A Distributed Graph Inference Computation Framework Based on Graph Neural Network Model

Zeting Pan, Yue Yu, Junsheng Chang*
College of Computer
National University of Defense Technology
Changsha, China
{pannudt, yuyue, junshengchang}@nudt.edu.cn

***Abstract*—A graph is a structure that can effectively represent objects and the relationships between them. Graph Neural Networks (GNNs) enable deep learning to be applied in the graph domain. However, most GNN models are trained offline and cannot be directly used in real-time monitoring scenarios. In addition, due to the very large data scale of the graph, a single machine cannot meet the demand, and there is a performance bottleneck. Therefore, we propose a distributed graph neural network inference computing framework, which can be applied to GNN models in the form of Encoder-Decoder. We propose the idea of "single-point inference, message passing, distributed computing", which enables the system to use offline-trained GNNs for real-time inference computations on graph data. To maintain the model effect, we add the second-degree subgraph and mailbox mechanism to the continuous iterative calculation. Finally, our results on public datasets show that this method greatly improves the upper limit of inference computation and has better timeliness. And it maintains a good model effect on three types of classical tasks. The source code is published in a Github repository.**

Keywords-component; graph inference; graph neural network; distributed graph computing

I. INTRODUCTION

A graph is an abstract data structure. A graph G = (V, E) consists of a vertex set V and an edge set E, which can be used to represent multiple objects and the relationship between them. Initially, scholars' research on graphs mainly focused on static graphs, that is, without considering temporal information. With the increase in applications, static graphs can no longer meet practical requirements, so more researchers begin to explore dynamic graphs, from discrete-time dynamic graphs to continuous-time dynamic graphs [1]. Generally speaking, for a dynamic graph, the vertices on the graph be represented as $\forall\, v_i \in V$, v_i=(id, feat, timestamp), i=1,2,..., and the edges bet can be represented as $\forall\, e_i \in E$, e_i=(src, dst, feat, timestamp), i=1,2,.... These properties can be summarized as identifiers, characteristics, and timestamps.

In practical applications, the scale of graph data is often very large, such as payment transactions, social interactions, and biological information [2]. A survey [3] showed that the graphs in practice typically contain more than 1 billion edges. Another survey [4] noted that over 68.5% of tasks applied machine learning algorithms (clustering, regression, etc.) on the graph. The popularity of deep learning has also prompted people to migrate it to the graph domain, such as graph neural networks (GNNs). The more well-known networks in GNNs are GCN [5], GAT [6], TGAT [7], etc. They mainly perform three types of tasks: link prediction (LP), node classification (NC), and edge classification (EC). And it has a good effect on task accuracy [8].

However, many problems arise when applying GNN models to graph inference computations. The first is the single-machine capacity problem. A single machine cannot withstand large-scale graph data, and there will be serious performance bottlenecks. Therefore, a distributed computing environment has become an urgent need for graph computing. Apache Spark [9] is a cluster computing framework, and GraphX [10] is a distributed graph processing framework. GraphX can be used to express graph computations, but it does not directly support GNNs. The second is the real-time problem of graph computing. The offline training method of GNNs limits its application to scenarios with low real-time requirements. If applied to payment security, the application will not be able to quickly intercept fraud, money laundering, and other dangerous behaviors. In conclusion, it is important to apply offline-trained GNN models to distributed environments for real-time inference computation. Therefore, this paper proposes a framework that maintains GraphX features while supporting GNN models, enabling real-time distributed graph inference computation.

The framework proposed in this paper is mainly to solve the following two problems. (1) How to apply the GNN model to a distributed system for graph inference computation. In the application, the GNN model does not support serialization and cannot meet the distributed requirements. To solve this problem, we first modify the model input and output so that they can be directly used for online inference. Next, we adopt the idea of "single-point inference, message passing, distributed computing". That is, the model is stored in a distributed manner, and models on different machines are dynamically called when used. Finally, the output of the model is passed as a message to the relevant vertices. In this way, the passed message size is reduced from model level (MB-GB) to matrix level (KB-MB). (2) How to perform real-time inference calculations and maintain model performance. When an event occurs, the scope of influence is often more than the source vertex (src) and the destination vertex (dst). Therefore, we have iteratively updated graph properties through incremental composition, computing

*Corresponding author.
DOI reference number: 10.18293/SEKE2022-042

second-degree subgraphs, and mailbox mechanisms. These steps will be disassembled into many tasks during execution and distributed to multiple worker nodes for parallel computing. In this way, we can apply the GNN model in the form of Encoder-Decoder to the actual environment for real-time distributed graph inference calculations. The contributions of this paper are summarized as follows:

- We put forward the idea of "single-point inference, message passing, distributed computing" so that the GNN model in the form of Encoder-Decoder can be applied to a distributed environment.
- We propose a method based on incremental composition, constructing second-degree subgraphs, and maintaining mailboxes so that distributed inference computing can ensure both timeliness and effects.
- Finally, we implemented the framework and tested it on Wikipedia and Reddit. The results show that the single-event inference time for a thousand events is 1.6203s, which is only 36.19% longer than that of a single machine under the same conditions, but the throughput is improved by 116.89%. In addition, the effect of three categories of tasks is maintained, among which the accuracy of the Wikipedia LP task is 87.03%.

II. Related Work

A. Distributed graph computing framework

MapReduce is a simple distributed computing framework that facilitates the processing of massive graph data, but it cannot iteratively compute efficiently. Bulk synchronous parallel (BSP) [11] proposed by Valiant in 1990 is suitable for iterative computing of graphs, which decomposes tasks into a series of iterative operations. Inspired by BSP, Google proposed the first vertex-centric distributed graph computing framework Pregel [12] in 2010. Since then, researchers have successively proposed a variety of distributed graph computing frameworks, including PowerGraph [13], GraphHP [14], and Hybrid [15]. These frameworks can efficiently perform graph iteration algorithms. However, their limited expressive computation makes it difficult to express important stages in a typical graph analysis pipeline, such as graph modification, cross-graph computation, etc.

Spark GraphX is a distributed graph processing framework, and its core abstraction is Resilient Distributed Property Graph, a directed multigraph with properties on both nodes and edges. GraphX extends the abstraction of Spark RDD and has two views (table and graph), and only needs one physical storage [16]. These two views have their unique operators, to obtain flexible operation and execution efficiency. In terms of calculation, all operations on the view will be converted into RDD operations of the associated table view to complete. In this way, graph computation is equivalent to the transformation process of a series of RDDs. Therefore, GraphX finally has three key features: Immutable, Distributed, and Fault-Tolerant.

B. Graph Neural Network

Dynamic graph representation has gone through four main stages of development, namely static, weighted edge, discrete, and continuous. There are also several types of methods for learning dynamic graph representations, including tensor decomposition, random walk, and deep learning. Among them, GNNs in deep learning methods have received extensive attention because they can combine time series encoding with aggregation of adjacent nodes. GCN [5] learns features better by aggregating the information of neighbor points, but it cannot use temporal information. However, in the field of graphs, timing has an important influence on the change of vertex-edge relationship, so the research direction of GNNs gradually shifts from static to dynamic.

Dynamic graph neural network (DGNN) aggregates deep time series encoding and node features and is mainly divided into discrete and continuous categories. Discrete DGNN first uses a certain GNN to obtain vertex embedding and then uses a certain RNN or Attention network for time series modeling. The representative networks are DySAT [17], etc. Most of the current methods of continuous DGNN use snapshot modeling, which is only a rough estimation of time, and related networks include TGAT [7], TGN [18], etc. TGN attempts to introduce the Encoder-Decoder neural network framework in the graph field. The encoder is responsible for encoding the vertex and edge features on the graph into vectors, and the decoder calculates and predicts attribute values for the encoded vectors according to specific executing tasks. This form of decoupling enables real-time inference computations on graphs.

III. Method

This section will first introduce the overall framework and then introduce the three main modules in the framework in detail.

A. Overall Framework

Fig. 1 presents an overview of our distributed graph inference. It is a distributed graph inference framework that supports the Encoder-Decoder form of GNN. The framework is mainly composed of the following modules: incremental composition, second-degree subgraph calculation, GNN encoder, mailbox, and GNN decoder.

- **Incremental composition module** corresponding to steps (a)-(b). When a new event is generated in the data source, the event will be added to the vertexRDD and edgeRDD of the historical graph event, and incremental composition will be performed to obtain the whole graph. Note that if it is an undirected graph, for an event e, we will generate both forward and reverse edges.
- **Second-degree subgraph calculation module**, which is the "Full Graph and 2D-Subgraph" in the figure. After the whole graph is obtained, the basic properties of some vertices and edges will be updated according to e and its second-degree subgraph will be calculated.
- **GNN encoder module** corresponding to steps (c)-(e). After loading the trained GNN encoder model, the second-degree subgraph feature matrix is used as the input of the model. Then, the embedding of each vertex in the second-degree subgraph can be obtained and then updated to the whole graph.

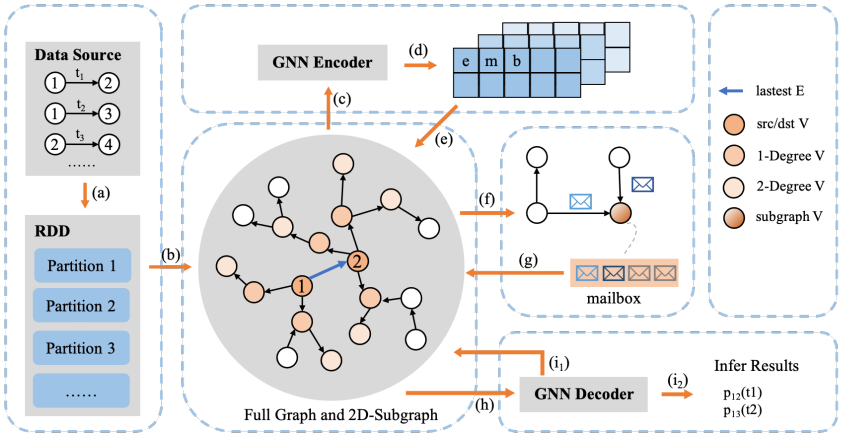

Figure 1. Distributed graph inference system framework.

- **The mailbox module** corresponds to steps (f)-(g). A mail will be sent along each edge in the second-degree subgraph, and the main content of the mail is the current features of the edge and some historical interaction information. The vertex that receives the mail will add it to its mailbox.
- **GNN decoder module** corresponding to steps (h)-(i). The above results will be decoded, and logical inference results will be given according to the types of tasks performed (LP/NC/EC).

B. *Second Degree Subgraph Algorithm*

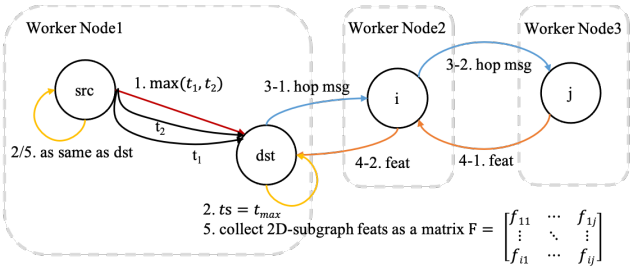

Figure 2. Second-degree subgraph algorithm.

When a new event e=(src, dst, timestamp, feat) occurs, the edges with the same starting vertex and destination vertex are first merged, as shown in Fig. 2. The way to merge is to leave the edge with the largest timestamp. Since historical events are often embedded into features by previous inferences, merging duplicate edges does not affect results. It can also reduce the number of edges in the graph to save resources, especially when the vertex-to-edge ratio is large. Then, as shown in step 2 in Fig. 2, the timestamps of src and dst will be updated to $\max(t_1, t_2)$.

Due to the particularity of graphs, an event often affects more than src and dst . Considering the effect and performance comprehensively, we decided to control its influence range within the subgraph reachable by two hops. To obtain this subgraph, we design steps 3 and 4, namely the sending of hop messages and the return of feature messages. From src and dst, two rounds of message sending operations will be performed. The first round will send its hop-1 to the neighbor and update the hop value of the neighbor vertex. In the second round, the neighbor vertex sends its hop-1 to its neighbor (excluding src and dst) and updates the hop value of the vertex that receives the message. This obtains a second-degree subgraph about the new event e. Step 4 is the opposite of Step 3, which will transmit its feature information back from the second-degree vertex. After two rounds of message return, src and dst will aggregate the features of all vertices in the second-degree subgraph. Step 5 stores these features in src as one of the attributes of the vertex.

C. *GNN Encoder algorithm*

Trained GNN models tend to take up a lot of storage space and are difficult to serialize. Even with serialization methods, transferring models between machines incurs a significant bandwidth overhead, which can severely impact performance. Therefore, the inference algorithm we designed will use the spark driver to uniformly manage the scheduling of machines, so that each worker node has a copy of the GNN model, as shown in Fig. 3. The model can be loaded directly and dynamically during the inference process, without the need for network transmission. Furthermore, all vertices in a second-degree subgraph need to be inferred, but doing inference at the same time introduces additional overhead. Therefore, we propose an algorithm for "single-point inference, message passing, distributed computing". Based on the subgraph collected in the previous step, we only load the model in src or dst, then get the model output and update the vertex features. Finally, the model output is sent to other vertices in the subgraph in the form of a message to complete the vertex update. For the GNN decoder algorithm, the process is the same as the GNN encoder. Due to space limitations, we will not repeat them here.

D. *Mailbox algorithm*

As shown in Fig. 3, after updating the vertex features using embedding, we will generate a mail along each edge in the subgraph (mail = feat_{src} + feat_{dst} + feat_{e}), and then sent it to the destination vertex. Each vertex in the subgraph will take an average of all the received emails, and then add it to the mailbox. That is to say, a vertex can only have one mail added to its mailbox in one iteration, which solves the problem of supernode message explosion. The mailbox maintained by each vertex is implemented using a list, and the default maximum capacity of the list is 10 (this value can be modified more practically). When

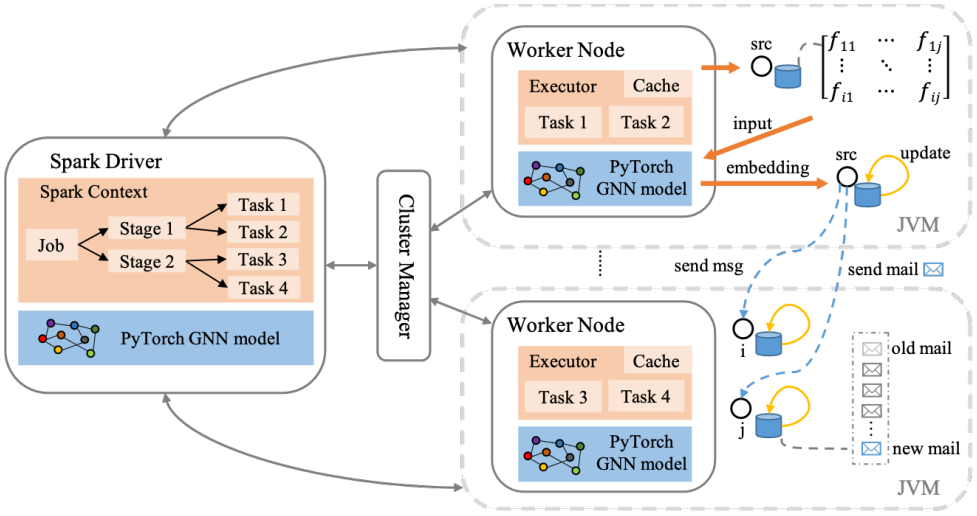

Figure 3. The execution process of inference and computation.

there are less than 10 mails, new mail will be added directly to the end of the list. When it is greater than 10, the oldest mail in the list will be deleted to leave space for new mail. Through the mechanism of the mailbox, the storage of each vertex itself includes vertex features, information about neighbor vertices, and features brought by historical events. When inferring, it can be encoded into the form of a matrix and directly input into the model.

E. Algorithm Pseudocode

Through the above algorithm, we realize distributed graph inference computing. That is, graphs can be incrementally composed and iteratively performed graph computation, graph inference, and graph update. The pseudocode of the overall logic is shown in Table 1 below.

TABLE I. PSEUDOCODE FOR DISTRIBUTED GRAPH COMPUTING INFERENCE

Algorithm: Distributed graph computing inference
Input: event data source e=(src,dst,feat,timestamp)
Output: logical inference results and accuracy
initialize spark and static resources config
use n-events warm up inference
while has new event e **do**
if $v_{src/dst} \notin$ vRDD **then**
initialize $v_{src/dst}$ and add them to vRDD
add e to eRDD, then create graph with vRDD and eRDD
merge duplicate edges and update $v_{src/dst}$ with the timestamp of the lastest edge
for v_{src} or v_{dst} **do**
send hop_{v_i}-1 to its neighbors
for each vertex that receives src or dst hop messages **do**
send hop_{v_j}-1 to its neighbors
collect all vertex features of received hop messages to src as 2D-subgraph
update feat of subgraph vertices to embedding=Encoder(2D-subgraph)
for each $e_i \in$2D-subgraph **do**
send mail messages to the dst along the edge, and calculate the average value of all mails at dst
for each $v_i \in$2D-subgraph **do**
if len($mailbox_i$)≥10 **then**
remove the oldest mail from $mailbox_i$ header
add average mail to the tail of the $mailbox_i$
for triplets(src,dst,e) **do**
update the logical results of triplets to logit=Decoder(concat(src,dst,e))
calculate the inference results of the whole graph to get the accuracy
return accuracy

IV. EXPERIMENTS

In this section, we will first introduce the experimental setup, including the device environment, datasets, GNN models, and inference system. Next, the results of the experiment will be explained in terms of performance and effects. The source code of our system is published at a Github repository[1].

A. Setting

- **Hardware and software environment.** Due to the limitation of cluster resources, this experiment uses multithreading to simulate a distributed environment. The processor is Intel(R) Core(TM) i5-8257U CPU @2.00GHz, 16GB memory, 500G hard disk. The operating system is macOS Catalina 10.15.7, and the development tool is IntelliJ IDEA 2020.1.2. The running environment is Spark 3.2.0, Hadoop 3.3.0, Scala 2.12.15, and Java 1.8.0.
- **Graph datasets.** This experiment uses Wikipedia [19], and Reddit [19] public datasets for experiments. Among them, Wikipedia represents the interaction between users and wiki pages. Reddit represents the interaction events of users in social networking. The timestamps of all edges in both datasets span 30 days.
- **GNN models.** The framework proposed in this paper is suitable for GNNs in the form of encoder-decoder, we chose APAN [20] and reproduced it. Considering that Java does not support graph input when calling the model, we modified the input and output data form of APAN to matrices. The main parameters when training the model are: the maximum epoch is 50, the batchsize is 100, the initial learning rate is 0.0001, and the dropout

[1]https://github.com/napdada/Distributed-Graph-Inference-Java

TABLE II. GRAPH INFERENCE CALCULATION RESULTS AND MAIN STEPS TIME-CONSUMING IN TWO ENVIRONMENTS

Environment	LP task inference time(s)		Throughput cap	Time-consuming inference calculation of specific steps(ms)								
	1000 events	*1000 events(*)*		*Create Graph*	*Merge Edges*	*Update Ts*	*2D-Subgraph*	*Encoder*	*Send Emb*	*Mailbox*	*Decoder*	*Evaluate*
SM	1.1897	1.1210	1835	4.24	0.37	1.76	733.51	2.16	374.62	3.62	0.44	68.79
DM	1.6203	1.5335	3980	4.40	0.38	1.84	951.76	2.24	568.35	3.77	0.48	86.83

a. SM: single machine environment, DM: distributed machine environment. The names of the specific steps from left to right represent incremental composition, merge duplicate edges, update timestamp, generate second-degree subgraph, call encoder model, send embedding, transfer mail, call decoder, and evaluate results

is 0.1. The ratio of training, testing, and validation data is 7:1.5:1.5. After training, the ".pt" models of three types of tasks (LP/NC/EC) on two datasets are obtained, and the effect is the same as in the original paper.

- **Inference system.** Considering the system startup process and graph initial features are zero, we set a warm-up process of inference. That is, by default, the first ten inferences are not included in the result. The number of distributed cores and partitions is both 2, and the partition strategy is EdgePartition2D. The JVM parameters will be adjusted according to the executed tasks, mainly adjusting the memory size. In addition, to reduce the effect of chance, all results are the mean of ten replicate experiments.

B. *Inference performance*

1) *Timeliness and throughput caps.*

We reproduce APAN and apply it to graph inference computation to compare single machine and distributed. We configure 4G memory for a single machine. We configured 2-core 8G memory for the distributed environment, which is equivalent to two single machines working at the same time. After that, we calculated the average time and upper limit of these two environments as shown in Table 2. It can be seen that when executing the LP task with a thousand events on Wikipedia, the distributed time is longer than that of a single machine. However, the upper limit of distributed computing is increased, and the number of iterations in a single-machine environment will overflow when about 1800. That's to say, in the face of large graph data scenarios, distributed inference computing can solve the capacity bottleneck problem of a single machine.

To further explore the time-consuming, we have detailed statistics on the time-consuming of the main steps, as shown in Table 2. The data in the table visually show that the extra time is mainly spent on 2D-Subgraph, SendEmb, and Evaluate. Because in a single-machine environment, vertices and edges are stored on one machine, they only need to be read when they are used. However, vertices are stored in partitions according to the policies in a distributed environment, so data exchange in different partitions will bring additional communication overhead. Furthermore, to evaluate the results, the system needs to retrieve data from all partitions. This operation is time-consuming, and we count it into the results (the data marked with "*" in Table 2 are not included in the time-consuming statistics of this operation), so we believe that a single time consumption of 1.62s is reasonable. Comparing the results in the two environments, it can be seen that when the iterative inference is executed 1000 times, the distributed graph inference calculation uses 36.19% of the time overhead, in exchange for 116.89% of the larger throughput.

Considering that the graph continues to grow during the iterative process, if the resource consumption increases exponentially as the graph grows, the huge overhead will inevitably make the system worthless. To this end, we conducted inference experiments with varying numbers of events, ranging from 50 to 1000, within a range where problems such as overflow do not occur. The average time-consuming inference calculation is shown in Fig. 4. It can be seen that with the increase of the graph size, the inference computation time increases linearly. That is, our framework can be extended to work on clusters. In the face of large graph data, it can still guarantee the linear growth of resource consumption, rather than the problem of exponential explosion.

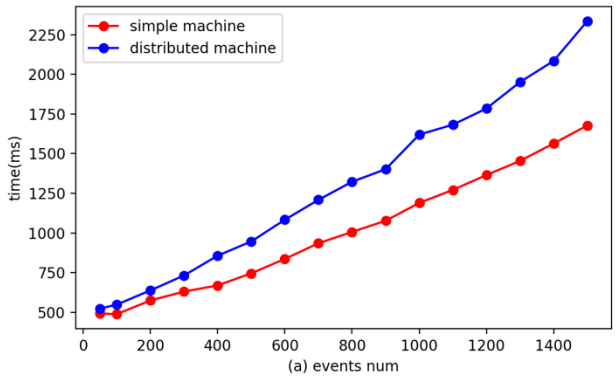

Figure 4. The relationship between the number of inference events and the time-consuming when performing LP tasks on Wikipedia

2) *Partitioning and Strategy.*

The number and strategy of partitions in a distributed environment often significantly impact the results, so we experimented with them. We set up a 4-core 5G distributed environment that used 100 events in Wikipedia to perform LP tasks. The results are shown in Table 3. Regarding the number of partitions, we can find that the computation time of graph inference increases rapidly with the number of partitions. Therefore, we need to dynamically adjust some configurations according to the actual situation. For small graphs, the number of partitions can be reduced. But for large graphs, we can increase the number of partitions and add machines to reduce the time-consuming impact. For the partitioning strategy, we can see that the random partitioning effect is the worst, followed by the EdgePartition1D (partitioning based on src only). The best partition strategy is EdgePartition2D (partition by both src and dst). This is in line with our understanding of graphs, that for edges, both the source and destination vertices are important.

TABLE III. THE EFFECT OF THE NUMBER OF PARTITIONS AND PARTITION STRATEGY ON TIME CONSUMPTION

Number of partitions	Partitioning strategy	Time (ms)
1	EdgePartition2D	606.24
2	EdgePartition2D	614.87
4	EdgePartition2D	655.79
8	EdgePartition2D	786.46
4	RandomVertexCut	674.38
4	EdgePartition1D	670.95

C. *Inference performance*

Table 4 shows the effect of APAN original data, reproduced APAN model data (APAN-Re), classic GNN model, and our distributed graph inference algorithm. That is the accuracy of performing three types of tasks on two datasets. It can be seen that our method maintains the effect of the model better. In addition, the method achieves the effect of classical GNN, and even outperforms classical GNN on some tasks.

TABLE IV. ACCURACY OF INFERENCE RESULTS ON THREE TYPES OF TASKS (LP/NC/EC)

	Wikipedia			Reddit		
	LP	***NC***	***EC***	***LP***	***NC***	***EC***
GAT	87.34	-	-	92.14	-	-
TGAT	88.14	-	-	92.92	-	-
TGN	89.51	-	-	92.56	-	-
APAN	90.74	-	-	94.34	-	-
APAN-Re	87.42	99.81	99.81	97.76	99.91	99.91
distributed graph inference	87.03	99.54	99.28	97.01	98.39	98.05

V. CONCLUSION AND FUTURE WORK

In this paper, we propose a distributed graph inference computing framework, aiming to use GNN models in the form of Encoder-Decoder for online deployment and real-time inference in distributed environments. The experimental results show that the upper limit of inference calculation has been greatly improved. It has also achieved good results in the timeliness of inference, and we believe that it can meet the actual needs in the case of limited resources. Furthermore, as graph iterative inference proceeds, our method can maintain the model's performance on three classes of tasks. In the future, we can extend the algorithm to adapt to more kinds of GNN models. And consider optimizing the communication overhead in a distributed environment, so that the system can have stronger real-time inference computing capabilities.

ACKNOWLEDGMENT

This work was supported by the National Key R&D Program of China (2020AAA0103500).

REFERENCES

[1] Skardinga J, Gabrys B, Musial K. Foundations and modelling of dynamic networks using dynamic graph neural networks: A survey[J]. IEEE Access, 2021.

[2] Zhang X C, Wu C K, Yang Z J, et al. MG-BERT: leveraging unsupervised atomic representation learning for molecular property prediction[J]. Briefings in Bioinformatics, 2021.

[3] Sahu S, Mhedhbi A, Salihoglu S, et al. The ubiquity of large graphs and surprising challenges of graph processing[J]. Proceedings of the VLDB Endowment, 2017, 11(4): 420-431.

[4] Sahu S, Mhedhbi A, Salihoglu S, et al. The ubiquity of large graphs and surprising challenges of graph processing: extended survey[J]. The VLDB Journal, 2020, 29(2): 595-618.

[5] Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks[J]. arXiv preprint arXiv:1609.02907, 2016.

[6] Veličković P, Cucurull G, Casanova A, et al. Graph attention networks[J]. arXiv preprint arXiv:1710.10903, 2017.

[7] Xu D, Ruan C, Korpeoglu E, et al. Inductive representation learning on temporal graphs[J]. arXiv preprint arXiv:2002.07962, 2020.

[8] Dwivedi V P, Joshi C K, Laurent T, et al. Benchmarking graph neural networks[J]. arXiv preprint arXiv:2003.00982, 2020.

[9] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: Cluster computing with working sets[J]. HotCloud, 2010, 10(10-10): 95.

[10] Gonzalez J E, Xin R S, Dave A, et al. Graphx: Graph processing in a distributed dataflow framework[C]//11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14). 2014: 599-613.

[11] Cormen T H, Goodrich M T. A bridging model for parallel computation, communication, and I/O[J]. ACM Computing Surveys (CSUR), 1996, 28(4es): 208-es.

[12] Malewicz G, Austern M H, Bik A J C, et al. Pregel: a system for large-scale graph processing[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. 2010: 135-146.

[13] Gonzalez J E, Low Y, Gu H, et al. Powergraph: Distributed graph-parallel computation on natural graphs[C]//10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12). 2012: 17-30.

[14] Chen Q, Bai S, Li Z, et al. GraphHP: A hybrid platform for iterative graph processing[J]. arXiv preprint arXiv:1706.07221, 2017.

[15] Wang Z, Gu Y, Bao Y, et al. Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing[C]//Proceedings of the 2016 International Conference on Management of Data. 2016: 479-494.

[16] Tang J, Xu M, Fu S, et al. A scheduling optimization technique based on reuse in spark to defend against apt attack[J]. Tsinghua Science and Technology, 2018, 23(5): 550-560.

[17] Sankar A, Wu Y, Gou L, et al. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks[C]//Proceedings of the 13th International Conference on Web Search and Data Mining. 2020: 519-527.

[18] Rossi E, Chamberlain B, Frasca F, et al. Temporal graph networks for deep learning on dynamic graphs[J]. arXiv preprint arXiv:2006.10637, 2020.

[19] Kumar S, Zhang X, Leskovec J. Predicting dynamic embedding trajectory in temporal interaction networks[C]//Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2019: 1269-1278.

[20] Wang X, Lyu D, Li M, et al. APAN: Asynchronous Propagation Attention Network for Real-time Temporal Graph Embedding[C]//Proceedings of the 2021 International Conference on Management of Data. 2021: 2628-2638.