (version 4.2)," 2017.

[51] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *2009 international conference on parallel processing*. IEEE, 2009, pp. 124–131.