STGraph: A Framework for Temporal Graph Neural Networks

Joel Mathew Cherian [1], **Nithin Puthalath Manoj** [1]
Kevin Jude Concessao [2], **Unnikrishnan Cheramangalath** [2]
[1] Department of CSE, NIT Calicut, [2] Department of CSE, IIT Palakkad
joelmathewcherian@gmail.com, nithinp.manoj@gmail.com
112014001@smail.iitpkd.ac.in, unnikrishnan@iitpkd.ac.in

Abstract

Real-life graphs from various application domains like social networks, transportation networks, and citation networks evolve over time. Temporal Graph Neural Networks (TGNNs) are used for analyzing spatial and temporal properties of graphs from these application domains. We propose STGraph, a framework to program TGNNs. The proposed framework extends Seastar, a vertex-centric programming model for training static GNNs on GPUs. STGraph supports TGNNs for static graphs with temporal signals and discrete-time dynamic graphs (DTDGs). Existing TGNN frameworks store DTDGs as separate snapshots, incurring high memory overhead. As an improvement, STGraph constructs each snapshot on demand during training. This is achieved by integrating the system with dynamic graph data structures capable of building graph snapshots from temporal updates. STGraph is benchmarked against PyTorch Geometric Temporal (PyG-T) on an NVIDIA GPU. For static graphs with temporal signals, STGraph shows up to 1.22× speedup and up to 2.14× memory improvement over PyG-T. For DTDGs, STGraph exhibits up to 1.70× speedup and 1.52× memory improvement over PyG-T.

1 Introduction

Graph Neural Networks (GNNs) have become robust tools in deep learning, revolutionizing how we infer information from graph data structures [1–4]. They leverage the inherent structural properties of static graphs to excel in tasks such as node classification, link prediction, and graph classification. Unlike static graphs, temporal graphs incorporate time as an extra dimension and capture sequential changes in graph structure and attributes. Temporal Graph Neural Networks (TGNNs) analyze these changes by extending the traditional GNN architecture [5, 6] with a recurrent unit or an attention-based mechanism. These models are valuable in real-life applications such as social network analysis, traffic forecasting, and epidemic prediction.

Numerous frameworks, such as DGL [7] and PyG [8], are available for creating GNN models. However, these frameworks suffer from high memory consumption, poor data locality, and a significant gap between the design and implementation of GNN models. *Seastar* [9], a vertex-centric programming model for GNN training on NVIDIA GPUs, addresses the limitations of frameworks like DGL and PyG. Additionally, GNNs are constructed in *Seastar* by defining the logic for a single vertex. The *Seastar* system outperforms state-of-the-art GNN frameworks but lacks support for TGNNs.

We introduce STGraph, a framework written in Python and CUDA/C++, built on top of *Seastar*. STGraph enables deep learning practitioners to develop TGNN models through a vertex-centric approach. We benchmarked STGraph against PyG-T[10], a state-of-the-art framework built on top of PyG for creating TGNN models. Similar to PyG-T, STGraph supports both static-temporal (static graphs with temporal signals) and discrete-time dynamic graphs (dynamic graphs with both static and dynamic signals). Overall, STGraph outperforms PyG-T, exhibiting 1.22× performance improvement

Temporal Graph Learning Workshop @ NeurIPS 2023, New Orleans.

and 2.14× lower memory consumption for static-temporal graphs. For discrete-time dynamic graphs, it demonstrates 1.70× performance improvement and 1.52× lower memory consumption.

The primary contributions of our work can be summarized as follows:-

- `STGraph`, a framework to train TGNN models seamlessly on the GPU.
- Integrating `STGraph` with dynamic graph data structures for optimized discrete-time dynamic graph processing.
- Publicly releasing `STGraph`[1] as an open-source deep learning library for graphs. The library includes GNN and TGNN layer APIs, dataset loaders and the `STGraph` framework for building custom TGNNs.

2 Proposed Work

The `STGraph` architecture is illustrated in Figure 1. This design uses components from *Seastar* (uncolored parts of Figure 1) along with custom modules, newly introduced by `STGraph`, for temporal functionality (colored parts of Figure 1). *Seastar* is used to generate forward and backward execution units (CUDA kernels) by tracing, auto-differentiating and optimizing the vertex-centric function (See Appendix A.1). The *`STGraph` Dataloader* processes datasets to generate graph objects. During the execution of the vertex-centric function, the *`STGraph` Backend Interface* passes these graph objects to the *Temporally-aware Executor*. The executor orchestrates which snapshot of the graph object is passed to forward and backward units during forward and backward propagation respectively. This orchestration is enabled by memory data structures such as *State-Stack* and *Graph-Stack* along with the *Graph Update* and *Graph Reverse* module. Additionally, all features integrated into `STGraph` confine their interactions with the backend through the *`STGraph` backend interface* (See Figure 1). This interface ensures the backend-agnostic nature of the `STGraph` framework.

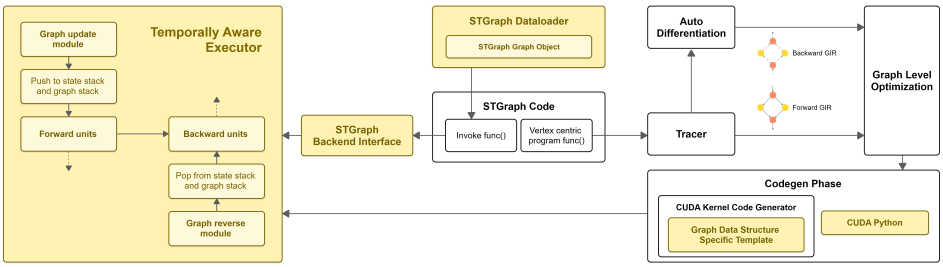

Figure 1: `STGraph` Architecture

2.1 TGNN Training in `STGraph`

The new components introduced by `STGraph` account for the differences in training TGNNs as compared to training conventional GNNs. During training, TGNNs need to process a sequence of features from consecutive timestamps before learning from that sequence. Consider that the total number of timestamps in the input is denoted by N, with each individual timestamp being denoted by $t_0, t_1, ...t_{N-1}$. The total timestamps are then divided into a set of sequences S_0 to S_{k-1} (k $< N$). Each sequence contains a set of consecutive timestamps. For any two sequences S_i and S_j with $i < j$, the timestamps in S_i and S_j are disjoint, and for any timestamp $x \in S_i$ and $y \in S_j$, $x < y$. Forward propagation on a sequence S_i ($i < k$) accumulates the loss for each timestamp in S_i. At the end of forward propagation, the model propagates backwards, updating its parameters repeatedly for all the timestamps in the sequence S_i. For every timestamp t in S_i, the node/edge features and graph structure used during forward and backward propagation at t must match. `STGraph` uses the **State Stack** and **Graph Stack** to maintain a memory of feature vectors and graph structure respectively.

Algorithm 1 describes how TGNN models are trained using `STGraph`. The algorithm takes as input G (input graph object), F (feature vectors for all timestamps), T (total number of timestamps), and N (number of epochs to be used in training). The *state-stack* and *graph-stack* are created initially (Lines 1- 2). M is initialized to an untrained TGNN model (See Line 3). The training happens for

[1] `https://github.com/bfGraph/STGraph`

N epochs (Lines 4- 14). The total timestamps T is divided into a set of *sequences*. Each *sequence* s is considered in an ordered fashion in training (See Lines 6- 13). The forward propagation for a *sequence* s happens in a `for` loop (See Lines 7- 9). The backward propagation happens in a `while` loop till the *state-stack* becomes empty (See Lines 10-12).

Algorithm 1: STGraph-Train(G, F, T, N)

Input: A graph object G, list of feature vectors F for all timestamps, total timestamps T and total number of epochs N
Output: Trained TGNN-model

```
1  state-stack = Stack()
2  graph-stack = Stack()
3  M = GNN-Model()
4  for epoch=1 to N do
5      loss = 0
6      for sequence s in T do
7          for timestamp t in s do
8              STGraph-Execute (fwd, loss, t)
9          end for
10         while state-stack is not empty do
11             STGraph-Execute (bwd, loss)
12         end while
13     end for
14 end for
15 return M
```

Algorithm 2: STGraph-Execute(D, $loss$, t)

Input: The direction of propagation D, Loss of the model $loss$, Timestamp of execution t (in the case of forward propagation)

```
1  g = G
2  if D is fwd then
3      if G is DTDG then
4          g = Get-Graph (G,t)
5          graph-stack.push(g)
6      end if
7      state-stack.push(F_t)
8      out = fwdprop-model-t (M, g, F_t)
9      loss += loss-fn (out)
10 end if
11 if D is bwd then
12     if G is DTDG then
13         g = graph-stack.pop()
14         g_B = Get-Backward-Graph (g)
15     end if
16     F_t = state-stack.pop()
17     bwdprop-model-t (M, g_B, F_t, loss)
18 end if
```

Algorithm 2 describes how `STGraph` handles forward (See Lines 2- 10) and backward (See Lines 11-18) propagation. Initially, the graph object g is initialized to G (Line 1). If the input graph object is a static graph, then the object g is never updated and the *graph-stack* is not used. If the graph object G is a DTDG then there is a separate graph snapshot for each timestamp $t \in s$. For forward propagation, the graph object corresponding to a timestamp t is assigned to g using the *Get-Graph()* function following which g is pushed to *graph-stack* (See Lines 3- 6). The state for each timestamp is pushed to the *state-stack* (See Line 7). This is followed by forward propagation on g and an update of the loss (See Lines 8- 9). For backward propagation, If G is a DTDG, then the *graph-stack* will be popped to obtain the corresponding forward graph snapshot g. *Seastar* generated kernels for backpropagation act on the reverse graph, hence the snapshot g has to be reversed. The *Get-Backward-Graph()* function is used to generate the reverse graph g_B for a given input graph (See Lines 12- 15). The feature vector of the current timestamp is retrieved by popping the *state-stack* (See Line 16). The model (M), feature vector (F_t), reverse snapshot (g_B) and loss are used for backpropagation on the current timestamp t (See Line 17).

2.2 Memory Optimization for DTDGs

DTDGs consist of a series of graph snapshots, where consecutive ones typically vary by less than 10%. Since there is a significant amount of redundancy between two adjacent snapshots, a memory-efficient implementation would be to store DTDGs as a series of temporal updates (addition/deletion of edges). However, when storing graphs in this format, an efficient update mechanism is required to generate snapshots for each timestamp on demand during training. *Seastar's* storage format, Compressed Sparse Row (CSR), does not support efficient insertions/deletions. Our work considers alternative storage formats that support faster updates.

GPMA [11] is a GPU-based data structure that uses GPU-optimized parallel algorithms for batch insertions/deletions. Using GPMA as the underlying storage format for DTDGs can speed up the process of generating graph snapshots on demand. `STGraph-GPMA` prefaces both forward and backward propagation with a snapshot generation step. The (*Graph-Update-Module*) generates the snapshot and the (*Graph-Reverse-Module*) reverses the snapshot (in the case of backpropagation).

Additionally, `STGraph-GPMA`'s performance is optimized using custom kernels for relabelling edges and sorting vertices.

3 Experimental Evaluation

We perform the experimental evaluation on a total of ten graph datasets (See Table 2), with five being static-temporal datasets (1-5) [10], and five dynamic graph datasets (6-10) [12]. Results from a few of these tests are discussed in detail below. The experimental setup is discussed in Appendix A.3.

3.1 Static-temporal Graph Analysis

The first three plots of Figure 2 compare the per-epoch time taken by TGCN models implemented in both PyG-T and `STGraph` for different static-temporal graphs. The `STGraph` framework performs up to 1.22× faster than PyG-T in terms of per-epoch time. This improvement is attributed to the efficient kernels and kernel-level optimizations associated with the underlying *Seastar GCN* layer compared to the PyG GCN layer used by PyG-T.

The first three plots of Figure 3 compare the memory consumption of PyG-T and `STGraph` on static-temporal graphs for a fixed feature size of 8 and varying sequence lengths. It is observed that `STGraph` consumes up to 2.14× less memory than PyG-T. This is because PyG GCN layers employ edge parallelism for GNN processing [13], this requires duplication of node features, which incurs a significant memory overhead. Since PyG-T has to retain these duplications over the entire sequence length till backpropagation occurs, its curve is steeper in comparison to that of `STGraph`.

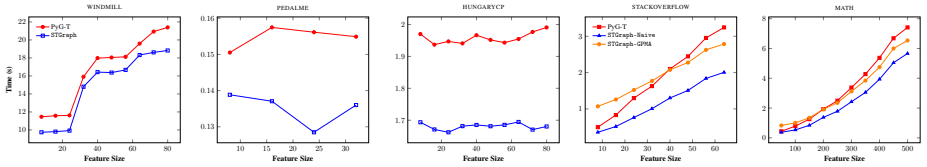

Figure 2: Per-Epoch Time vs Feature Size

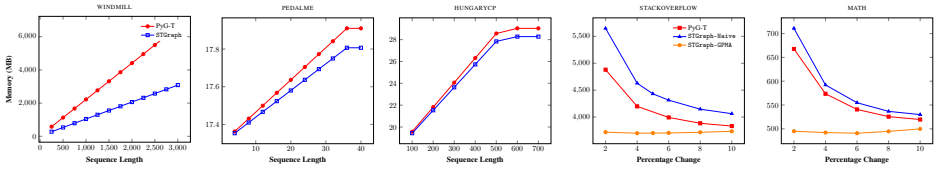

Figure 3: Memory Consumption vs Sequence Length/Percentage Change between snapshots

3.2 Dynamic Graph Analysis

The fourth and fifth plot of Figure 2 compares the per-epoch time taken by TGCN models implemented using `STGraph-Naive`, `STGraph-GPMA`, and PyG-T for DTDGs with fixed 5% change between snapshots. It is observed that `STGraph-Naive` outperforms PyG-T with up to 1.70× speed up. This result follows the same reasoning for why `STGraph` outperformed PyG-T in static-temporal graphs. `STGraph-GPMA` shows up to 1.19× speed up over PyG-T for per-epoch time. The slower performance here, in comparison to `STGraph-Naive`, is associated with the time taken to build graph snapshots on demand.

The fourth and fifth plot of Figure 3 compares the performance of the three systems when varying the percentage change between snapshots. It is observed that `STGraph-GPMA` consumes upto 1.30× and 1.52× less memory than `STGraph-Naive` and PyG-T respectively. Additionally, `STGraph-GPMA` is barely affected by varying percentage changes, while the other two systems consume significantly larger memory for smaller percentage changes. `STGraph-GPMA` is an ideal choice for graph snapshots with small percent changes because it offers memory benefits without heavily compromising on speed.

4 Conclusion and Future Work

This paper introduces `STGraph`, a novel backend-agnostic Python framework designed for training TGNNs on real-world temporal and dynamic graph datasets. Compared to PyG-T, `STGraph` demonstrates superior performance on benchmarking with real-life graph datasets. The empirical evidence shows that `STGraph` is a powerful and effective tool for deep learning researchers and practitioners

working with temporal and dynamic graphs. In the future, this system can be extended to support Heterogeneous graphs along with backend support for frameworks like TensorFlow and MXNet.

References

[1] 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," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, pp. 549–556, Apr. 2020.

[2] 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*, ser. WWW '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 417–426.

[3] H. Dai, C. Li, C. Coley, B. Dai, and L. Song, "Retrosynthesis prediction with conditional graph logic network," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019.

[4] S. Kumar, B. Hooi, D. Makhija, M. Kumar, C. Faloutsos, and V. Subrahmanian, "Rev2: Fraudulent user prediction in rating platforms," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, 2018, pp. 333–341.

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

[6] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[7] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[8] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[9] Y. Wu, K. Ma, Z. Cai, T. Jin, B. Li, C. Zheng, J. Cheng, and F. Yu, "Seastar: Vertex-centric programming for graph neural networks," 2021.

[10] B. Rozemberczki, P. Scherer, Y. He, G. Panagopoulos, A. Riedel, M. Astefanoaei, O. Kiss, F. Beres, G. Lopez, N. Collignon, and R. Sarkar, "PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models," in *CIKM*, 2021.

[11] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating dynamic graph analytics on gpus," *Proc. VLDB Endow.*, vol. 11, no. 1, 2017.

[12] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[13] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, "Understanding and bridging the gaps in current gnn performance optimizations," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 119–132.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, 2019.

[15] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between c++11 and python," 2017.

[16] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," in *GPU computing gems Jade edition*. Elsevier, 2012, pp. 359–371.

[17] L. Zhao, Y. Song, C. Zhang, Y. Liu, P. Wang, T. Lin, M. Deng, and H. Li, "T-gcn: A temporal graph convolutional network for traffic prediction," *IEEE transactions on intelligent transportation systems*, vol. 21, no. 9, 2019.