DyLGNN: Efficient LM-GNN Fine-tuning with Dynamic Node Partitioning, Low-degree Sparsity, and Asynchronous Sub-batch

Zhen Yu[12*], Jinhao Li[1*], Jiaming Xu[12], Shan Huang[1], Jiancai Ye[1], Ningyi Xu[1], Guohao Dai[12†]
[1]Shanghai Jiao Tong University, [2]Infinigence-AI, *Equal contributions
†Corresponding author: daiguohao@sjtu.edu.cn

***Abstract*—Text-Attributed Graphs (TAGs) tasks involve both textual node information and graph topological structure. The top-k method, using Language Models (LMs) for text encoding and Graph Neural Networks (GNNs) for graph processing, offers the best accuracy while balancing memory and training time. However, challenges still exist: (1) Static sampling of k neighbors reduces performance. Using a fixed k can result in sampling too few or too many nodes, leading to a 3.2% accuracy loss across datasets. (2) Time-consuming processing for non-trainable nodes. After partitioning all nodes into with-gradient trainable and without-gradient non-trainable sets, the number of non-trainable nodes is ∼9-10× larger than trainable nodes, resulting in nearly 70% of the total time. (3) Time-consuming data movement. For processing non-trainable nodes, after the text strings are tokenized into tokens on the CPU side, the data movement from host memory to GPU takes 30%-40% of the time.**

In this paper, we propose DyLGNN, an efficient end-to-end LM-GNN fine-tuning framework through three innovations: (1) Heuristic Node Partitioning. We propose an algorithm that dynamically and adaptively selects "important" nodes to participate in the training process for downstream tasks. Compared to the static top-k method, we reduce the training memory usage by 24.0%. (2) Low-Degree Sparse Attention. We point out that the embedding of low-degree nodes has minimal impact on the final results (*e.g.* ∼1.5% accuracy loss), therefore, We perform sparse attention computation on low-degree nodes to further reduce the computation caused by "unimportant" nodes, achieving an average 1.27× speedup. (3) Asynchronous Sub-batch Pipeline. Within the top-k framework, we analyze the time breakdown of the LM inference component. Leveraging our heuristic node partitioning, which effectively minimizes memory demands, we can asynchronously execute data movement and computation, thereby overlapping the time required for data movement. This improves GPU utilization and results in an average 1.1× speedup. We conduct experiments on several common graph datasets, and by combining the three methods mentioned above, DyLGNN achieves a 22.0% reduction in memory usage and a 1.3× end-to-end speedup compared to the top-k strategy.

I. Introduction

In recent years, Graph Neural Networks (GNNs) [1]–[3] have demonstrated increasingly advanced capabilities in handling graph-related tasks such as recommender systems [4], [5], social networks [6], [7], and citation graphs [8], [9]. Most of the graphs involved in these tasks contain textual information. These types of graphs are referred to as Text-Attributed Graphs (TAGs). Taking citation networks as an example, each node in the graph represents a paper, where the textual information of the node includes the paper's title and abstract. The traditional

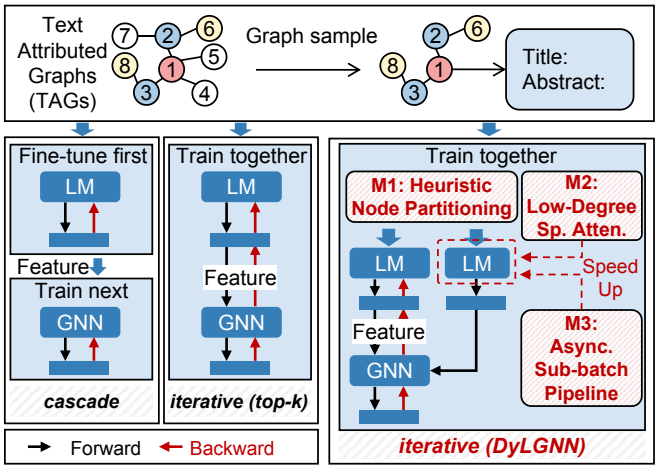

Fig. 1. DyLGNN overall training process compared to the previous works.

TAGs task pipeline can be divided into two steps. First, methods such as Bag-of-Words (BoW) [10] or Word2Vec [11] are used to process the raw textual information to obtain node features. Then, the message-passing mechanism of GNNs is employed to aggregate these features. Most recent works have focused on the latter, proposing various novel and powerful GNN architectures [12]–[14] to achieve better performance (*e.g.*, classification accuracy). However, the former, which involves the deeper extraction of textual information from TAGs, has often been overlooked.

With the rapid development of Language Models (LMs) such as BERT [15] and DeBERTa [16], they have demonstrated increasingly powerful performance in language understanding and generation [17], compared to methods like BoW or Word2Vec. Researchers have begun using LMs to replace traditional methods in processing textual information in TAGs, aiming for better downstream task performance. These works, which integrate LMs and GNNs, can be broadly categorized into two approaches: cascade [18]–[23] and iterative [24]–[29]. In the cascade approach, as shown on the left side of Fig. 1. Due to the separate training, the LM cannot perceive the graph's topological structure during training, leading to suboptimal performance. The iterative approach, on the other hand, as shown in the middle of Fig. 1, jointly trains the LM and GNN, allowing the LM to consider the graph's topological structure extracted by the GNN.

However, the existing top-k strategy, being a typical iterative

978-3-9826741-0-0/DATE25/© 2025 EDAA

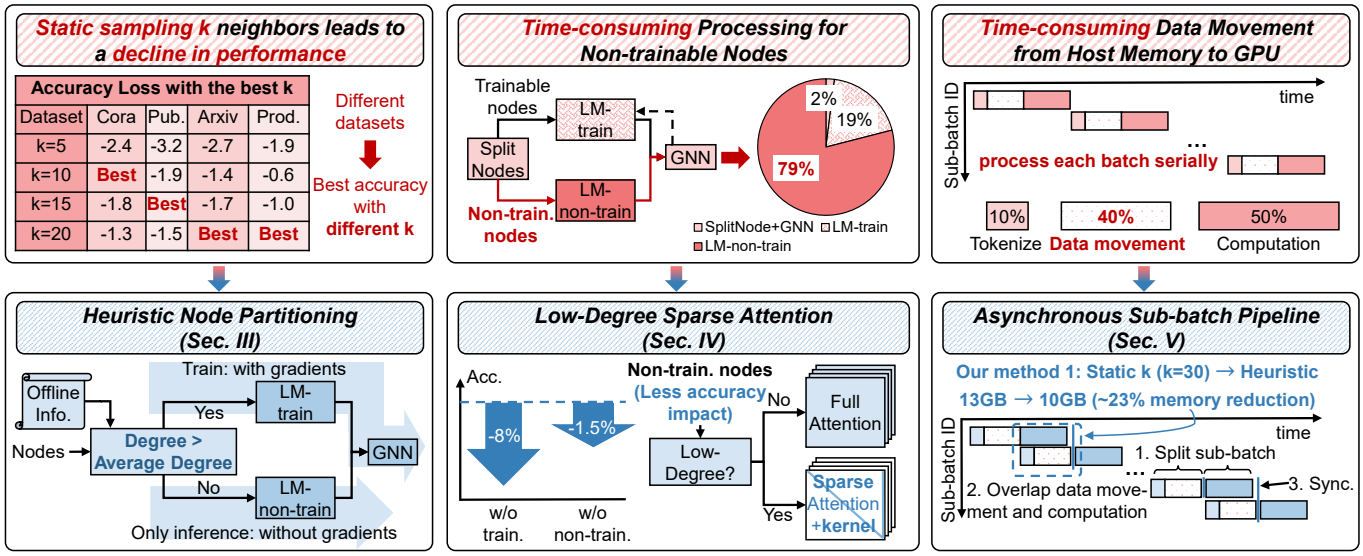

Fig. 2. Challenges: (1) static sampling k neighbors leads to a decline in performance, (2) time-consuming processing for non-trainable nodes, and (3) time-consuming data movement. We propose three novel contributions in DyLGNN: (1) heuristic node partitioning, (2) low-degree sparse attention, and (3) asynchronous sub-batch pipeline, to solve these challenges.

method, continues to pose significant challenges. **(1) Static sampling k neighbors leads to a decline in performance.** The neighbor explosion problem [30] in GNN message passing creates memory and computation challenges with the top-k strategy, as it requires generating features for both target nodes and their neighbors. Some works [27], [28] mitigate the OOM issue by fixing the number of nodes involved in backpropagation. However, using a fixed k window size across datasets does not optimize results, making k selection a key challenge. **(2) Time-consuming processing for non-trainable nodes.** The top-k strategy divides nodes into trainable and non-trainable categories. Non-trainable nodes, which are roughly ∼9-10× more numerous than trainable ones, only contribute to forward computation and account for nearly 70% of the total computation time. However, their impact on accuracy is much smaller than that of the trainable nodes (middle bottom of Fig. 2). **(3) Time-consuming data movement.** Existing works [26]–[28] handle non-trainable nodes by splitting them into sub-batches, where tokenization, data movement, and computation occur serially. As a result, the GPU waits for data from the i-th sub-batch before computation. Data movement takes up 30%-40% of the time in each sub-batch.

To address the aforementioned issues, we propose an efficient end-to-end fine-tuning method, which can be summarized by the following contributions:

(1) Heuristic Node Partitioning. We design a dynamic node partitioning algorithm that automatically segments the appropriate sets of trainable and non-trainable nodes based on different datasets. Compared to the top-k strategy, which often sets the k window size larger than necessary, our approach reduces memory requirements by an average of 22%.

(2) Low-Degree Sparse Attention. We find that the impact of non-trainable nodes on the final results is negligible, allowing us to perform sparse attention on these nodes. We further analyze the attention sparsity pattern of DeBERTa and implement an efficient GPU kernel based on this unique sparsity pattern, achieving a 1.27× speedup.

(3) Asynchronous Sub-batch Pipeline. During the sub-batch processing, due to our Heuristic Node Partitioning, we can asynchronously execute the data movement and computation stages, effectively overlapping the data movement time. This allows the GPU to compute the i-th sub-batch without waiting for data movement, thereby improving GPU utilization and achieving an average of 1.1× speedup.

We propose the DyLGNN and conduct experiments on multiple commonly used graph datasets. Compared to the top-k strategy, we achieve a 1.2% improvement in accuracy, reduce memory consumption by 22%, and obtain a 1.3× speedup in end-to-end training. We also conduct separate experiments on the three proposed methods, demonstrating their effectiveness.

II. Background

A. LMs integrated with GNNs

In recent years, many works have been proposed to solve the problem of LM being unable to consider graph topology information due to the cascading training of LM and GNN separately. LLMTTT [31] utilizes large language models (LLMs) to generate additional explanations and information for nodes that GNNs struggle to classify, aiming to enhance GNN classification accuracy. ENG [32] generates additional central virtual nodes for the entire graph based on node categories and connects these nodes as neighbors to others with the same classification. LOGIN [33], STAGE [34], E-LLaGNN [29], SimTeG [19] leverage LMs to generate enriched features based on the textual information of nodes, using these enhanced features as initial node representations, thereby harnessing the strong language understanding capabilities of LMs. Graphedit [35] employs LLMs to predict the strength of neighbor correlations in the original graph structure and modify the graph by adding or removing edges based on LLM

predictions. GPEFT [36], LLaGA [37], DGTL [38] transform GNN information into sequences and use LMs downstream to process these sequence-based representations derived from graphs. UniGraph [39], GALLON [40], GraphAdapter [41] do not use LMs or GNNs independently for downstream tasks; instead, they aim to align the feature spaces of both, achieving mutual enhancement of LM and GNN performance.

B. Efficient tuning end-to-end LM+GNN

We focus on the LM-GNN paradigm, a simple framework that seamlessly integrates LMs into the most widely used GNN training setups, and which has shown the best performance in our experimental configuration I, as illustrated in the middle of Fig. 1. Ioannidis et al. [28] are the first to propose a GNN training architecture that seamlessly incorporates LMs. LEADING [27] introduced two pipelines: Pipeline 1 processes target nodes with gradient-based computation, while Pipeline 2 performs inference on nodes without gradients. E-LLaGNN [29] enhances target nodes by sampling their top-k neighbors using an LLM. These works address the memory issues arising from integrating LMs into GNN training frameworks by fixing the number of nodes participating in training. And many works [42]–[44] are proposed to accelerate LM computing. However, limiting training participation to target nodes or a fixed top-k sample is not suitable for different graph datasets, as shown in the upper left of Fig. 2.

III. Heuristic Node Partitioning

A. Challenge

Due to the neighborhood explosion problem [30] in GNNs, simply using an LM to encode both the target node and its neighbors during GNN training requires enormous memory. Previous works mostly adopt a top-k strategy, selecting k neighbors of the target node with higher degrees to participate in the backpropagation, while the remaining nodes only need to go through forward propagation to generate embeddings that are consistent within the same embedding space. However, different graph datasets show varying distributions, and the randomness in subgraph sampling causes each sampled subgraph to be different. Using a fixed window size k result in sampling more or fewer nodes than necessary, leading to increased memory consumption and suboptimal training accuracy (Fig. 2 left top).

B. Motivation and Insight

The degree of a node often reflects its importance within the graph—the higher the degree, the more central the node is and the more information it contains. **Our key insight is that, among a target node's neighbors, those with higher degrees exert a greater influence on the target node.**

C. Approach

Our training algorithm has two parts: node partitioning and end-to-end training. **Node Partitioning:** This algorithm identifies nodes with significant encoding changes during training, which impact the final results. **End-to-End Training:** This part integrates the LM into the GNN training framework, preserving GNN's core structure while leveraging the LM's efficient text extraction to improve overall training performance.

Algorithm 1 end-to-end LM+GNN train Algorithm

Input: Graph $G \in \{V, S, E\}$
Output: Fine-Tuned LM+GNN
1: **for** each sub-graph $G_{sub} \in \{V_{sub}, S_{sub}, E_{sub}\}$ in G **do**
2: $V_1, V_2 = splitNode(V_{sub}, degT, maxNode)$
3: $S_1 \leftarrow$ select text information from S_{sub} by V_1
4: $X_1 = LM(S_1)$ (with gradients)
5: cache $\leftarrow X_1(detached)$
6: **if** V_2 in cache **then**
7: $X_2 \leftarrow$ cache
8: **else**
9: $S_2 \leftarrow$ select text information from S_{sub} by V_2
10: $X_2 = LM(S_2)$ (without gradients)
11: cache $\leftarrow X_2$
12: **end if**
13: $X = Concat(X_1, X_2)$
14: $y_{pred} = GNN(X, G_{sub})$
15: Compute loss and backward
16: **end for**
17: **def** $splitNode(V_{sub}, degT, maxNode)$:
18: add V_{target} node to V_1, remove V_{target} from V_{sub}
19: sort node set V_{sub} by degree
20: **for** $i = 0, i < maxNode$ and i-th node degree $> degT$ **do**
21: add i-th node in V_{sub} to V_1
22: **end for**
23: add remain node to V_2
24: **return** V_1, V_2

Node partitioning: Based on the above analysis, we propose Algorithm 1 line 17 for partitioning node sets. This algorithm takes as input the V_{sub} to be partitioned, the node degree threshold $degT$, and the maximum number of trainable nodes allowed $maxNode$. The steps are as follows: ①**Initialization:** Add the target nodes V_{target} to V_1 (line 19). These nodes are included in the training set, regardless of their degree, as we must ensure that the LM considers their influence on the final classification result. ②**Reordering:** Reorder V_{sub} based on node degrees, placing nodes with higher degrees at the front (line 20). ③**Node Selection:** Traverse the reordered set, adding nodes with degrees above $degT$ to V_1 (line 22), ensuring the total number of nodes in V_1 does not exceed $maxNode$ (line 21). This ensures that nodes exceeding $maxNode$ are excluded from gradient consideration, enabling dynamic and adaptive node partitioning for various sampling subsets. ④**Final Partitioning:** Place the remaining nodes into V_2 (line 24). The LM performs inference on these nodes without considering their gradients from back-propagation. This approach ensures efficient memory and resource management while adapting dynamically to different node subsets.

End-to-End Training: The basic training framework for large-scale GNNs, shown in Algorithm 1, includes sampling (line 1), GNN forward pass (line 14), and backward training (line 15). DyLGNN integrates seamlessly with this framework (lines 2–13), as follows: ①**Node Partitioning:** First, use line 17 to partition the sampled subset of nodes into V_1 and V_2

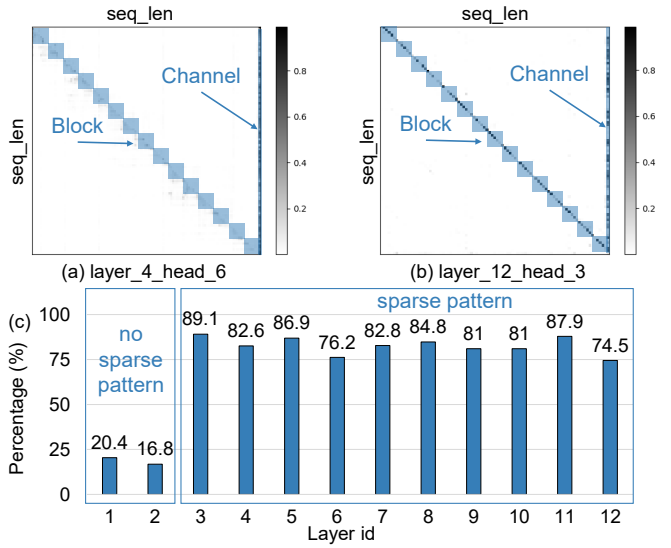

Fig. 3. Sparse pattern of different layers and heads in DeBERTa. (a) Head 6 in layer 4. (b) Head 3 in layer 12. (c) Percentage of dense elements contained in sparse pattern per layer.

(line 2). For nodes in V_1, retrieve their corresponding textual information (line 3), and then use the LM to perform with-gradient forward encoding of this textual information (line 5). Nodes in V_1 might include some with lower degrees that should ideally be in V_2. However, due to being target nodes in this batch, they are included in V_1. Therefore, cache these nodes' embedding representations for future batches (line 6). ②**LM Inference:** For nodes in V_2, check if their embeddings are cached (line 6). If so, retrieve them directly (line 7). For first-time sampled nodes, perform non-gradient forward computation with the LM to obtain and cache their embeddings (lines 10–11). ③**Feature Aggregation:** Combine features from V_1 and V_2 to form the mini-batch features (line 13) and perform the GNN forward pass for final classification results (line 14). ④**Backward Training:** DyLGNN uses the GNN classification loss for backpropagation (line 15). This method allows efficient integration of LM embeddings with GNN training while managing computational resources effectively.

IV. Low-Degree Sparse Attention

A. Challenge

After classifying the graph nodes into forward and inference nodes using line 17, the number of forward nodes is limited to under $maxNode$, reducing storage overhead during training. Although inference nodes don't require backpropagation and don't add extra storage overhead, their large number (about 9-10 times that of forward nodes) results in them consuming nearly 70% of the computation time. Therefore, reducing the computation time for inference nodes is key to further accelerating model training.

B. Motivation and Insight

During training, the embeddings of inference nodes are processed by the transformer with a fixed input sequence length of 2048. Due to the quadratic complexity of the attention mechanism, this part of the computation accounts for over 50% of the overall inference node processing. As shown in Fig. 2 middle bottom, only reserving trainable nodes for training leads to 8% accuracy loss while only reserving non-trainable nodes leads to 1.5% loss on Cora dataset. It indicates that the non-trainable nodes show less impact on accuracy but deleting them completely is unacceptable. Therefore, **our key insight is to further partition inference nodes and apply sparse attention to non-trainable nodes with lower degrees to reduce computational load**.

Algorithm 2 Sparse Attention GPU Kernel

Input: Query matrix $Q \in R^{N\times d}$, Key matrix $K \in R^{N\times d}$, Value matrix $V \in R^{N\times d}$, sequence length $seqL$, Block size bs.
Output: Attention output O

1: Divide Q into $T_r = \lceil \frac{seqL}{bs} \rceil$ blocks $Q_1, ..., Q_{T_r}$ of size $bs \times d$ each. K and V as same.
2: **for** $1 \le i \le T_r$ **do**
3: Load Q_i, K_i, V_i, last row of K, K_{seqL} and last row of V, V_{seqL} to on-chip SRAM.
4: initialize $O_i = (0)_{bs\times d} \in R^{bs\times d}$.
5: compute $S_i = Q_i K_i^T \in R^{bs\times bs}$.
6: compute $S_{seqL} = Q_i K_{seqL}^T \in R^{bs\times 1}$.
7: compute $P_i, P_{seqL} = softmax(S_i, S_{seqL})$.
8: compute $O_i = P_i V_i \in R^{bs\times d}, O_{seqL} = P_{seqL} V_{seqL} \in R^{bs\times d}$.
9: $O_i = O_i + O_{seqL}$.
10: Write O_i to i-th block of O.
11: **end for**

Output: O

C. Approach

First, we analyze the attention sparsity pattern of DeBERTa, which serves as the LM component of DyLGNN. As shown in Fig. 3, Fig. 3 (a) illustrates the sparsity pattern of the 6th head in the 4th Transformer layer, while (b) presents the sparsity pattern of the 3rd head in the 12th layer. This sparsity pattern predominantly focuses on the diagonal and the last column, indicating that each token primarily acquires local information from its neighboring tokens and global information from the last token. We also observed a phenomenon where the sparsity becomes more concentrated in later layers, which has impacted our implementation: the first two layers do not use sparse masks, while the subsequent layers employ a mask with a block size of 16.

Based on the unique sparse pattern of DeBERTa, we design a GPU kernel, as detailed in Algorithm 2, to compute sparse attention. Our kernel implementation draws inspiration from Flash Attention [45]. In addition to accepting Q, K, and V as inputs, it also requires a variable $seqL$ to represent the length of the current sentence, and a block size bs, which is set to 16 in DyLGNN. Similar to Flash Attention, we first divide Q, K and V into T_r blocks and process these blocks in parallel (line 1). Since the sparse pattern only requires computation along the diagonal and the last column, for the i-th thread, it is only necessary to load Q_i, K_i, V_i and the K_{seqL} and V_{seqL} blocks needed for the last column of the attention computation (line 2). Subsequently, we compute the attention scores for the diagonal (line 5) and for the last column (line 6). These two sets of

results are then subjected to a $softmax$ operation to obtain the corresponding P matrices (line 7). Finally, we use V_i and V_{seqL} to compute the final output O_i and write it back (lines 8, 9, and 10).

V. Asynchronous Sub-batch Pipeline

A. Challenge

Training graph neural networks relies on a heterogeneous system consisting of CPU, GPU, and memory. Typically, text graph datasets are stored in the host memory and need to be transferred to the GPU memory before computation. Through profiling, we find that during the inference path of the training process, after the text strings are tokenized into tokens on the CPU side, transferring the data for different datasets takes between 30%-40% of the time. Furthermore, only after the data transfer can subsequent computations be performed. Therefore, how to reduce the data transfer time from host memory to GPU memory is a challenging problem.

B. Motivation and Insight

By applying our heuristic node partitioning instead of top-k, the memory consumption is decreased by 23% (from ∼13GB to ∼10GB), which reserves the possibility of trading memory for computation for sub-batches. Furthermore, we find that in the heterogeneous training hardware system, the data transfer operation from the CPU host memory to the GPU memory is independent of the GPU's computational operation. Furthermore, the sub-batch computation mentioned in Section III is easier to exploit the independence. Therefore, **our key insight is to parallelize data transfer and data computation between sub-batches to reduce the data transfer overhead**.

C. Approach

The non-trainable nodes are divided into sub-batches of 32 nodes each and each sub-batch is processed serially by the same thread, as shown in Fig. 4 top. To further minimize the processing time of each sub-batch, we propose an asynchronous pipeline. As shown in Fig. 4, each sub-batch includes tokenization, data loading, and computation. The time spent on tokenization and data loading accounts for nearly 50%. We adopt a two-thread parallel processing approach: one thread handles tokenization and data loading for the next sub-batch, while the other thread performs computation on the current sub-batch. Specifically, one thread tokenizes the text string and then copies it from host memory to GPU memory, placing it into a First-In-First-Out (FIFO) queue. This thread then repeats this process. The other thread, once finding that the queue is not empty, retrieves data sequentially for computation. Setting the FIFO queue depth to 1 is sufficient to achieve simple synchronization and significantly reduce the processing time for each sub-batch.

VI. Experiment

A. Experimental Setup

We implement DyLGNN on an Intel Xeon Gold 6138 CPU at 2GHz and one NVIDIA GeForce RTX 3090 GPU. The software

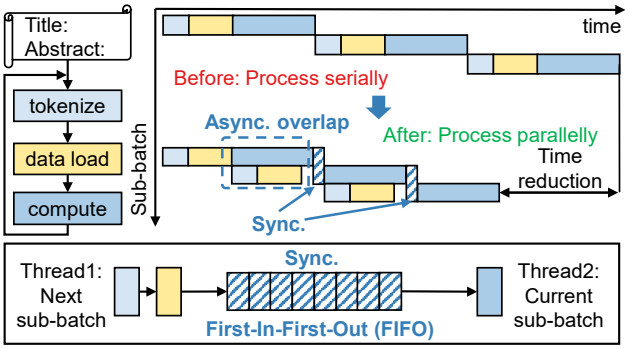

Fig. 4. Asynchronous sub-batch pipeline with FIFO synchronization.

environment includes CUDA Driver 12.1, PyTorch v2.2.2, PyG v2.5.2, and transformers v4.41.2.

Dataset. The datasets used in our experiments include Cora, PubMed [46], OGBN-Arxiv, OGBN-Products [9], and Arxiv_2023. Since Cora and PubMed do not provide textual information in PyG, we use the text provided by TAPE [22]. For OGBN-Arxiv and OGBN-Products, we also use TAPE's configuration, working with subgraphs instead of full graphs.

Baseline. In the cascade approach, we select TAPE and SimTeG [19] as representatives, and in the iterative approach, we choose the top-k strategy. Both TAPE and SimTeG first fine-tune an LM and then input the node features generated by the LM into the GNN. The difference lies in that TAPE employs full-parameter fine-tuning, while SimTeG uses LoRA for fine-tuning. top-k combines the ideas of E-LLaGNN [29] and LEADING [27] by selecting a fixed number of nodes during the partitioning of trainable nodes. In our accuracy experiments, we select the highest accuracy for each dataset, while in the ablation study, the k window size is fixed at 30.

Implementation. For the detailed implementation, we use the following configurations: All GNN models are configured with GCN [1], using 3 layers and 64 hidden dimensions, sourced from PyG. For LM, TAPE chooses DeBERTa, and SimTeG chooses e5-large. DyLGNN utilize a combination of DeBERTa and GCN.

Metrics. We perform node classification tasks on these datasets, aiming to determine the category of each node. Classification accuracy is used as the evaluation metric.

B. Accuracy Evaluations

With baselines. Table I illustrates the comparison of classification accuracy between DyLGNN and other types of baseline methods. It can be observed that the iterative method achieves better performance than the cascade method. This demonstrates that incorporating the graph topology information captured by the GNN into the LM enhances the model's accuracy. DyLGNN outperforms the cascade method by 2.8% in terms of accuracy. Compared to the top-k strategy, which also uses an iterative approach, DyLGNN dynamically samples the number of neighbors and can adaptively find the optimal window size k across different datasets. Additionally, it can dynamically adjust during the training process, which static strategies cannot achieve, resulting in a 1.2% improvement in accuracy.

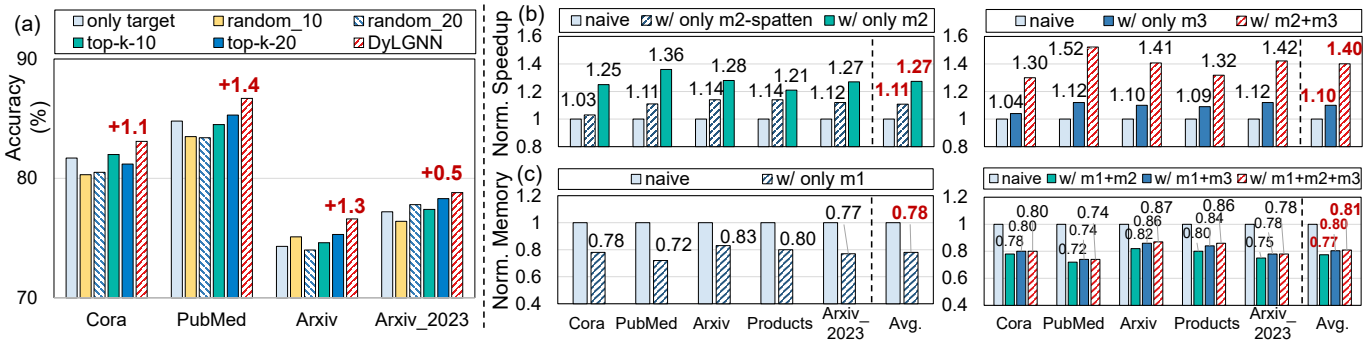

Fig. 5. Ablation study in DyLGNN. (a) The accuracy of different strategies. (b) Normalized speedup. (c) Normalized memory consumption.

TABLE I
ACCURACY COMPARISON WITH BASELINES

Accuracy(%)(↑)	Cascade		Iterative	
Dataset	TAPE [22]	SimTeG [19]	top-k	DyLGNN
Cora	80.2	80.8	82.0	**82.8**
PubMed	83.1	84.2	85.1	**87.7**
Arxiv	**76.8**	74.3	75.5	76.6
Products	78.5	77.4	79.8	**81.2**
Arxiv_2023	77.8	75.2	78.1	**78.3**
Average	79.2	78.3	80.1	**81.3**

Further discussion. Fig. 5 (a) shows the accuracy results of our dynamic node partitioning, only target, random selection, and top-k methods on multiple datasets. The random selection strategy always has a negative impact on accuracy because there are only a few nodes in the target node's neighbors that play an important role, and most other nodes may have minimal or even harmful effects on the target node. Random selection has a greater probability of selecting these "harmful" nodes, leading to performance degradation. The top-k strategy selects the k nodes with the highest degree among the neighbors of the target node. However, this fixed window size is not applicable in all cases. In some cases, there may not be enough "important" nodes around the target node, and other "harmful" nodes may be sampled, resulting in accuracy degradation like random selection. DyLGNN can dynamically adjust the size of the k-window without allowing unnecessary nodes to affect the accuracy, thus showing the generalization for different datasets.

C. Efficiency Evaluations

Total time. Table II illustrates the performance advantages of DyLGNN over the baseline on the Arxiv dataset. DyLGNN accelerates training by 1.2× speedup. This improvement mainly stems from our low-degree sparse attention and asynchronous sub-batch pipeline. The low-degree sparse attention reduces the computational load during the LM inference phase, which typically accounts for 70% of the forward time. The asynchronous sub-batch pipeline further enhances efficiency by overlapping data movement with computation, effectively hiding data loading time.

Memory. Compared to topk-30, DyLGNN reduces memory consumption by 22%, primarily due to our heuristic node partitioning. This algorithm dynamically adjusts the number of nodes in V_1, eliminating the need to meet a fixed window size, which reduces the number of nodes involved in training and, consequently, the memory required.

TABLE II
COMPARISON OF MEMORY CONSUMPTION AND TOTAL TIME

Methods	**Updated parameters**	**Memory (GB)**	**Total time**
SimTeG	2,013,328	3.8	25.3h
topk-30	138,660,011	13.3	21.5h
DyLGNN	138,660,011	10.5	16.5h

Split Node Overhead. The additional overhead introduced by heuristic node partitioning is minimal. As shown in Fig. 2 middle top, it occupies only 2% of the total forward time.

D. Ablation Study

Speedup. Fig. 5 (b) presents the speedup of our low-degree sparse attention (m2) and asynchronous sub-batch pipeline (m3). The naive chart is only applied the heuristic node partitioning (m1) without other optimizations. With m2-spatten, it achieves 1.11× speedup, while by incorporating the fused CUDA kernel it raises to 1.27×. With only m3, it achieves 1.1× speedup. By integrating both m2 and m3, DyLGNN ultimately achieves an average 1.40× end-to-end speedup.

Memory reduction. Fig. 5 (c) demonstrates the memory reduction by applying different methods. Due to our dynamic node partitioning algorithm, we effectively reduce the window size k compared to the fixed top-k strategy (k=30), achieving a 22% memory reduction. By applying all methods, we achieve 21% memory reduction because m3 brings a little bit of memory increment for each sub-batch.

VII. CONCLUSION

We propose DyLGNN, an efficient end-to-end LM-GNN fine-tuning framework. We first propose heuristic node partitioning to select nodes to participate in the training process adaptively. Then, we propose low-degree sparse attention for "unimportant" nodes. Last, we propose an asynchronous sub-batch pipeline to overlap the data movement and computation. Experimental results show DyLGNN achieves higher accuracy with 24% memory reduction and a 1.3× end-to-end speedup compared to the baseline top-k strategy.

VIII. ACKNOWLEDGEMENTS

This work was sponsored by the National Natural Science Foundation of China (No. 62104128, U21B2031), Shanghai Rising-Star Program (No. 24QB2706200).

REFERENCES

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

[2] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[3] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[4] C. Ma, L. Ma, Y. Zhang, J. Sun, X. Liu, and M. Coates, "Memory augmented graph neural networks for sequential recommendation," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 5045–5052.

[5] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The world wide web conference*, 2019, pp. 417–426.

[6] T. Bian, X. Xiao, T. Xu, P. Zhao, W. Huang, Y. Rong, and J. Huang, "Rumor detection on social media with bi-directional graph convolutional networks," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 01, 2020, pp. 549–556.

[7] C. Li and D. Goldwasser, "Encoding social information with graph convolutional networks forpolitical perspective detection in news media," in *Proceedings of the 57th annual meeting of the association for computational linguistics*, 2019, pp. 2594–2604.

[8] Z. Yang, W. Cohen, and R. Salakhudinov, "Revisiting semi-supervised learning with graph embeddings," in *International conference on machine learning*. PMLR, 2016, pp. 40–48.

[9] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *Advances in neural information processing systems*, vol. 33, pp. 22 118–22 133, 2020.

[10] Y. Zhang, R. Jin, and Z.-H. Zhou, "Understanding bag-of-words model: a statistical framework," *International journal of machine learning and cybernetics*, vol. 1, pp. 43–52, 2010.

[11] T. Mikolov, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[12] I. Spinelli, S. Scardapane, and A. Uncini, "Adaptive propagation graph convolutional network," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 10, pp. 4755–4760, 2020.

[13] G. Li, G. Qian, I. C. Delgadillo, M. Muller, A. Thabet, and B. Ghanem, "Sgas: Sequential greedy architecture search," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 1620–1630.

[14] C. Xu, Z. Cui, X. Hong, T. Zhang, J. Yang, and W. Liu, "Graph inference learning for semi-supervised classification," *arXiv preprint arXiv:2001.06137*, 2020.

[15] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[16] P. He, X. Liu, J. Gao, and W. Chen, "Deberta: Decoding-enhanced bert with disentangled attention," *arXiv preprint arXiv:2006.03654*, 2020.

[17] J. Li, J. Xu, S. Huang, Y. Chen, W. Li, J. Liu, Y. Lian, J. Pan, L. Ding, H. Zhou *et al.*, "Large language model inference acceleration: A comprehensive hardware perspective," *arXiv preprint arXiv:2410.04466*, 2024.

[18] X. Huang, K. Han, D. Bao, Q. Tao, Z. Zhang, Y. Yang, and Q. Zhu, "Prompt-based node feature extractor for few-shot learning on text-attributed graphs," *arXiv preprint arXiv:2309.02848*, 2023.

[19] K. Duan, Q. Liu, T.-S. Chua, S. Yan, W. T. Ooi, Q. Xie, and J. He, "Simteg: A frustratingly simple approach improves textual graph learning," *arXiv preprint arXiv:2308.02565*, 2023.

[20] H. Xie, D. Zheng, J. Ma, H. Zhang, V. N. Ioannidis, X. Song, Q. Ping, S. Wang, C. Yang, Y. Xu *et al.*, "Graph-aware language model pre-training on a large graph corpus can help multiple graph applications," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5270–5281.

[21] H. Liu, J. Feng, L. Kong, N. Liang, D. Tao, Y. Chen, and M. Zhang, "One for all: Towards training one graph model for all classification tasks," *arXiv preprint arXiv:2310.00149*, 2023.

[22] X. He, X. Bresson, T. Laurent, A. Perold, Y. LeCun, and B. Hooi, "Harnessing explanations: Llm-to-lm interpreter for enhanced text-attributed graph representation learning," *arXiv preprint arXiv:2305.19523*, 2023.

[23] W. Wei, X. Ren, J. Tang, Q. Wang, L. Su, S. Cheng, J. Wang, D. Yin, and C. Huang, "Llmrec: Large language models with graph augmentation for recommendation," in *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*, 2024, pp. 806–815.

[24] J. Zhao, M. Qu, C. Li, H. Yan, Q. Liu, R. Li, X. Xie, and J. Tang, "Learning on large-scale text-attributed graphs via variational inference," *arXiv preprint arXiv:2210.14709*, 2022.

[25] E. Chien, W.-C. Chang, C.-J. Hsieh, H.-F. Yu, J. Zhang, O. Milenkovic, and I. S. Dhillon, "Node feature extraction by self-supervised multi-scale neighborhood prediction," *arXiv preprint arXiv:2111.00064*, 2021.

[26] Y. Zhu, Y. Wang, H. Shi, and S. Tang, "Efficient tuning and inference for large language models on textual graphs," *arXiv preprint arXiv:2401.15569*, 2024.

[27] R. Xue, X. Shen, R. Yu, and X. Liu, "Efficient large language models fine-tuning on graphs," *arXiv preprint arXiv:2312.04737*, 2023.

[28] V. N. Ioannidis, X. Song, D. Zheng, H. Zhang, J. Ma, Y. Xu, B. Zeng, T. Chilimbi, and G. Karypis, "Efficient and effective training of language and graph neural network models," *arXiv preprint arXiv:2206.10781*, 2022.

[29] A. Jaiswal, N. Choudhary, R. Adkathimar, M. P. Alagappan, G. Hiranandani, Y. Ding, Z. Wang, E. W. Huang, and K. Subbian, "All against some: Efficient integration of large language models for message passing in graph neural networks," *arXiv preprint arXiv:2407.14996*, 2024.

[30] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," *arXiv preprint arXiv:1907.04931*, 2019.

[31] J. Zhang, Y. Wang, X. Yang, S. Wang, Y. Feng, Y. Shi, R. Ren, E. Zhu, and X. Liu, "Test-time training on graphs with large language models (llms)," *arXiv preprint arXiv:2404.13571*, 2024.

[32] J. Yu, Y. Ren, C. Gong, J. Tan, X. Li, and X. Zhang, "Empower text-attributed graphs learning with large language models (llms)," *arXiv preprint arXiv:2310.09872*, 2023.

[33] Y. Qiao, X. Ao, Y. Liu, J. Xu, X. Sun, and Q. He, "Login: A large language model consulted graph neural network training framework," *arXiv preprint arXiv:2405.13902*, 2024.

[34] A. Zolnai-Lucas, J. Boylan, C. Hokamp, and P. Ghaffari, "Stage: Simplified text-attributed graph embeddings using pre-trained llms," *arXiv preprint arXiv:2407.12860*, 2024.

[35] Z. Guo, L. Xia, Y. Yu, Y. Wang, Z. Yang, W. Wei, L. Pang, T.-S. Chua, and C. Huang, "Graphedit: Large language models for graph structure learning," *arXiv preprint arXiv:2402.15183*, 2024.

[36] Q. Zhu, D. Zheng, X. Song, S. Zhang, B. Jin, Y. Sun, and G. Karypis, "Parameter-efficient tuning large language models for graph representation learning," *arXiv preprint arXiv:2404.18271*, 2024.

[37] R. Chen, T. Zhao, A. Jaiswal, N. Shah, and Z. Wang, "Llaga: Large language and graph assistant," *arXiv preprint arXiv:2402.08170*, 2024.

[38] Y. Qin, X. Wang, Z. Zhang, and W. Zhu, "Disentangled representation learning with large language models for text-attributed graphs," *arXiv preprint arXiv:2310.18152*, 2023.

[39] Y. He and B. Hooi, "Unigraph: Learning a cross-domain graph foundation model from natural language," *arXiv preprint arXiv:2402.13630*, 2024.

[40] J. Xu, Z. Wu, M. Lin, X. Zhang, and S. Wang, "Llm and gnn are complementary: Distilling llm for multimodal graph learning," *arXiv preprint arXiv:2406.01032*, 2024.

[41] X. Huang, K. Han, Y. Yang, D. Bao, Q. Tao, Z. Chai, and Q. Zhu, "Can gnn be good adapter for llms?" in *Proceedings of the ACM on Web Conference 2024*, 2024, pp. 893–904.

[42] J. Li, S. Li, J. Xu, S. Huang, Y. Lian, J. Liu, Y. Wang, and G. Dai, "Enabling fast 2-bit llm on gpus: Memory alignment, sparse outlier, and asynchronous dequantization," *arXiv preprint arXiv:2311.16442*, 2023.

[43] J. Li, S. Huang, J. Xu, J. Liu, L. Ding, N. Xu, and G. Dai, "Marca: Mamba accelerator with reconfigurable architecture," *arXiv preprint arXiv:2409.11440*, 2024.

[44] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, Y. Dong, Y. Wang *et al.*, "Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 148–161, 2024.

[45] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with io-awareness," *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.

[46] Z. Yang, W. Cohen, and R. Salakhudinov, "Revisiting semi-supervised learning with graph embeddings," in *International conference on machine learning*. PMLR, 2016, pp. 40–48.