Vertex-Centric Visual Programming for Graph Neural Networks

Yidi Wu[†], Yuntao Gui[†], Tatiana Jin[†], James Cheng[†], Xiao Yan[‡], Peiqi Yin[‡], Yufei Cai[‡], Bo Tang[‡], Fan Yu[*]
[†] The Chinese University of Hong Kong
[†] {ydwu,ytgui, tjin, jcheng}@cse.cuhk.edu.hk
[‡] Southern University of Science and Technology, [*] Huawei Technologies Co.Ltd
[‡] {yanx,yinpq2018,caiyf2018,tangb3}@sustech.edu.cn, [*] fan.yu@huawei.com

ABSTRACT

Graph neural networks (GNNs) have achieved remarkable performance in many graph analytics tasks such as node classification, link prediction and graph clustering. Existing GNN systems (e.g., PyG and DGL) adopt a tensor-centric programming model and train GNNs with manually written operators. Such design results in poor usability due to the large semantic gap between the API and the GNN models, and suffers from inferior efficiency because of high memory consumption and massive data movement. We demonstrate *Seastar*, a novel GNN training framework that adopts a *vertex-centric programming paradigm* and supports *automatic kernel generation*, to simplify model development and improve training efficiency. We will (i) show how to express GNN models succinctly using **a visual "drag-and-drop" interface or Seastar's vertex-centric python API;** (ii) demonstrate the performance advantage of Seastar over existing GNN systems in convergence speed, training throughput and memory consumption; and (iii) illustrate how Seastar's optimizations (e.g., operator fusion and constant folding) improve training efficiency by profiling the run-time performance.

CCS CONCEPTS

• **Software and its engineering → Compilers**; **Graphical user interface languages**; **Source code generation**.

KEYWORDS

Graph Neural Network; Deep Learning System.

ACM Reference Format:
Yidi Wu[†], Yuntao Gui[†], Tatiana Jin[†], James Cheng[†], Xiao Yan[‡], Peiqi Yin[‡], Yufei Cai[‡], Bo Tang[‡], Fan Yu[*]. 2021. Vertex-Centric Visual Programming for Graph Neural Networks. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3448016.3452770

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452770

1 INTRODUCTION

The fast growth in the volume and variety of graph data in recent years has led to the development of many systems for graph analytics, e.g., batch processing systems [5, 15, 18, 22, 29–31] for graph tasks such as PageRank and label propagation, graph databases [1–3, 7, 10–12, 14], and graph mining systems [8, 9]. Among the many graph analytics methods, lately *graph neural networks (GNNs)* [19, 20, 26] have attracted great interests from both academia and industry. Compared with traditional techniques that are based on matrix factorization or random walks (e.g., DeepWalk [23], LINE [25]), GNNs provide superior performance on popular graph workloads such as node classification, link prediction and graph clustering.

A GNN model typically learns a *node representation*, also called *node embedding*, for each node in a graph and then use the embeddings to process downstream tasks such as node classification. During the training of a GNN model, we recursively update the embedding of each node by transforming the embeddings of its neighbors with a neural network and train the model parameters with back propagation. Existing GNN systems such as DGL [27], PyG [13], DGCL [6], Euler [4], NeuGraph [21] and ROC [17] generally suffer from the following limitations:

- **Obscure programming models.** The programming models of existing GNN systems are *tensor-centric* and *coarse-grained*. Specifically, both node embeddings and graph connectivity are represented as high-dimensional tensors and manipulated using coarse-grained operators that span the entire graph. Users are required to cast GNN computations defined for a center node (w.r.t. its neighbors) into global tensor operations, which is a task that should be handled by the GNN system for good usability.
- **Lack of holistic optimizations.** Existing GNN systems focus on developing efficient graph propagation kernels and delegate neural network computations to the underlying deep learning (DL) systems. Joint optimizations such as operator fusion are mostly ignored, and thus a large amount of intermediate data are materialized between operators, which results in excessive memory consumption and massive data movement in the memory hierarchy of GPUs.

We propose a new GNN training system, **Seastar** [28], which features two key designs to address the limitations of existing GNN frameworks. First, we introduce a *vertex-centric GNN programming model*, with which users only need to specify the operations on a single vertex using native Python expressions (without learning a new API as in existing GNN systems). GNN implementation in Seastar is simple and intuitive as users can write their code by

directly following the mathematical formula of a GNN model. We also built a visual "drag-and-drop" interface, through which users can easily construct GNN models by selecting and connecting operators to express the computation of a single vertex. Second, to improve training efficiency, we design a *generic kernel generator*, which produces high-performance kernels for both forward and backward passes. We propose *operator fusion* and *kernel-level* optimizations such as feature-adaptive thread mapping, locality-centric execution, and dynamic load balancing.

We will demonstrate that compared with existing GNN systems, Seastar achieves better usability, higher training throughput, and lower memory consumption. First, we will show how Seastar's vertex-centric visual interface and Python API simplify the development of various GNN models (e.g., GCN [19], GAT [26], APPNP [20], R-GCN [24]), by comparing with their implementations in existing GNN systems (e.g., DGL, PyG). Second, we will demonstrate Seastar's superior performance in terms of training throughput, peak memory consumption, and convergence speed across a wide range of models and datasets. Third, we will illustrate how Seastar's design and optimizations improve training efficiency, by visualizing the raw computational graphs extracted from user-defined functions and Seastar's optimized computational graphs, along with running time decomposition for various optimizations.

2 AN OVERVIEW OF SEASTAR

We give an overview of Seastar in this section, and readers may refer to [28] for details.

2.1 Vertex-centric GNN Programming

We denote a directed edge from vertex u to vetex v as uv and the neighbors of vertex v as $\mathcal{N}(v)$. In general, a layer in a GNN model can be expressed as follows:

$$h_v^{l+1} = g(\bigoplus_{u \in \mathcal{N}(v)} (f(h_u^l, h_{uv}, h_v^l))), \tag{1}$$

where h_v^l is the embedding of vertex v in layer l, h_{uv} is the feature vector associated with edge uv, f and g are customizable neural network modules or functions, and $\oplus$ is a function that aggregates the embeddings of v's neighbors and in-edges. We show the formula of GCN in Figure **??** as an instance of this general model. We show the formula of GCN below as an instance of this general model.

$$h_v^{(l+1)} = \sigma(b^{(l)} + \sum_{u \in \mathcal{N}(v)} \frac{1}{c_{uv}} h_u^{(l)} W^{(l)}). \tag{2}$$

The data flow pattern of GNN in Eq.(1), or Eq.(2), can be described as: propagate the embedding of each **s**ource vertex u along directed **e**dge uv, and **a**ggregate the embeddings of all sources to the center vertex v. The pattern has a **star** shape as the computation is vertex-centric with edges around a center vertex.

To implement Eq.(1) in a vertex-centric manner, users need to have access to the embeddings of u, v and edge uv, and freely invoke the library functions of the underlying DL system (e.g., PyTorch) to implement f, g, and $\oplus$. Instead of inventing domain-specific APIs as in existing GNN systems, Seastar enables users to perform these operations with native Python syntax. We show Seastar's implementation of GCN below as an example.

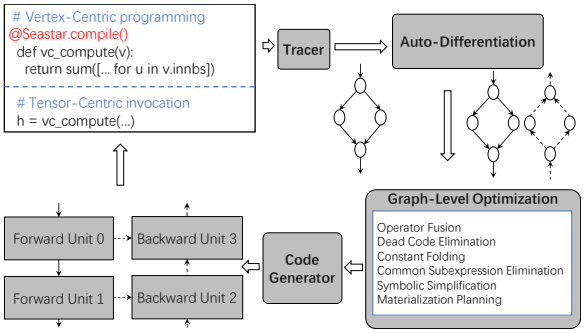

Figure 1: The architecture of Seastar

```
1 def seastar_compute(v):
2   h = sum([torch.mm(u.h,self.W)*u.norm for u in v.innbs])
3   if self.bias:
4     h = h + self.bias
5   if self.activation:
6     h = self.activation(h)
7 return h
```

Seastar's implementation begins by defining a Python function (a UDF) with a single argument, vertex v, in Line 1. Then Line 2 iterates over v's in-neighbors ($v.innbs$) using list comprehension in Python. We access the embedding ($u.h$) of each neighbor u using the *dot* operator in Python. Users can also access PyTorch's built-in functions (e.g., torch.mm) and variables defined in the enclosing GNN layer (e.g., $self.W$). We use the *sum* operator to aggregate over the list of embeddings in Line 2. Lines 3-6 add bias to aggregated embedding h and invoke activation function. Line 7 returns the new embedding. A comparison with DGL's implementation in [28] shows that Seastar's vertex-centric programming model is not only more user-friendly but also leads to more kernel-level optimizations.

2.2 System Architecture

Figure 1 depicts the system architecture of Seastar. Seastar generates efficient GPU kernels according to a vertex-centric function decorated by the "*compile()*" function decorator. Seastar's primary backend is MindSpore [16] — a DL system developed by Huawei. MindSpore embeds a DL compiler to fully exploit the computing power of hardware. It simplifies and accelerates distributed training by automatic model partitioning. Note that Seastar can also be integrated with other DL backends such as PyTorch. We explain the key components of Seastar as follows.

Tracer. The Tracer captures the operators in the UDF of a GNN implementation in a way similar to PyTorch's JIT tracing. We monkey-patch the operators such that their signatures are recorded as a node in the *computational graph* when they are invoked for the first time. Via the Tracer, we can rely on the DL backend for (i) verifying the correctness of a user program (e.g., whether the shape and data type of the inputs match the requirements of the operators) during the first run and (ii) inferring the output shape of the operators. We also need to determine whether an operator requires graph-dependent execution. Consider the *add* operator, when one of its inputs belongs to a center vertex v and the other belongs to v's neighbor u, we need to *add* the embeddings according to the

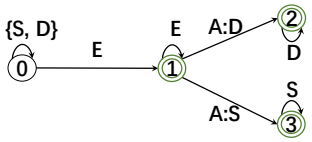

Figure 2: Finite state machine for operator fusion

edge connectivity. We use *graph type annotation* for tensors and operators for this purpose. Tensors are annotated with *S* (source), *D* (destination), *E* (edge) or *P* (parameter). S or D means that the tensor resides on a vertex and is accessed from u or v, respectively. E means that the tensor resides on edges. The graph-dependency of an operator can be inferred from the graph types of its inputs. More details can be found in Seastar's full paper in [28].

Optimizer. We embed an automatic differentiation engine to get the backward computational graph. The computational graphs (both forward and backward) are then optimized by the Seastar optimizer. Seastar conducts various optimizations such as operator fusion, dead code elimination, constant folding, common subexpression elimination, and more. We will discuss the most important optimization, i.e., *Seastar operator fusion*, in Section 2.3.

Code generator. Operator fusion groups multiple operators into one execution unit, and an optimized computational graph usually contains multiple such execution units. The code generator emits fused GPU kernels for an execution unit according to an *execution sketch* by instantiating placeholders in the sketch with scalar implementations of operators. For the other (i.e., not fused) operators, the code generator directly emits the corresponding operators in the DL system.

Runtime execution. Seastar stores graph data in compressed sparse row (CSR) format for its low memory consumption and high access efficiency. The embeddings of all vertices/edges are batched together as tensors, where a row of tensor corresponds to the embedding of a vertex/edge. When a user program invokes vertex-centric computation, Seastar traverses the computational graph and executes the operators following the operator dependency. We use a map to track the intermediate tensors and eagerly recycle tensors once no operators in both the forward and backward computational graphs rely on them.

2.3 Seastar Operator Fusion

Fusion opportunities. The *seastar pattern* naturally divides GNN computation into three stages: *S*, *E*, and *A*. Existing GNN systems have to explicitly materialize tensors at each stage boundary. Our key insight is that a pair of producer and consumer operators from consecutive stages (in the seastar pattern) can be fused if they are both injective. Take *S-E fusion* for example, which combines a producer in stage S and a consumer in stage E into a single operator. When executing stage-E operation alone, we first load the id of a source (or neighbor) vertex u into register and use it to query the ids of the center v and the edge uv in the CSR indexes. Then we use the retrieved ids to load the corresponding rows of tensors and conduct the computation. For a *fused S-E operator*, before performing stage-E computation, we execute the S-typed operator by querying the tensor using the id of u. The consumer-E operator can thus directly access the producer's output from register. Similarly, an *E-A fusion* first executes E operator, stores its output in register and then executes A operator.

Fusion algorithm. We summarize the valid fusions as the finite state machine shown in Figure 2. We differentiate E-A fusion into two sub-types, i.e., *A:D* (which returns a D-typed tensor) and *A:S* (which returns an S-typed tensor). This is because their fusible downstream operators are of different stages. For example, if a producer operator is in State 3, a fusible downstream consumer operator must be in stage S. A:S aggregation usually happens in the backward pass where gradients flow from destination to source vertex following the reverse direction of the edges. To conduct fusion, we start by sorting the computational graph in topological order and visit the nodes one by one. We set the root node's state by transitioning from State 0 according to its own graph type. Then for each visited operator, we recursively set the state of its children. The graph type of a child is set to its parent's state transitioned with its own graph type. When all nodes in the computational graph have been visited, we merge operators in State 1/2/3 with their fusible parent in a recursive manner.

2.4 Seastar Execution Sketch

Seastar execution sketch *sketches* how to map the fused operators to GPU threads, and placeholders are left for the code generator to fill in instructions according to the signatures of the fused operators. We adopt the following key optimizations in execution sketch.

Feature-adaptive thread grouping. Seastar forms a *feature-adaptive thread (FAT)* group using GPU threads with consecutive global ids. The group size is 2^k, where k is the largest integer satisfying $2^k \leq D$ and D is the dimension of the embedding associated with a vertex/edge. Note that D may vary significantly for different GNN layers and datasets. We instruct the threads in a FAT group to access consecutive memory addresses (e.g., a row in a tensor) and execute the same operation for coalesced memory access and SIMT execution. A thread's FAT group id is calculated by dividing its global thread id by the group size.

Locality-centric execution. When aggregating the neighbor/edge information for the vertices, DGL [27] maps several thread blocks to each edge. We found that this approach incurs a high synchronization overhead due to data racing for the same destination vertex and suffers from poor data locality because of the repetitive loading of the same destination vertex. To resolve these problems, Seastar conducts computations for edges incident to the same vertex sequentially by a single FAT group, and executes the computations for different vertices in parallel by different FAT groups.

Dynamic load balancing. Many real-world graphs have a power-law vertex degree distribution. This causes skewed workload distribution among the FAT groups in our *locality-centric* scheme. Our key insight is to conduct computations for the high-degree vertices earlier in order to overlap with (the computations of) low-degree vertices. To this end, Seastar sorts the vertices according to their degrees and processes them in descending order of their degrees. Another benefit of the degree sorting is that consecutive vertices have similar degrees, and thus consecutive FAT groups achieve good load balancing (intra-block/warp load balancing for small D).

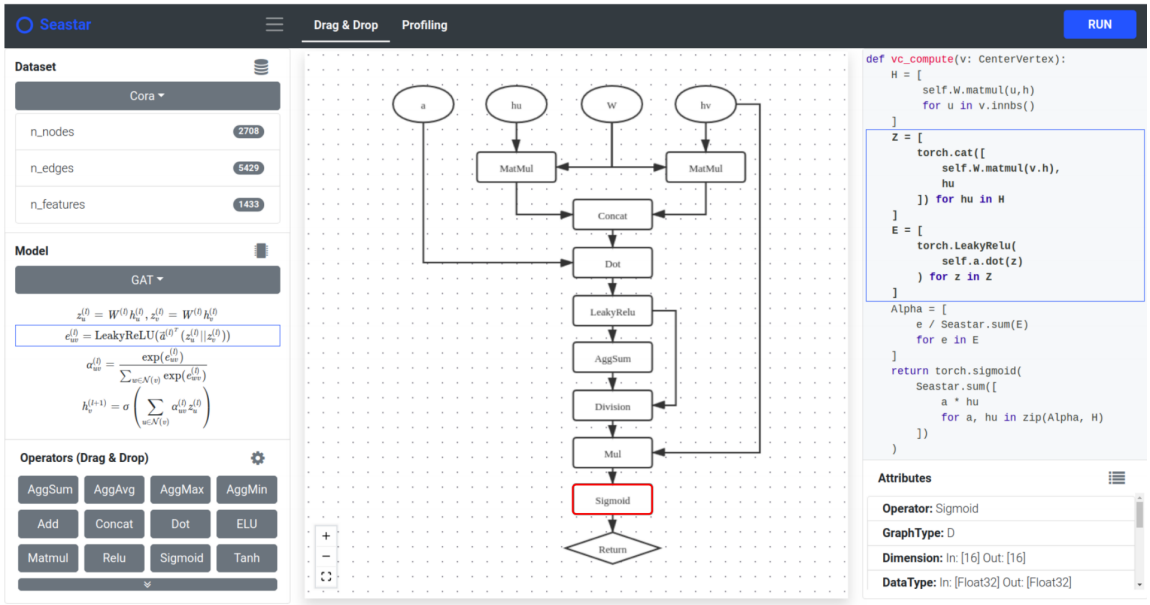

Figure 3: Seastar's user interface

3 THE DEMONSTRATION PLAN

In this demo, we will show SIGMOD attendees: (1) the good usability of Seastar's vertex-centric API in expressing various GNN models; (2) Seastar's advantages over existing GNN systems in training efficiency and resource utilization; and (3) Seastar's real-time performance monitor, with intuitive visualizations to understand how Seastar optimizes execution and why the optimizations improve performance. We will deploy Seastar (and the systems to be compared) on remote servers or cloud, while Demo attendees may interact with Seastar via a web-based front-end.

Usability of Seastar's API. We will show that Seastar's API leads to more intuitive and succinct implementations of popular GNN models (e.g., GCN, GAT, APPNP, R-GCN). Users can select a model on the left panel of the GUI in Figure 3 to show its mathematical formula along with its implementations in Seastar, with graphical display of the correspondence between the formula and the code. Such a display will help attendees see how simple it is to just follow a GNN formula to program in Seastar's vertex-centric API. When compared with the implementations in popular GNN systems (e.g., DGL and PyG), attendees may also understand the significance of designing a more user-friendly API for GNN programming. Novice users may also try to implement a GNN model using Seastar's "*Drag-and-Drop*" interface by dragging operators and connecting them following a GNN formula. If attendees are interested, we may also demonstrate the effectiveness of GNNs for graph analytics using applications such as node classification and graph clustering.

Performance comparison. Table 1 shows that Seastar achieves shorter per-epoch time and lower peak memory consumption than both DGL and PyG for training three models on the Reddit and Physics datasets. We will also show Seastar's much superior performance on 12 widely used datasets and other popular GNN models. Note that in all experiments we made sure that Seastar produced the same output and gradient as DGL and PyG to guarantee the correctness of generated kernel, and we used PyTorch as the DL backend for Seastar, DGL and PyG for fair comparison. By clicking the "*Run*" button, an attendee can execute a selected GNN model using Seastar and DGL/PyG in parallel on different GPUs. The demo interface will plot the accuracy-time curves of the systems in real time to show the performance difference in every stage. We will visualize Seastar's optimized computational graph and compare with that of DGL/PyG to explain why Seastar's fused operators achieves higher efficiency and lower memory usage.

Table 1: Performance comparison with DGL and PyG

		Epoch Time (ms)			Memory Consumption (MB)		
Model	**Dataset**	**DGL**	**PYG**	**Seastar**	**DGL**	**PYG**	**Seastar**
GAT	Physics	72.0	63.2	**32.8**	3440	6339	**2806**
	Reddit	-	-	**1182.1**	-	-	**7965**
GCN	Physics	5.8	9.3	**5.1**	1126	1376	**1111**
	Reddit	83.8	-	**38.5**	834	-	**722**
APPNP	Physics	37.5	16.5	**13.4**	2465	2459	**2443**
	Reddit	4100.5	-	**271.6**	5170	-	**2147**

Analysis on system design and performance. Seastar also offers a "*Profiling*" interface for users to understand its design and performance. Using this interface, attendees may see the differences between the computational graphs of a model *before* and *after* Seastar's optimizations, which allows attendees to understand how Seastar's optimizations work and which parts of the GNN training pipeline are modified. Attendees may also decompose the running time among the operators (will be plotted as a step-level breakdown in the "*Profiling*" GUI, not displayed in Figure 3 due to space limitation) to see where the most significant performance improvement comes from. Attendees can further disable/enable various optimizations to see their effects on the performance from the "*Profiling*" GUI.

Acknowledgments. We thank the reviewers for their constructive comments that help improve the quality of the paper. This work was supported by GRF 14208318 from the RGC of HKSAR.

REFERENCES

[1] JanusGraph, 2017. http://janusgraph.org/.
[2] Neo4J, 2018. https://neo4j.com/.
[3] OrientDB, 2019. https://orientdb.com/.
[4] Alibaba. Euler. https://github.com/alibaba/euler, 2020.
[5] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11, 2011.
[6] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu. DGCL: An Efficient Communication Library for Distributed GNN Training. In *EuroSys*, 2021.
[7] H. Chen, C. Li, J. Fang, C. Huang, J. Cheng, J. Zhang, Y. Hou, and X. Yan. Grasper: A high performance distributed system for OLAP on property graphs. In *SoCC*, pages 87–100, 2019.
[8] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. G-Miner: an efficient task-oriented graph mining system. In *EuroSys*, pages 32:1–32:12, 2018.
[9] H. Chen, X. Wang, C. Huang, J. Fang, Y. Hou, C. Li, and J. Cheng. Large Scale Graph Mining with G-Miner. In *SIGMOD*, pages 1881–1884, 2019.
[10] H. Chen, B. Wu, S. Deng, C. Huang, C. Li, Y. Li, and J. Cheng. High Performance Distributed OLAP on Property Graphs with Grasper. In *SIGMOD*, pages 2705–2708, 2020.
[11] A. Deutsch, Y. Xu, M. Wu, and V. Lee. Tigergraph: A native MPP graph database. *CoRR*, abs/1901.08248, 2019.
[12] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer. Weaver: A high-performance, transactional graph database based on refinable timestamps. *PVLDB*, 9(11):852–863, 2016.
[13] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop*, 2019.
[14] Z. Fu, Z. Wu, H. Li, Y. Li, M. Wu, X. Chen, X. Ye, B. Yu, and X. Hu. Geabase: A high-performance distributed graph database for industry-scale applications. In *Fifth International Conference on Advanced Cloud and Big Data, CBD 2017, Shanghai, China, August 13-16, 2017*, pages 170–175, 2017.
[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30, 2012.
[16] Huawei. Mindspore. https://e.huawei.com/us/products/cloud-computing-dc/atlas/mindspore, 2021.
[17] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In *MLSys*, 2020.
[18] T. Jin, Z. Cai, B. Li, C. Zheng, G. Jiang, and J. Cheng. Improving resource utilization by timely fine-grained scheduling. In *EuroSys*, pages 20:1–20:16, 2020.
[19] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *ICLR (poster)*, 2017.
[20] J. Klicpera, A. Bojchevski, and S. Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank. In *ICLR (poster)*, 2019.
[21] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai. Neugraph: Parallel deep neural network computation on large graphs. In *USENIX ATC*, pages 443–458, 2019.
[22] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
[23] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: online learning of social representations. In *SIGKDD*, pages 701–710, 2014.
[24] M. S. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks. In *ESWC*, volume 10843 of *Lecture Notes in Computer Science*, pages 593–607, 2018.
[25] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. LINE: large-scale information network embedding. In *WWW*, pages 1067–1077, 2015.
[26] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *ICLR (poster) 2018,*, 2018.
[27] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019.
[28] 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. In *EuroSys*, 2021.
[29] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.*, 7(14):1981–1992, 2014.
[30] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web, WWW*, pages 1307–1317, 2015.
[31] D. Yan, Y. Tian, and J. Cheng. *Systems for Big Graph Analytics*. Springer Briefs in Computer Science. Springer, 2017.