Distributed Graph Neural Network Inference With Just-In-Time Compilation For Industry-Scale Graphs

Xiabao Wu, Yongchao Liu, Wei Qin, Chuntao Hong
Ant Group, China
{wuxiabao.wxb,yongchao.ly,johnny.qw,chuntao.hct}@antgroup.com

Graph neural networks (GNNs) have delivered remarkable results in various fields. However, the rapid increase in the scale of graph data has introduced significant performance bottlenecks for GNN inference. Both computational complexity and memory usage have risen dramatically, with memory becoming a critical limitation. Although graph sampling-based subgraph learning methods can help mitigate computational and memory demands, they come with drawbacks such as information loss and high redundant computation among subgraphs. This paper introduces an innovative processing paradgim for distributed graph learning that abstracts GNNs with a new set of programming interfaces and leverages Just-In-Time (JIT) compilation technology to its full potential. This paradigm enables GNNs to highly exploit the computational resources of distributed clusters by eliminating the drawbacks of subgraph learning methods, leading to a more efficient inference process. Our experimental results demonstrate that on industry-scale graphs of up to **500 million nodes and 22.4 billion edges**, our method can produce a performance boost of up to **27.4 times**.

1 Introduction

As the graph data scale increases exponentially, GNN inference is encountering significant performance bottlenecks. The vast number of nodes and edges in large-scale graphs substantially amplifies core computational tasks such as graph convolution, message passing, and aggregation, leading to a notable decrease in inference speed. In addition, handling large-scale graph data requires substantial memory. Surpassing memory limits can result in memory overflow, compromising system stability and availability.

To mitigate these issues, practitioners often utilize graph sampling-based subgraph learning techniques [5], which involve selecting specific nodes and edges to create mini-batches of subgraphs for inference, thereby curtailing computational and memory demands. However, it has some notable drawbacks, such as causing information loss, decreasing inference accuracy, and introducing redundant computational overhead among subgraphs.

Moreover, JIT (Just-In-Time) compilation technology offers distinct advantages for enhancing inference acceleration and deployment. It compiles code into machine code during runtime, merging the benefits of interpretive execution with those of static compilation. In deep learning, JIT has enhanced the training and inference performance of conventional neural networks by operation fusion, storage and overhead reduction, and code optimization, thereby providing novel solutions for graph learning inference.

In this study, we introduce a novel distributed processing paradigm along with a new set of programming interfaces for GNN inference that overcomes the limitations of conventional subgraph learning methods. Our paradigm eliminates the need for subgraph extraction operations during inference on large-scale graph learning models, theoretically enabling the handling of graph data of any size. The key concept involves breaking down complex GNN models into multiple simple deep learning modules that function independently without requiring machine communication. By harnessing the full potential of JIT compilation technology, we can produce highly efficient executable programs, facilitating easy deployment and optimized performance.

2 Methods

Our paradigm is implemented based on DFOGraph [4] (integrated into our graph intelligent computing system [2] deployed in production), but introduces moderate modifications allowing for acquiring batches of dense data during processing, particularly in message passing. We introduce a new set of programming interfaces, including two processing interfaces and two data retrieval functions for node and edge features. In theory, our paradigm can be incorporated into any distributed graph processing engine.

Processing interfaces are **transform** and **message_passing**, while data retrieval functions include **get_vertex** and **get_edge**. **transform** is used mainly to process node or edge data locally, while **message_passing** deals with the logic involved in remote message passing. Notably, unlike existing frameworks such as PyG and DGL, users do not need to concern themselves with how messages are transmitted through edges during actual implementation, and also do not need to manually write or invoke functions related to message passing concerning edges. Instead, they are automatically managed by the runtime. We implement our JIT compilation with `torch.jit`. Upon decomposition, the graph algorithms convert sparse graph data into dense matrices during data processing, thus further improving inference performance.

Programming Interfaces and JIT Compilation To clarify our design rationale, we have chosen some specific steps from the `GATConv` algorithm [3] in PyG [1] as an example.

The following code snippet shows how the module performs linear projection on node features and aggregates them into new features based on edges. We will elaborate the programming interfaces and methodologies below.

```
1   import torch
2   from torch.nn.linear import Linear
3
4   class GATConvVertex(torch.nn.Module):
5       def __init__(...):
6         self.lin = Linear(...)
7         self.att_src = Parameter(torch.empty(...))
8         self.att_dst = Parameter(torch.empty(...))
9         ...
10      def forward(self, x):
11        ...
12        x_src = x_dst = self.lin(x)
13        alpha_src = (x_src * self.att_src).sum(-1)
14        alpha_dst = (x_dst * self.att_dst).sum(-1)
15        return alpha_src, alpha_dst
16
17  class GATConvMP(torch.nn.Module):
18      def __init__(...):
19          ...
20      def forward(self, src: Tensor, dst: Tensor):
21          alpha = dst + src
22          alpha = F.leaky_relu(alpha)
23          return alpha
24
25  vertexmodel = GATConvVertex(...)
26  mpmodel = GATConvMP(...)
27  x = get_vertex("input")
28  alpha_src, alpha_dst = transform(x, vertexmodel)
29  alpha = message_passing(alpha_src, alpha_dst, mpmodel)
```

- **get_vertex / get_edge**: enables users to access the relevant data for processing based on the feature name. This interface merely returns a reference to the data and does not carry out any actual read or write operations (line 27).
- **transform**: is primarily utilized for independently processing the features of individual nodes or edges. This interface does not require message passing and can be executed locally on each distributed compute node. It takes as input the features of the nodes or edges that are to be processed and the associated processing logic (line 28).
- **message_passing**: With this interface, users don't have to consider specific aggregation methods like `concat`, add, or others. Its message passing is implicitly handled through the GAS model [4], operating transparently so that users remain completely unaware of the underlying process. They only need to use node or edge data flexibly according to their own needs just like handling dense tensors within a single-machine setup. Line 29 demonstrates this, where the implemented functions only require simple addition, multiplication, or other operations such as `ReLU`.
- **JIT compilation**: After the reconstruction of the PyG algorithm, it is transformed into the deep learning modules `GATConvVertex` and `GATConvMP`, optimized for independent dense tensor processing, as shown in the above code snippet. By utilizing torch.jit, efficient machine code can be produced. Depending on the application requirements, the code can be generated either offline or online.

3 Results

We utilize graphs from 3 real-world business scenarios in Ant Group and compare our method (on CPU docker clusters) with in-house graph sampling-based subgraph learning implementations (on V100 GPU clusters) of Ant Group. Our comprehensive comparison yields the following results: ① For the fraud detection graph, which consists of **340 million nodes and 1.1 billion edges**, the inference speed of the HGT model is improved by **12.8 times**. ② In the digital technology business graph, featuring **800 million nodes and 7.4 billion edges**, the GeniePath model gets its inference speed improved by **a factor of 8**. ③ Regarding the credit graph, which includes **500 million nodes and 22.4 billion edges**, the performance of the GAT model shows a significant improvement of **27.4 times**.

The experimental results indicate that the improvements primarily arise from two key factors. Firstly, full-graph inference effectively mitigates the issue of both redundant computations and significant time consumption associated with graph sampling-based methods, especially for large-scale graphs. Secondly, performance enhancements are achieved via JIT technology, which converts the original sparse graph data computation with communication involved into local dense data computation, and optimizes it into machine code.

4 Conclusion

The paper introduces a novel, efficient processing paradigm that distinctly organizes the workflow of GNNs, allowing each section to operate concisely. This separation minimizes unnecessary communication, thereby simplifying and enhancing the processing efficiency. By utilizing `torch.jit`'s robust JIT compilation capability, the solution generates highly efficient machine code. This optimization greatly improves the computational efficiency and reduces the inference time for GNNs, making deployment more friendly. Consequently, our paradigm simplifies complex GNN inference tasks into a more manageable and deployable process, offering substantial support for the advancement of industry-scale graph data processing and related applications. Finally, it is worth mentioning that our paradigm has been deployed in production for more than two years.

References

[1] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop*.

[2] Yongchao Liu, Houyi Li, Guowei Zhang, Xintan Zeng, Yongyong Li, Bin Huang, Peng Zhang, Zhao Li, Xiaowei Zhu, Changhua He, and Wenguang Chen. 2023. GraphTheta: A Distributed Graph Neural Network Learning System With Flexible Training Strategy. arXiv:2104.10569

[3] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *ICLR*.

[4] Jiping Yu, Wei Qin, Xiaowei Zhu, Zhenbo Sun, Jianqiang Huang, Xiaohan Li, and Wenguang Chen. 2021. DFOGraph: an I/O- and communication-efficient system for distributed fully-out-of-core graph processing. In *PPoPP*.

[5] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-purpose Graph Machine Learning. *Proceedings of the VLDB Endowment* 13, 12 (2020).