FlexGraph: A Flexible and Efficient Distributed Framework for GNN Training

Lei Wang[1]   Qiang Yin[1]   Chao Tian[1]   Jianbang Yang[1,2]   Rong Chen[2]
Wenyuan Yu[1]   Zihang Yao[1,2]   Jingren Zhou[1]

[1]Alibaba Group   [2]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

{jiede.wl, qiang.yq, tianchao.tc, wenyuan.ywy, jingren.zhou}@alibaba-inc.com

{jianbangyang, rongchen, esdeath}@sjtu.edu.cn

Abstract

Graph neural networks (GNNs) aim to learn a low-dimensional feature for each vertex in the graph from its input high-dimensional feature, by aggregating the features of the vertex's neighbors iteratively. This paper presents Flex-Graph, a distributed framework for training GNN models. FlexGraph is able to efficiently train GNN models with flexible definitions of neighborhood and hierarchical aggregation schemes, which are the two main characteristics associated with GNNs. In contrast, existing GNN frameworks are usually designed for GNNs having fixed definitions and aggregation schemes. They cannot support different kinds of GNN models well simultaneously. Underlying FlexGraph are a simple GNN programming abstraction called NAU and a compact data structure for modeling various aggregation operations. To achieve better performance, FlexGraph is equipped with a hybrid execution strategy to select proper and efficient operations according to different contexts during aggregating neighborhood features, an application-driven workload balancing strategy to balance GNN training workload and reduce synchronization overhead, and a pipeline processing strategy to overlap computations and communications. Using real-life datasets and GNN models GCN, PinSage and MAGNN, we verify that NAU makes FlexGraph more expressive than prior frameworks (*e.g.,* DGL and Euler) which adopt GAS-like programming abstractions, *e.g.,* it can handle MAGNN that is beyond the reach of DGL and Euler. The evaluation further shows that FlexGraph outperforms the state-of-the-art GNN frameworks such as DGL and Euler in training time by on average 8.5× on GCN and PinSage.

1 Introduction

A large number of real-world datasets can be naturally represented by graphs. Typical cases are knowledge graphs [7, 29], social networks [15, 36, 37], traffic networks [22] and web graphs [28, 43]. Recently, with the rapid growth of deep learning technologies, *graph neural networks* (GNNs), a rising family of models that directly apply neural networks on graph-structured data, have proved extremely effective yet efficient in handling various graph-related tasks [14, 17, 39, 40, 44, 47]. GNNs combine standard neural network (NN) operations and iterative graph propagation. Many GNN frameworks have been developed to address the challenges regarding the expressivity, efficiency, scalability and implementation in training GNN models [9, 13, 20, 24, 26, 38, 50].

Most GNN models fit into the "neural message passing" framework [11], in which each GNN layer is bound with a graph propagation process including two steps. For each vertex v in a graph, it first aggregates the features of v's "neighbors"[1] to compute a neighborhood representation using NN operations (see Figure 1a). Then it combines v's own feature with the neighborhood representation to adjust the feature of v via an update operation (see Figure 1b). As GNNs involve both NN and graph propagation operations, *i.e.,* features are propagated from the "neighbors", algorithm developers have to encode the graphs as sparse tensors and manually simulate the graph propagation using tensor operations in existing tensor-oriented deep learning frameworks, *e.g.,* TensorFlow [1], PyTorch [30] and MXNet [5]. This introduces an implementation challenge for GNN models.

---

[1]In GNNs, "neighbors" of each vertex are not limited to its direct 1-hop neighbors in the graph. See Section 2.2 for more general definitions of "neighbors" in GNNs.
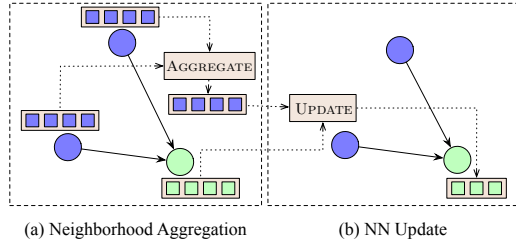
Figure 1. The computation of one target vertex (in green) in a GNN layer first aggregates 4-dimensional features of its neighbors (in purple), and then applies the UPDATE operation to get the new 3-dimensional feature.

Inspired by vertex-centric programming models, *e.g.,* GAS (Gather-Apply-Scatter) [12], applied in graph processing systems, some programming abstractions for GNNs have been proposed [9, 26, 38] to support GNN models succinctly. Analogously to GAS, such abstractions split the computation within a GNN layer into multiple stages. For each stage, users specify the computation that needs to be conducted over a set of vertices and their adjacent edges in a graph. All the current state-of-the-art GNN frameworks, *e.g.,* DGL [38], PyG [9], NeuGraph [26], ROC [20] and Euler [13], adopt GAS-like abstractions. They are able to express a few popular GNN models, *e.g.,* GCN [21], GIN [41] and G-GCN [27], which can be categorized as DNFA models in the sense that these models use *direct neighbors* (*i.e.,* 1-hop neighbors in the graphs) and enforce *flat aggregations* (*i.e.,* single-step operations).

Besides DNFA models, we find that GNN models have other various definitions of "neighbors" and neighborhood aggregation schemes. Some define *indirect neighbors* (*i.e.,* structured data in the graph other than the 1-hop neighbors) and apply *flat aggregation*, referred to as INFA models, *e.g.,* PinSage [43]; and another class of INHA models employs *indirect neighbors* and *hierarchical aggregation* (*i.e.,* multi-step operations when aggregating features from the "neighbors"), *e.g.,* MAGNN [10], P-GNN [45] and JK-Net [42]. The INFA and INHA models are as important as DNFA models in facilitating many graph-related tasks. For example, recent work [33, 34, 43] shows that INFA models like PinSage have been widely used in recommendation systems in industry, *e.g.,* Pinterest. Compared with state-of-the-art DNFA models, the INHA model P-GNN achieves much better performance in protein functional analysis over protein-protein interaction networks [45]. Unfortunately, current GNN frameworks are mainly designed for DNFA models only and lack the support for INFA and INHA models (see Section 2.3).

In this paper, we make a comprehensive analysis of various GNN models and establish a corresponding categorization for GNNs. Based on the analysis, we introduce a stage-based GNN programming abstraction NAU which splits the computation of a GNN layer into *NeighborSelection*, *Aggregation* and *Update* stages. NAU is capable of training DNFA, INFA and INHA models in a uniform manner. Different from GAS-like abstractions that directly utilize input graphs to capture the dependencies among vertices, NAU employs *hierarchical dependency graphs* (HDGs) to encode different kinds of "neighbors", not limited to direct neighbors. Users can define how each vertex chooses its "neighbors" and NAU constructs the HDGs accordingly. Moreover, users can specify an aggregation function for each level of the HDGs, and the neighborhood information is aggregated in a bottom-up manner automatically. Thus, NAU supports flexible neighborhood definitions and hierarchical aggregation schemes of GNNs.

To improve the performance of hierarchical aggregation, we propose a hybrid execution strategy. It differentiates the aggregations in the hierarchical scheme and completes them via graph processing, sparse NN operations or dense NN operations, depending on their different contexts. The idea of this strategy is to take advantage of both efficient graph processing and NN operations.

In addition, to efficiently train GNN models in a distributed environment, we develop an application-driven approach to balance the workload and reduce synchronization overhead. It divides the vertices of the input graph into disjoint sets by minimizing the estimated training and communication costs. We further provide a pipeline processing strategy that exploits partial aggregation to aggregate features in advance, and overlaps partial aggregation and communication to improve the throughput in distributed training.

We have implemented a distributed GNN framework FlexGraph upon NAU that adopts all optimization strategies mentioned above. We evaluated FlexGraph on training DNFA model GCN, INFA model PinSage and INHA model MAGNN over four real-life and synthetic graphs, and compared it with state-of-the-art NN framework PyTorch and GNN frameworks DGL, DistDGL and Euler. Experimental results show that for MAGNN, only FlexGraph can efficiently support its training on large graphs, due to the increased expressive power of NAU and effective storage and execution optimizations. For PinSage, FlexGraph outperforms others by on average 25.30× (up to 119.33×) in training time. For GCN that has been well supported in existing frameworks, FlexGraph can still achieve a at least 1.50× speedup.

Contributions. We summarize our contributions as follows.

(1) A categorization of GNNs that considers both the neighborhood definitions and aggregation schemes to reveal the key expressivity and performance challenges of GNN frameworks in training GNN models (Section 2).

(2) The GNN programming abstraction NAU that supports the training of DNFA, INFA and INHA models (Section 3).

(3) A hybrid execution scheme for hierarchical aggregation (Section 4), as well as an application-driven workload balancing strategy and a pipeline processing strategy for efficient distributed training (Section 5).

(4) An extensive evaluation of GNN framework FlexGraph that demonstrates its efficacy (Section 7).

2   Categorization of GNNs

In this section, we present a new 2-dimensional categorization for various GNN models regarding their neighborhood definitions and aggregation schemes. We start by reviewing GNNs (Section 2.1), and then give the categorization (Section 2.2). We also identify the challenges in supporting different GNN models under this categorization (Section 2.3).

2.1   Graph Neural Networks

Graph neural networks (GNNs) are a family of machine learning algorithms that apply neural network (NN) operations on graph-structured data. Initially, each vertex in a graph is associated with a high-dimensional feature vector. The goal of a GNN model is to learn a low-dimensional feature representation for each vertex, which can be further fed into various downstream tasks, *e.g.,* vertex classification [21], link prediction [46] and vertex clustering [2].

Similar to traditional neural networks, a GNN model stacks multiple GNN layers to update the vertex features iteratively, in which each successive layer uses the outputs of its previous layer as inputs. Here both the inputs and outputs of a layer contain the features of all vertices. In each GNN layer, the new feature of each vertex is computed by aggregating the features of its "neighbors" from the previous layer. Given a graph G with raw input features X_* for the vertices in G (*e.g.,* word embeddings in the Reddit dataset), the computation within the k-th GNN layer can be expressed as follows:

$$a_v^{(k)} = \text{AGGREGATE}^{(k)}(\{h_u^{(k-1)} | u \in \mathcal{N}(v)\}), \quad (1)$$

$$h_v^{(k)} = \text{UPDATE}^{(k)}(h_v^{(k-1)}, a_v^{(k)}), \quad (2)$$

where $\text{AGGREGATE}^{(k)}$ and $\text{UPDATE}^{(k)}$ are two operations conducted during the k-th GNN layer for $k > 1$, $h_v^{(k)}$ and $a_v^{(k)}$ denote the feature vector and neighborhood representation of vertex v at the k-th layer, respectively, and $\mathcal{N}(v)$ denotes the "neighbors" of v.

The operation $\text{AGGREGATE}^{(k)}$ is also referred to as *neighborhood aggregation at layer k* (see Figure 1a). Intuitively, it gathers the features of v's "neighbors" using accumulation functions to produce neighborhood representation $a_v^{(k)}$. This is followed by the invocation of $\text{UPDATE}^{(k)}$, which computes the new feature $h_v^{(k)}$ by combining v's previous feature $h_v^{(k-1)}$ and the newly computed $a_v^{(k)}$ (see Figure 1b). Note that $\text{AGGREGATE}^{(k)}$ involves both graph propagation, *i.e.,* features are propagated from the "neighbors" to v, and NN operations, while $\text{UPDATE}^{(k)}$ only includes NN operations.

2.2   A 2-Dimensional Categorization

There has been a lot of GNN models proposed for different applications [10, 21, 27, 41, 43, 45] (see also a recent survey [48]). However, to the best of our knowledge, no categorization exists for GNN models that takes both the static and operational characteristics of GNNs into account. In general, there are no constraints imposed on how to define
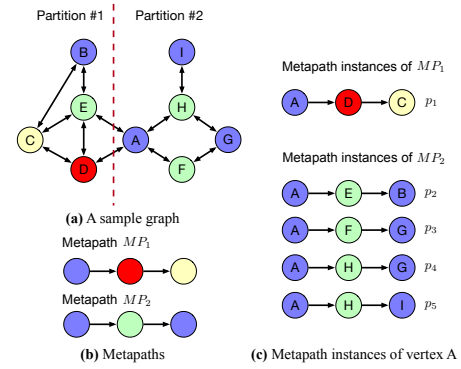

Figure 2. A sample graph to illustrate different GNN models.

the neighborhood, *i.e.,* "neighbors" of vertices using static expressions and how the neighborhood aggregation behaves in a GNN model. Therefore, we can divide GNN models into three categories based on their static *neighborhood definitions* and operational *aggregation schemes*.

(1) DNFA *(direct neighbors with flat aggregation).* For GNN models that fall into this category, each vertex's neighborhood consists of all its 1-hop neighbors, *i.e., direct neighbors* in the input graph; and the aggregation function only conducts a single-step operation, *i.e., flat aggregation.* Representative DNFA models include GCN [21], GIN [41], and G-GCN [27]. Take GCN as an example and consider vertex A in the sample graph depicted in Figure 2a. In every layer of GCN, we have that $\mathcal{N}(A) = \{D, E, F, H\}$. To produce the neighborhood representation of A, AGGREGATE just sums the features collected from the vertices in $\mathcal{N}(A)$.

(2) INFA *(indirect neighbors with flat aggregation).* For INFA models, a "neighbor" of a vertex can be a vertex from a nearby subgraph, instead of its actual 1-hop neighbor. Analogously to DNFA, neighborhood information is aggregated in a flat manner. For example, PinSage [43] is an INFA model having an architecture similar to GCN. It generates neighborhood representations by summing neighbors' features as in GCN. However, PinSage adopts an importance-based neighborhood definition. It defines $\mathcal{N}(v)$ as the top-k visited vertices in several random walks starting from v. Let $k = 2$, then $\mathcal{N}(A) = \{C, G\}$ for the sample graph shown in Figure 2a, since C and G have the highest visit counts. Both C and G are *indirect neighbors* of A as there are no edges directly connecting C and G to A.

(3) INHA *(indirect neighbors with hierarchical aggregation).* In an INHA model, a "neighbor" can be any graph-structured data. In the operations of neighborhood aggregation, multi-step aggregation operations, *i.e., hierarchical aggregations,* are allowed. MAGNN [10], P-GNN [45] and JK-Net [42] fall into this category.

In MAGNN, the "neighbors" of a vertex are a set of metapath instances that match the *methpaths* defined by the model. Each metapath is an ordered sequence of vertex types. In Figure 2a, vertices of different types have different colors; and

Figure 2b shows two metapaths MP_1 and MP_2. As a result, $\mathcal{N}(A) = \{p_1, \ldots, p_5\}$ (Figure 2c), *i.e.,* five instances match the metapaths by the vertex colors.

The aggregation operation of MAGNN is conducted in a *hierarchical* way. More specifically, when computing the neighborhood representation of vertex A in the sample graph of Figure 2a, (i) MAGNN first derives the representations of instances $p_1, \ldots, p_5$ by gathering features of the vertices that belong to each instance, respectively; (ii) it then combines these representations of the same metapath type to compute the representation of MP_1 and MP_2; and (iii) finally it aggregates the representations of MP_1 and MP_2 to get the neighborhood representation of vertex A.

Note that there should be a fourth category DNHA, *i.e., direct neighbors with hierarchical aggregation*. For brevity, we treat DNHA as a special case of INHA. Furthermore, to the best of our knowledge, no existing GNN model falls into this category. Also observe that DNFA can be a special case of INFA. We distinguish DNFA from INFA on purpose, since existing GNN frameworks mostly focus on DNFA only. As a result, we only discuss the first three categories in this work.

Observation. Clearly one can see that GNN models have huge diversity in the neighborhood definitions and aggregation schemes. On the one hand, one vertex's "neighbor" can be its 1-hop neighbor (*e.g.,* GCN), another vertex without direct connecting edge (*e.g.,* PinSage), or even a path (*e.g.,* MAGNN). On the other hand, neighbors' features can be aggregated in a flat manner (*i.e.,* single-step aggregation) like in GCN and PinSage, or in a hierarchical manner (*i.e.,* multi-step aggregation) such as in MAGNN. This categorization *w.r.t.* neighborhood definitions and neighborhood aggregation schemes reveals both expressivity and performance challenges for GNN frameworks. On the expressivity side, the hierarchical dependencies among vertices should be captured by the framework. On the performance side, efficient access to indirect neighbors and their features is a crucial issue.

2.3 Challenges

As indicated in Section 2.1, GNN models call for not only NN operations but also graph propagation. It is hence desirable to have a framework that achieves both the two types of operations simultaneously with good performance; and better still effectively supports GNN models from different categories (Section 2.2). However, this is non-trivial.

Existing tensor-based deep learning frameworks lack intuitive support for graph propagation. Inspired by the vertex-centric programming abstraction GAS [12] used in graph processing systems, some GAS-like programming abstractions for GNNs [9, 26, 38] have been recently proposed to accomplish both graph propagation and NN operations. As an example, SAGA-NN is presented in the GNN framework NeuGraph [26]. It extends the classical GAS abstraction for graph computation with NN operations. SAGA-NN splits

the computation within a GNN layer into 4 stages: *Scatter*, *ApplyEdge*, *Gather*, and *ApplyVertex*. Here the first 3 stages aim to support AGGREGATE operations (Equation (1)), while the last *ApplyVertex* corresponds to the UPDATE NN operations (Equation (2)). To compute neighborhood representations, each vertex first *scatters* its feature along its outgoing edges (also include incoming edges sometimes) to produce a feature for each edge; it then *applies* NN operations to generate new edge representations; at last each vertex *gathers* features of its incoming edges and combines them as its neighborhood representation. In the UPDATE operation, each vertex assembles its neighborhood representation and its own previously-computed vertex feature to compute its updated vertex feature via *ApplyVertex*.

Although the GAS-like GNN programming abstractions (*i.e.,* SAGA-NN and its variants) perform well on DNFA models and have been widely used in most existing GNN solutions [48], including DGL [38], PyG [9], NeuGraph [26] and Euler [13], they are essentially designed for the DNFA category only. Due to the fact that in SAGA-NN, each vertex computes its feature by aggregating the features of all its 1-hop neighbors in a flat manner, it is hard to express the computations of INFA and INHA models using SAGA-NN directly. For instance, there is no efficient implementation of PinSage with SAGA-NN because the "neighbors" are beyond the 1-hop scope under this case. Indeed, DGL implements PinSage by simulating random walks with several graph propagation stages of SAGA-NN, which is very inefficient. Note that PinSage and GCN share the same aggregation function, and each vertex in PinSage has a relatively small size of neighbors. However, the training time of one epoch for PinSage is over 10 times compared with GCN in DGL with SAGA-NN abstraction on dataset Reddit. Over 95% of total training time is used to simulate random walks and get features of indirect neighbors (see details in Section 7.1). There are also approaches that utilize an additional sampling phase to obtain the relationships with indirect neighbors as tensor structures and simulate graph propagation through tensor operations [13, 50]. However, they fail to apply some efficient graph-related operations due to the lack of graph structure (see performance analysis of PinSage in Section 7.1).

Similarly, INHA models are beyond the reach of GAS-like GNN abstractions for the existence of various forms of "neighbors" and the multi-step aggregation operations.

3 A Flexible GNN Framework

We propose a flexible GNN framework FlexGraph to tackle the challenges given in Section 2.3. In a nutshell, FlexGraph employs *hierarchical dependency graphs* (HDGs) to accommodate both the diversified definitions of "neighbors" and the hierarchical aggregation schemes in GNN models (Section 3.1). Underlying FlexGraph is a new GNN programming abstraction called NAU (Section 3.2). In this new abstraction, FlexGraph first builds HDGs for the input GNN model;
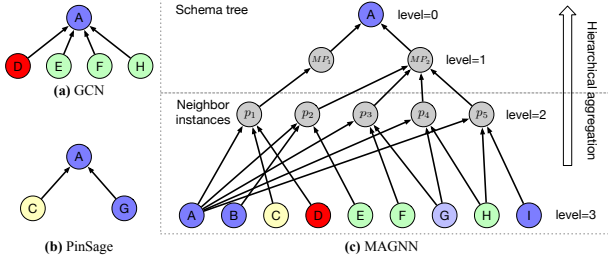
Figure 3. The HDG(A) for vertex A in different models.

guided by HDGs, FlexGraph aggregates the features from the "neighbors" and updates all vertex features iteratively in a uniform manner. With NAU, GNN models of different categories can be expressed naturally in FlexGraph (Section 3.3).

3.1 Hierarchical Dependency Graph

Given a GNN model $\mathcal{M}$ and an input graph G, a *hierarchical dependency graph* (HDG) *for* $\mathcal{M}$ w.r.t. *a vertex v in G* is a directed acyclic graph (DAG), denoted as HDG(v). It characterizes how the feature of v is aggregated from its "neighbors" in the GNN model $\mathcal{M}$. The collection of all HDG(v) for vertices v in G is referred to as HDGs.

The DAG HDG(v) uses a hierarchical structure, where both the top and bottom levels contain vertices from the input graph G (see Figure 3). Each HDG(v) consists of two parts: a *schema tree* and a set of *neighbor instances*.

(1) The *schema tree T* of HDG(v) encodes the hierarchical (tree) structures for neighbor types defined by the GNN model $\mathcal{M}$. Specifically, vertex v is treated as the root of T and each leaf of T represents a neighbor type in $\mathcal{M}$. We stipulate $T = v$ when T has a single neighbor type.

(2) The *neighbor instances* of HDG(v), denoted as $\mathcal{N}_v$, is a collection of the "neighbors" of v defined by model $\mathcal{M}$. Each neighbor instance in $\mathcal{N}_v$ refers to a vertex in HDG(v) and is linked to a leaf in T that encodes its type. If a neighbor instance does not follow the flat structure, *i.e.,* it is beyond a single vertex from the input graph G, then HDG(v) connects all its associated vertices in G to the instance.

For GNN models that fall into DNFA and INFA categories, their schema trees and neighbor instances have flat structures, *i.e.,* they only contain vertices from the input graph (see Figures 3a-3b). In contrast, both the schema trees and neighbor instances may have hierarchical structures for the models in INHA category. For example, the HDG(A) for MAGNN *w.r.t.* vertex A of the sample graph of Figure 2a is shown in Figure 3c. Observe that the top and bottom levels of HDG(A) have vertices from the input graph. The root of HDG(A) is vertex A itself and each of the two vertices at level 1 represents a metapath type. These three vertices compose the schema tree of HDG(A). The vertices at level 2 in HDG(A) encode neighbor instances, *i.e.,* metapath instances of different types, and each one is linked to a corresponding vertex at level 1 indicating its type, *e.g.,* p_2

```
interface GNNLayer(){
  //select neighbors and build HDGs
  NeighborSelection(g,schema, nbr_udf) → HDGs;
  // aggregate nbr features guided by HDGs
  Aggregation(feas^(k-1), HDGs) → nbr_feas^(k);
  // NN update
  Update(feas^(k-1), nbr_feas^(k)) → feas^(k);
}
```

Figure 4. The user interface of NAU.

is an instance matching MP_2. In addition, HDG(A) connects the vertices of the input graph to the metapath instances that they lie in, *e.g.,* vertices A, C, and D are linked to p_1.

3.2 Programming Abstraction NAU

FlexGraph adopts a new GNN programming abstraction NAU. As opposed to GAS-like abstractions, NAU introduces a *NeighborSelection* stage to build HDGs to capture hierarchical dependencies among vertices. Having HDGs in place, in the *Aggregation* stage, NAU asks users to specify an aggregation function as a UDF (user-defined function) for each vertex in the HDGs. That is, users provide a UDF for (i) each vertex in the schema trees and (ii) each neighbor instance in the $\mathcal{N}_v$'s, regardless of whether they have flat structures or not. Then NAU automatically executes the hierarchical aggregation of the GNN model based its HDGs, by applying these UDFs in a bottom-up fashion. In this way, NAU supports both flexible neighborhood definitions and hierarchical neighborhood aggregation schemes of various GNNs.

The Workflow of NAU. NAU consists of three stages for each GNN layer, *i.e., NeighborSelection*, *Aggregation* and *Update* (see Figure 4). The *NeighborSelection* and *Aggregation* stages specify the computation of AGGREGATE operation in Equation (1), while the *Update* stage involves the UPDATE operation of Equation (2).

NeighborSelection. This stage builds HDGs to meet the requirements of various neighborhood definitions in GNN models. Taking an input graph g, a schema tree, *i.e.,* schema, of the GNN model, and a vertex UDF nbr_udf as inputs, this stage outputs the HDGs for the GNN model. Here the UDF nbr_udf customizes how each vertex retrieves its "neighbors" from the input graph g.

To build HDGs, users are requested to implement a UDF for each neighbor type associated with the leaves in the schema tree. Here a "neighbor" can be a single vertex or a set of vertices, *e.g.,* paths of the input graph. Figure 5 shows the UDFs for GCN, PinSage and MAGNN, respectively. One can see that each UDF takes a schema_tree of the GNN model and a vertex v as inputs. It returns the "neighbors" of v and their corresponding neighbor types as outputs. For example, the UDF for PinSage first starts num_traces many random walks of length n_hops from v; then it selects the vertices with top_k highest visit counts as v's "neighbors". These "neighbors" have a flat structure. Thus their type is simply

```
def gnn_nbr(schema_tree, v): ## nbr_udf for GCN
    return nbr(v.neighbors, type="vertex")

def pinsage_nbr(schema_tree, v): ## nbf_udf for PinSage
    for i in range(num_traces):
        path = random_walk(start=v, hops=n_hops)
        for node in path.nodes:
            visited_frequency[node]++
    return nbr(top_k(visited_frequency), type="vertex")

def magann_nbr(schema_tree, v): ## nbr_udf for MAGNN
    for meta_path in schema_tree.leaves:
        for p in search_paths(start=v, length=meta_path.
            length):
            if path_match(p, meta_path):
                nbr.append(p.nodes, type=meta_path.type)
    return nbr
```

Figure 5. UDFs for neighbor types of different models.

```
def Aggregation(feas, HDGs):
    for i in range(HDGs.depth, 1):
        g_s = HDGs.sub_graph(level=i)
        feas = apply(g_s, feas_i, aggr_udf_i)
    return feas
```

Figure 6. Implementation of `Aggregation`.

vertex. Note that this may involve sophisticated graph computations, *e.g.,* it is required to find paths matching a specific pattern (metapath) in constructing HDGs for MAGNN (see Figure 5). This is clearly out of the reach of NN operations.

Aggregation. Given the features $\text{feas}^{(k-1)}$ from the previous layer and the HDGs built by *NeighborSelection*, this stage computes the neighborhood representation $\text{nbr\_feas}^{(k)}$.

With HDGs, users are allowed to specify multi-step aggregation operations in a bottom-up way, beyond the single-step ones. In various GNN models, vertices at the same level of HDGs often, if not always, share the same operational semantics, *i.e.,* aggregation function. Thus a typical implementation of the `Aggregation` function follows the level-wise paradigm, shown in Figure 6. Starting from the bottom level, each time it first obtains a subgraph `g_s` of the HDGs at a specific level i, which includes vertices at levels i and $i - 1$, as well as the edges connecting them. It then applies a user defined aggregation function `aggr_udf_i`, which takes the subgraph `g_s` and features `feas_i` of vertices at level i as inputs and computes the features for vertices at level $i - 1$. In the end, it returns the features of the roots of the HDGs as the neighborhood representation.

Update. This stage combines the old features $\text{feas}^{(k-1)}$ and the neighborhood representations $\text{nbr\_feas}^{(k)}$ by NN operations. It outputs new representations $\text{feas}^{(k)}$ for the next layer, using NN operations only.

Discussion. SAGA-NN can be considered as a special case of NAU, where in stage *NeighborSelection*, each vertex selects its all 1-hop neighbors and all such neighbors have the same

```
class GCNLayer(GNNLayer):
    def Aggregation(feas, HDG):
        dst_ids, src_ids = HDG.sub_graph(level=1)
        nbr_feas = scatter_add(feas[src_ids], dst_ids)
        return nbr_feas
    def Update(feas, nbr_feas):
        return ReLU(W * feas.add(nbr_feas))

class PinSageLayer(GNNLayer):
    def Aggregation(feas, HDG):
        dst_ids, src_ids = HDG.sub_graph(level=1)
        nbr_feas = scatter_add(feas[src_ids], dst_ids)
        return nbr_feas
    def Update(feas, nbr_feas):
        return ReLU(W * CONCAT(feas, nbr_feas))

class MAGNNLayer(GNNLayer):
    def Aggregation(feas, HDG):
        udf = [scatter_mean, scatter_softmax, scatter_mean]
        for i in range(3,1):
            g_s = HDG.sub_graph(level=i);
            nbr_feas = apply(g_s, nbr_feas, udf[i])
        return nbr_feas
    def Update(self, nbr_feas, h):
        return ReLU(W * nbr_feas)
```

Figure 7. GCN, PinSage and MAGNN in NAU.

type. In stage *Aggregation*, all neighborhood features are aggregated via single-step operation. Moreover, NAU does not require the users to define or execute stage *NeighborSelection* in every GNN layer. This is because for some GNN models, *e.g.,* PinSage, the HDGs can be cached and shared among some layers, *e.g.,* in one epoch. They can be shared even during the entire training process, *e.g.,* MAGNN, because they do not change across training iterations. Under such circumstances, a specific layer can directly utilize the results of previous *NeighborSelection* stage, making NAU more flexible.

As another example to demonstrate the expressiveness of NAU, we show that two popular INHA models P-GNN [45] and JK-Net [42] can be succinctly expressed in NAU. In P-GNN, each "neighbor" of a vertex is defined as an "anchor-set" containing several vertices, and each vertex has k anchor-sets as its neighbors. In stage *Aggregation*, each vertex first aggregates features of vertices from the same anchor-set to produce the feature of each anchor-set, then combines features of its k anchor-sets to compute its neighborhood feature. The HDGs for P-GNN have three levels, where each vertex at level 1 (resp. 2) represents an anchor-set (resp. a vertex from the input graph). With such HDGs, the *Aggregation* stage performs the hierarchical aggregation in a bottom-up fashion as in MAGNN. In JK-Net, each vertex v has k "neighbors", and its i-th neighbor contains all vertices whose shortest path length to v is i. In the *Aggregation* stage, each vertex first combines features of vertices from the same "neighbor" to generate the feature of each neighbor, then aggregates features of its all k neighbors. The neighborhood aggregation schemes of P-GNN and JK-Net are quite similar. Thus JK-Net can also be easily expressed by NAU.
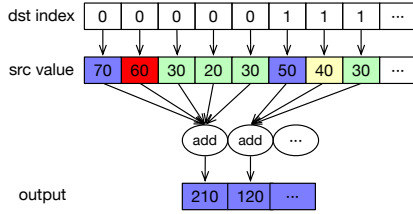
Figure 8. Example tensor scatter operation.

3.3 Expressing GNN Models

As a proof of concept, we give NAU programs for various GNN models. Figure 7 outlines their sample implementations for GCN, PinSage and MAGNN. We only show the implementations of Aggregation and Update functions, as UDFs of NeighborSelection function are provided in Figure 5.

Remark. Note that in stage *Aggregation*, the subgraph of the HDGs *w.r.t.* a specific level can be encoded in the classical co-ordinate list (COO) format, and each vertex deduces its neighborhood representation by using scatter operations [31], *e.g.,* scatter_add. Indeed, scatter operations can be achieved by standard reduce operations using a "value" tensor and an "index" tensor, where for each element in tensor *value*, its index is specified by the corresponding value in tensor *index*. By doing so, all elements with the same index in tensor *value* are aggregated via a given accumulation operation. Figure 8 illustrates the computation in scatter_add operation. With COO format, for each vertex at level i, we can first obtain its relevant vertices at level $i + 1$ since they have the same destination index in COO. Then we aggregate the features of these vertices using scatter operations.

4 Hierarchical Aggregation

We next show how FlexGraph conducts hierarchical aggregation in a space and time-efficient way, starting with the storage mechanism of HDGs employed in FlexGraph.

4.1 Construction of HDGs

As indicated in Section 3, conceptually the *NeighborSelection* stage is responsible for building HDGs. FlexGraph carries out this by using a set of *formatted records*, and each record represents a "neighbor" of a vertex in GNN models. From the results, *i.e.,* "neighbors" returned by the UDFs in the *NeighborSelection* stage, it first creates a set of records in the form of (root, nei = [leaf_0, leaf_1, . . . , leaf_n], nei_type). Here each record is a tuple consisting of three elements, including (i) the root vertex root who owns the "neighbor" nei, (ii) the type of the "neighbor" nei_type (recall that there may exist multiple neighbor types, *e.g.,* metapath types in MAGNN), and (iii) n vertices leaf_0, leaf_1, . . . , leaf_n from the original graph linked to the "neighbor" nei (*i.e.,* leaves in HDGs). Then FlexGraph constructs HDGs using these records in a top-down manner, where the schema tree pre-defined by a given GNN model is firstly included. After that, the nei and
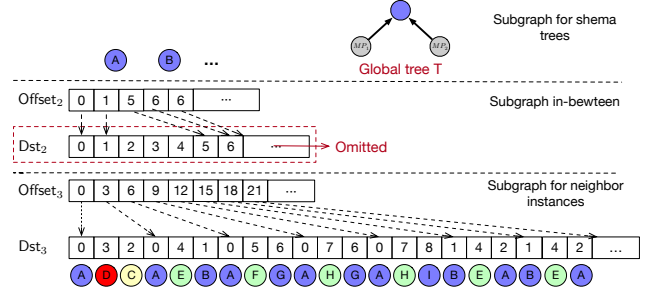

Figure 9. Data storage of HDGs in MAGNN.

leaf_i's in each record are encoded as vertices and added to HDGs. Finally, it includes the edges from nei to nei_type, and from each leaf_i to nei.

HDG storage optimization. To support hierarchical aggregation, FlexGraph needs to store a subgraph for each level (see Figure 6). Clearly, each such subgraph can be stored using a classical CSC (compressed sparse column) format [3]. To do this, FlexGraph orders the vertices in the same level of HDGs, and the *local rank* of each vertex v is defined as v's order in the level that it resides in. For example, the local rank of vertex p_4 in Figure 3c is 3, as it is the fourth vertex at level 2 and the local ranks start from 0. Having defined local ranks, each subgraph at level i can be represented by two arrays: (i) one vertex array Dst_i consisting of local ranks of destination vertices; (ii) and one offset array $Offset_i$ for source vertices that defines the destination vertices of their associated edges.

Observe that HDGs have the following distinctive properties: (a) the HDGs *w.r.t.* different root vertices share the same schema tree structure, (b) all vertices in HDGs only have one outgoing edge except for those at the bottom level. In light of this, we further revise the CSC format as follows to get a more space-efficient storage.

(1) Subgraph of neighbor instances. This subgraph consists of edges between level max_level and max_level−1. It is stored as CSC format by two arrays as above. For the MAGNN case (Figure 2), we have max_level = 3 and this subgraph is represented by arrays Dst_3 and $Offset_3$ (see Figure 9).

(2) Subgraph in-between. This subgraph of HDGs connects the neighbor instances and schema trees. It clearly can be stored as the above one, *e.g.,* with an offset array $offset_2$ and a vertex array Dst_2 for the MAGNN case. However, the vertex array can actually be *omitted* for space efficiency. Indeed, observe that each element in the vertex array that represents a neighbor instance, *e.g.,* $p_1,. . . , p_5$ in Figure 3, has exactly one outgoing edge. As a result, we can order the source vertices consecutively according to the destination vertices and the vertex array Dst_2 becomes an ordered array and can be omitted (see Dst_2 in Figure 9). That is, to represent this subgraph, it suffices to keep the offset array only, *e.g.,* $offset_2$, and omit the vertex array.

(3) Subgraphs for schema trees. FlexGraph does not store the subgraphs for levels starting from leaves of schema trees. It

only keeps a single *global* schema tree such that it is reused by all roots in HDGs, rather than maintaining multiple physical copies. As shown in Figure 9, all root vertices share a same schema tree structure consisting of 3 vertices, a root vertex and 2 children indicating two metapath types MP_1 and MP_2.

4.2 Hybrid Execution

With the HDGs that are stored efficiently, FlexGraph applies a hybrid processing strategy for hierarchical aggregation. It distinguishes the aggregation operations in the hierarchy according to their different contexts, which demand distinct approaches for better performance.

(1) Feature fusion operation for neighbor instance level. The first step of hierarchical aggregation is to aggregate the features of "pure" vertices at level max_level of HDGs. It outputs the features for the neighbor instance vertices. Note that each "pure" vertex is taken from the original input graph and is usually linked to multiple neighbor instance vertices (see Figure 3). When performing this aggregation with widely-used sparse tensor operations [31] (*e.g.,* scatter operations in Section 3.3), the features of all related "pure" vertices need to be collected and materialized along their adjacent edges in HDGs prior to doing the actual aggregation. It often leads to memory explosion when the input graph becomes larger. Take GCN and the Reddit dataset which contains 232K vertices and 114M edges as an example. When implementing GCN via sparse tensor operations, each vertex first sends a message (*i.e.,* its vertex feature) to its outgoing edges as edge features, then the vertices aggregate the messages (*i.e.,* edge features) received from their incoming edges. Such sparse tensor operations require that messages be explicitly materialized, resulting in increased memory consumption by about 500× for the features of "pure" vertices.

Inspired by the vertex reduce operations in graph processing systems, FlexGraph exploits *vertex feature fusion* when executing this first-step aggregation. This is very useful since most of GNN models apply simple aggregation functions after gathering "pure" vertex features, *e.g.,* sum, min, max and mean. The idea of vertex feature fusion is similar to the kernel fusion optimization applied in DGL [38]. Specifically, each thread first loads the feature of some source "pure" vertices from level max_level into the per-thread local memory. Using HDGs, FlexGraph then appends them to the buffers *w.r.t.* the designated destination vertices at level max_level − 1; and performs the aggregation within the buffers directly. In this way, FlexGraph can eliminate the overhead of materializing vertex features on massive edges.

(2) Sparse NN operation for intermediate level. In the second step in hierarchical aggregation, FlexGraph aggregates the features of neighbor instances to get the features for leaves of schema trees. Unlike the neighbor instance level, each neighbor instance at the intermediate level is linked to only one schema tree leaf. Therefore, sparse NN operations do not
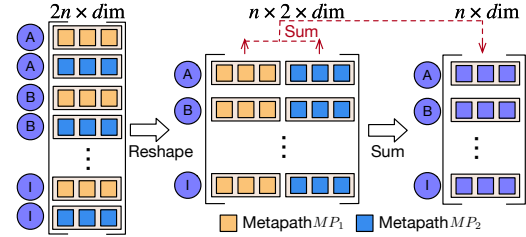

Figure 10. Example dense tensor operation in *Aggregation*.

cause substantial memory overhead like neighbor instance level. Under such condition, FlexGraph encodes HDGs at the intermediate level as sparse tensor format, and performs *sparse NN operations* to achieve aggregation of neighbor instance features in this step, as showcased in Section 3.3.

(3) Dense NN operation for schema level. After the first two steps, all subsequent steps in hierarchical aggregation are conducted over the vertices in the schema trees of the HDGs. As mentioned above, the schema tree has a fixed regular form shared by all roots in HDGs. It is known that the dense NN operations are more efficient in current deep learning frameworks than the sparse ones but require a regular form of the input. This is fully satisfied by the aggregation *w.r.t.* features of the vertices in the schema tree. Therefore, Flex-Graph carries out all the remaining aggregation steps via *dense NN operations* to improve performance.

As an example, consider the schema tree for MAGNN shown in Figure 3, where each root vertex in HDGs is incident to 2 types of metapath. For an input graph having n vertices, *i.e.,* roots in HDGs, the raw input for aggregating metapath types can be represented by a tensor of shape $2n \times$ dim, where dim refers to the feature dimension of vertices at level 1. If each root vertex needs to sum all the relevant features of metapath types, we can obtain a new tensor by reshaping the raw tensor to $n \times 2 \times$ dim. Here the reshaping operation only changes the logical layout of tensors, and does not involve memory copy. We next sum each row of the new tensor in the first dimension to get the results (see Figure 10).

5 Distributed GNN Training

FlexGraph adopts a shared-nothing architecture for distributed GNN training. Given a graph $G = (V, E)$, where V and E are the vertex and edge sets of G, FlexGraph divides V into k disjoint partitions. For each partition of vertices, FlexGraph constructs a subgraph of HDGs in parallel. Such HDGs are assigned to k shared-nothing workers for distributed training. Messages are exchanged to synchronize features at the end of each GNN layer.

To improve the performance of distributed GNN training with FlexGraph, we further develop two main optimizations: workload balancing and pipeline processing. Observe that the ideas of both optimizations are not conceptually new and have been commonly used to improve the performance of distributed systems. Here we translate these generic ideas
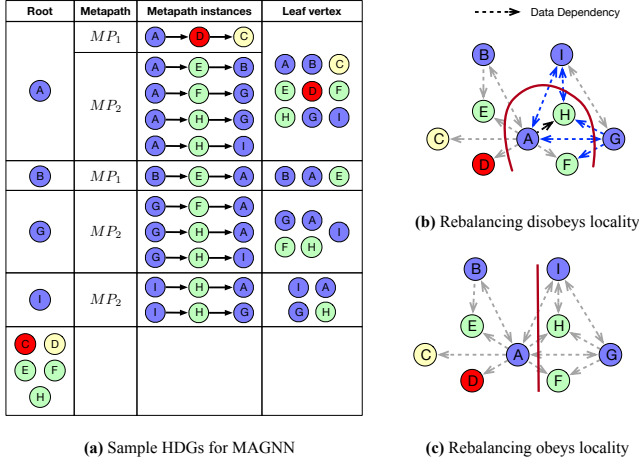
(a) Sample HDGs for MAGNN

(b) Rebalancing disobeys locality

(c) Rebalancing obeys locality

Figure 11. Workload balancing in FlexGraph.

and principles into effective optimization techniques for distributed GNN training with HDGs.

Workload balancing. As we have seen in Section 2.2, GNN models exhibit diversified definitions of "neighbors" and aggregation schemes. Conventional graph partitioning usually takes static metrics, *e.g.,* vertex weight or edge weight, as a balance indicator, which is insufficient to ensure workload balancing for GNN model training. Inspired by Fan *et al.* [6], FlexGraph adopts an application-driven strategy to balance GNN training workload. In a nutshell, (i) FlexGraph learns a cost function f to estimate the training cost for the given GNN model, instead of static metrics; (ii) with function f, FlexGraph identifies and migrates HDGs from overloaded partitions to underloaded ones to balance workload.

Cost function. FlexGraph introduces a cost function f to characterize the training cost of GNN models. Intuitively, the cost function f estimates the training cost *w.r.t.* each root vertex in the HDGs, including those costs consumed by AGGREGATE and UPDATE operations. The training cost of a partition is the sum of the costs incurred by all its owned root vertices. Following Fan *et al.* [6], FlexGraph models f as a polynomial function over a metric set, which consists of two kinds of metric variables. (i) The first set of variables $n_1, \ldots, n_k$ indicate the numbers of neighbors for different types, *e.g.,* for vertex A in MAGNN we have $n_1 = 1$ and $n_2 = 4$, since A has 1 path matching the first metapath MP_1 and 4 paths matching MP_2 (see Figure 11a). (ii) The second kind of variables $m_1, \ldots, m_k$ represent the size of each type of neighbor instance, *e.g.,* in the MAGNN case, if each vertex has a feature of dimension 20, then $m_1 = m_2 = 60$ since each path instance has 3 vertices.

Taking MAGNN as an example, we can define its cost function as $f = n_1 m_1 + n_2 m_2$, which essentially estimates the size of its neighbors for a root vertex. Suppose that the vertex set is partitioned as shown in Figure 2a. We thus put root vertices B, C, D, E and their associated HDGs in one partition and root vertices A, F, H, I, G and their associated HDGs in another. Note that $f(\text{partition \#1}) = 60$. This is because there

is only one path of size 60 that matches MP_1 in partition #1 (see vertex B in Figure 11a). Similarly, $f(\text{partition \#2}) = 600$.

That is, although the partition is well-balanced *w.r.t.* vertex size and edge size of the input graph, the GNN training workload can be very skewed and needs to be re-balanced.

Online workload balancing. FlexGraph uses an online workload balancing strategy. FlexGraph first estimates workload of each partition by its cost function f. It identifies and migrates some HDGs from overloaded partitions to underloaded ones. The migration process should comply with the data dependency of HDGs to avoid high communication overheads in GNN training. Observe that synchronization is only needed for root vertices and leaf vertices across partitions, since only these vertices may be replicated. Thus we can use an *induced graph* of HDGs to assist the migration process. The induced graph can be constructed by connecting each root vertex and its leafs in HDGs. Figure 11a depicts the HDGs of MAGNN for the sample graph from Figure 2a and its induced graph is shown in Figure 11b, where the edges represent data dependency in MAGNN training.

To bound the communication cost in distributed training, FlexGraph first generates a pre-defined number of balancing plans and chooses the one that cuts the fewest edges in the induced graph. A balancing plan is generated by identifying migration candidates in overloaded partitions, as in the heuristic ParE2H [6]. Specifically, in an overloaded partition, FlexGraph first conducts a BFS traversal starting from a seed vertex. Following the BFS order, it includes the vertices within a cost budget in a greedy manner. The budget is computed using the cost function f. The vertices excluded in the process are treated as migration candidates. Recall that we have $f(\text{partition\#1}) = 60$ and $f(\text{partition\#2}) = 600$ for MAGNN over the partitions of Figure 2a. Two possible workload balancing plans are as follows: from partition #2 to partition #1, it (i) either migrates root vertices $\{I, G\}$ and their associated HDGs, or (ii) $\{A\}$ and its associated HDG. Both plans would result in balanced workload, *i.e.,* 360 vs. 300. FlexGraph chooses the second plan. This is because the second plan does not increase the number of cut edges (see Figure 11c), while the first one increases the number of cut edges by 5 (new cut edges are colored blue in Figure 11b).

Pipeline processing. FlexGraph also inherits pipeline processing to improve distributed GNN training. HDGs characterize the dependencies among vertices, and each vertex captures its required vertices at the bottom level of HDGs, while the features of these vertices may reside in other partitions under a distributed environment. As a result, FlexGraph *partially aggregates* the features of such vertices that colocate at the same partition when possible. At the same time, FlexGraph overlaps partial aggregations and communications. Once the communication is finished, FlexGraph aggregates the partially aggregated results and synchronized messages to finish the bottom-level aggregation of HDGs.

Here we take GCN as an example to show how pipeline processing works. In distributed training, each vertex may only have features of its partial 1-hop neighbors at the partition it resides in (local partition). To aggregate its all 1-hop neighbors' features, a straightforward way is to first collect features of its 1-hop neighbors at other partitions (remote partitions) via message passing, then sum received features and features of its neighbors at local partition. To overlap partial aggregations and communications, for each vertex, FlexGraph first combines its *partial* 1-hop neighbors' features at a remote partition into a single assembled message that includes the sum, reducing the number of messages that must be transmitted and buffered. While messages are transmitted to the local partition, each vertex starts to sum its *partial* 1-hop neighbors' features at the local partition and obtains an intermediate result. After receiving messages from remote partitions, each vertex directly sums the message with the intermediate result. While pipeline processing has been widely adopted by parallel graph processing systems, as far as we know, none of dataflow-based deep learning frameworks, *e.g.,* TensorFlow, takes it into account. On average, this optimization improves the performance by 11.06% (see details in Section 7.7). Note that partial aggregation is available only when the aggregation function is commutative. In other cases (*e.g.,* neighbors' features are aggregated via an LSTM), FlexGraph benefits from the above batching communication strategy, as combining and transmitting features of multiple vertices in a single large message is more efficient than directly transmitting multiple small messages.

6 Implementation of FlexGraph

We next outline an implementation of the distributed GNN framework FlexGraph.

Architecture Overview. FlexGraph adopts a four-tier architecture shown in Figure 12. (1) Its top layer is user interfaces of NAU. Algorithm developers are allowed to provide 3 UDFs for each stage in NAU to express a GNN model. (2) At the core of FlexGraph is a GNN execution engine. It translates GNN models expressed by 3 UDFs in NAU into execution plans in NN framework and graph processing engine. (3) Underlying the execution engine are (a) a NN framework which is responsible for NN operations in GNN models, (b) a graph processing engine for graph-related operations in models, (c) an application-driven load balancer to balance the workloads across different workers, and (d) an MPI (message passing interface) controller for communications among different workers (machines). (4) The bottom layer is a storage system, which manages large graph data and vertex feature data in DFS (distributed file system). It is accessible to the NN framework, graph processing engine and load balancer.

FlexGraph is implemented on top of PyTorch and libgrape-lite [23], an open-source library for parallel graph processing. FlexGraph utilizes PyTorch as the NN execution runtime, and applies libgrape-lite to express graph-related operations.
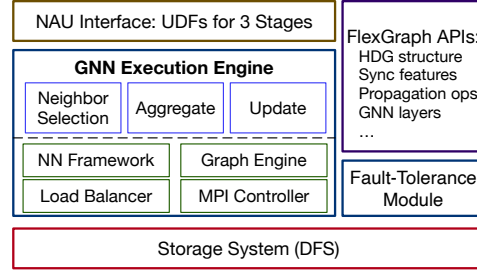

Figure 12. FlexGraph Architecture.

Integrating libgrape-lite into FlexGraph. A GNN model usually involves many operations on the graph structure. It is hard to manage (large) graph data and express graph-related operations on existing deep learning frameworks. Current graph processing systems [4, 8, 12] can naturally express many graph-related applications by adopting vertex-centric or subgraph-centric programming paradigm, and scale them to graphs with billions of vertices and edges. We develop a daemon program for FlexGraph to co-work with PyTorch by processing requests of graph operations from Python code, and putting the query results into shared memory. In the daemon, libgrape-lite is utilized for storing and processing graphs and features attached to the vertices.

Hybrid aggregate executions. FlexGraph adopts different aggregate approaches (feature fusion operations, sparse and dense tensor operations) for different levels of HDGs. Sparse/dense tensor operations can be implemented in PyTorch, while feature fusion operations are implemented in libgrape-lite, as these operations rely on graph structures. To make feature fusion operations more efficient, FlexGraph takes full advantage of powerful SIMD (single instruction, multiple data) operations (*e.g.,* Intel AVX-512 instructions) to process data of multiple dimensions. It is also equipped with memory padding for better cache efficiency. Since FlexGraph uses PyTorch to support the training process of GNN models, operations implemented in libgrape-lite have to be registered in PyTorch. FlexGraph provides a set of built-in aggregation functions, including sum, average, max and min. Along the same line, users can add other aggregation functions as UDFs (see Section 4.2).

Workload balancing. We implemented an application-driven load balancing component ADB in FlexGraph that works as follows. ADB first partitions the input graph offline, utilizing conventional graph partitioning algorithms, *e.g.,* PuLP [32] or Hash. In each training process for a specific GNN model, it samples running logs (including the values of variables defined in Section 5 for each root vertex in HDGs) online. Once the balance factor exceeds a pre-defined threshold, it learns a cost function f via polynomial regression from the logs, to characterize the GNN training cost patterns. Guided by f, ADB then generates 5 balancing plans and chooses the one that incurs the least communication cost in GNN training (Section 5).

Table 1. Datasets used in evaluation.

Dataset	#vertices	#edges	#features	#labels
Reddit	233K	11.6M	1433	41
FB91	16M	1.3B	50	10
Twitter	42 M	1.5B	50	5
IMDB	11616	34212	5257	4

7 Evaluation

We demonstrate the efficiency and scalability of FlexGraph by evaluating it on different datasets and GNN models.

Datasets. We used 4 datasets in our evaluation, including (a) Reddit [16], an online discussion forum graph; (b) FB91 [19], a synthetic dataset from LDBC benchmark; (c) Twitter [35], a social network with 42 million users; and (d) IMDB [10, 18], a dataset about movies and television programs. Table 1 summarizes their statistics.

GNN models. We used three GNN models, GCN [21], PinSage [43], MAGNN [10], one for each GNN model category. We used the default implementations in each competitors or provided by the paper authors. All these models have 2 layers. We determine the values of parameters (*e.g.,* path length in PinSage) for those GNN models by comprehensively referring to the reported details [10, 21, 43] together with official examples and guidelines of DGL and Euler. In PinSage, each vertex starts 10 random walks with length 3, and chooses top-10 visited vertices as its neighbors. In MAGNN, the input graph consists of 3 types of vertices, and we define 6 metapath types, with each metapath instance containing 3 vertices. Reddit, FB91 and Twitter are used in all three models. In addition, the implementation of MAGNN on other frameworks reports OOM (out of memory) on these three big graphs. Therefore, we introduce a small graph IMDB for MAGNN.

Baselines. We compared FlexGraph with 4 state-of-the-art baselines: (a) PyTorch v1.5.1 [30], (b) DGL v0.4.3 [38], a GNN framework for a single machine environment that adopts GAS-like programming abstraction, (c) DistDGL [49], an extension of DGL to support distributed GNN training, and (d) Euler [13], a distributed GNN framework with TensorFlow as its NN backend. We do not take NeuGraph as a baseline, as its implementation is not yet available publicly. PyG cannot efficiently support GNN training at scale, as it relies on sparse tensor operations (*i.e.,* scatter operations) to aggregate features while sparse tensor operations generate large intermediate message tensors, as explained in Section 4.2 and Wang *et al.* [38]. Therefore, PyG is not utilized in our evaluation. AliGraph [50] is another popular distributed GNN framework. AliGraph and Euler adopt the same mini-batch training strategy and similar optimization strategies (*e.g.,* an efficient graph sampling query engine that supports Gremlin, a functional language that enables users to succinctly express complex graph sampling). So we do not adopt AliGraph as the baseline system in experiments.

All experiments were evaluated on an HPC cluster with 16 machines, each with 96 cores powered by 2.5GHz, 512 GB

Table 2. Runtime in seconds for 1 epoch of 3 GNN models on a single machine; "X" indicates the system does not support the target GNN model and OOM represents out of memory.

Model	Datatset	PyT.	DGL	DistD.	Euler	FlexG.
GCN	Reddit	20.7	2.6	937.3	>3600	0.7
	FB91	400.9	40.2	>3600	OOM	26.6
	Twitter	393.1	244	>3600	OOM	162.5
PinSage	Reddit	71.6	25.1	25.8	1.3	0.6
	FB91	787.9	311.4	308.1	72.1	21.2
	Twitter	917.8	350.1	353.3	166	64.9
MAGNN	IMDB	97.5	X	X	X	0.8
	Reddit	OOM	X	X	X	7.3
	FB91	OOM	X	X	X	114.8
	Twitter	OOM	X	X	X	234

RAM, and 3.25GB/s NIC. In experiments, we focused on the end-to-end time to scan one epoch of data. All results are computed by calculating the averages over 10 epochs.

7.1 Performance on a Single Machine

We first evaluated FlexGraph by comparing it with its competitors on a single machine. Table 2 reports the end-to-end time of one epoch for three models. We find the following.

(1) Overall, FlexGraph achieves 2.4 ~ 119.3× speedups compared with PyTorch, 1.5 ~ 41.8× speedups compared with DGL, 5.4 ~ 1399× speedups compared with DistDGL, and 2.2 ~ 3.4× speedups compared with Euler on GCN and PinSage. In addition, only FlexGraph can efficiently support MAGNN on large graphs.

(2) For GCN, its implementation in PyTorch is based on sparse tensor operations (*i.e.,* sparse-dense matrix multiplication), while FlexGraph exploits efficient feature fusion operations to offload inefficient sparse operations. DGL adopts similar strategies with FlexGraph, but does not perform as well as FlexGraph. The reason is that FlexGraph applies efficient SIMD instructions to process multi-dimensional data. Note that DistDGL and Euler perform much worse on GCN than other frameworks; and Euler even reports OOM on FB91 and Twitter. This is caused by their mini-batch strategy. For GCN with 2 layers, it needs to first gather full neighbors within 2-hops for each vertex, and then converts these vertices and their relationships into a new subgraph. For dense graphs (*e.g.,* Reddit) and graphs with highly skewed power-law degree distributions (*e.g.,* FB91 and Twitter), this operation incurs tremendous computation and memory overhead.

(3) For PinSage, we found that most (over 95%) of time in PyTorch, DGL and DistDGL is used to conduct random walks. DistDGL reports almost the same performance with DGL, as they have the same implementation of conducting random walks and aggregating neighbors' features. Considering FlexGraph is more efficient in handling graph-related operations, we further directly replace random walks in DGL (DistDGL) with the corresponding implementation in FlexGraph. After that, we found that the performance of PinSage on DGL (DistDGL) is almost the same as it on FlexGraph. Euler

outperforms PyTorch and DGL (DistDGL), thanks to its efficient graph sampling engine using Gremlin as its query language. Euler also performs worse than FlexGraph, as it only uses sparse tensor operations in stage *Aggregation*, while FlexGraph applies efficient feature fusion operations.

(4) On MAGNN, FlexGraph achieves large speedups compared with PyTorch as (i) FlexGraph utilizes efficient parallel graph processing to find metapath instances and build HDGs; and (ii) the hybrid aggregation strategy is very efficient. Note that in the implementation with PyTorch on IMDB, over 95% of the total time is used to find metapath instances. Once these instances are found, they can be used during the whole training process. Having metapath instances in place, the training process of one epoch can be finished within 1 second. The hybrid aggregation strategy helps to achieve a up to 3.28× speedup for stage *Aggregation* on MAGNN (see Section 7.5). Note that PyTorch reports OOM on Reddit, FB91 and Twitter, as it explicitly generates large intermediate tensors to store features of vertices in each metapath instance. Instead, FlexGraph applies feature fusion operations to avoid this.

Summary of performance gain. FlexGraph has the following features: (a) replacing sparse tensor operations with efficient graph operations (feature fusion operations) in flat aggregation; (b) efficient *NeighborSelection* stage; (c) HDG-based hybrid execution. We next analyze the performance gain of FlexGraph on three types of models, respectively.

(1) DNFA models mainly benefit from (a). Instead of sparse tensor operations, FlexGraph applies feature fusion operations to eliminate the overhead of materializing vertex features on massive edges at the bottom level of HDGs, as well as SIMD instructions to accelerate aggregation operations.

(2) INFA models benefit from both (a) and (b). Like DNFA, aggregation of INFA models is conducted via efficient feature fusion operations. The improvement is also from efficient neighbor selection accomplished via graph processing.

(3) INHA models benefit from (a), (b) and (c). FlexGraph outperforms because of the feature fusion operations in aggregating features at the bottom level of HDGs, the efficient extractions of neighbor instances, and moreover, the HDG-based hybrid execution of hierarchical aggregation.

7.2 Simulating INFA and INHA in Existing Systems

To get more insights of performance improvement in Flex-Graph, we next compared FlexGraph with another baseline Pre+DGL. Pre+DGL "simulates" FlexGraph by combing existing GAS-like frameworks with a pre-computation process. Specifically, it first pre-computes an expanded graph to materialize the HDGs; and then applies GAS-like operations on the expanded graph. We used DGL as the GAS-like GNN framework. Since DNFA models do not need to build HDGs, we only evaluated the training of INFA and INHA models.

Table 3. Runtime in seconds of PinSage and MAGNN; "X" indicates that the target GNN model is not supported.

Model	Datatset	DGL	Pre+DGL	FlexGraph
PinSage	Reddit	25.1	9.98	0.6
	FB91	311.4	112.9	21.2
	Twitter	350.1	287.4	64.9
MAGNN	Reddit	X	5.8	4.2
	FB91	X	78.9	59.3
	Twitter	X	202.4	132.3

For PinSage, HDGs in distinct epochs are different, since "neighbor" selection process (*i.e.,* conducting random walks) is involved at each epoch. Therefore, the expanded graph materializing the HDGs cannot trivially be pre-computed but only approximated. To simulate PinSage, we pre-compute lots of random walks, associate an "importance" weight with each pair of vertices, and do weighted sampling at runtime. The results would be qualitatively the same (for enough random walks performed offline in advance).

For MAGNN, since the HDGs do not change across epochs, we pre-compute and materialize HDGs as an expanded graph in advance, and directly conduct GAS-like operations on the expanded graph. Note that MAGNN involves multi-step aggregation operations. Therefore, multiple GAS-like operations are conducted in each layer of MAGNN.

The comparison of Pre+DGL with FlexGraph is displayed in Table 3, where the reported time only considers the computations on the expanded graph, and does not contain the cost of pre-computing and storing the expanded graph. Note that in MAGNN, the HDGs remain unchanged at different epochs and NAU does not require to execute the *NeighborSelection* stage in every GNN layer. Therefore, here we only report the total time of MAGNN's *Aggregation* and *Update* stages in FlexGraph. We can observe that for PinSage, Pre+DGL performs much better than DGL, since the complexity of weighted sampling on the expanded graph is much lower than conducting random walks on the original graph. Nevertheless, FlexGraph still outperforms Pre+DGL, as Pre+DGL needs to build HDGs by conducting weighted sampling on the (perhaps larger) expanded graph. For MAGNN, Flex-Graph achieves better performance than Pre+DGL. The performance gain is partly from our hybrid aggregation strategy which is more efficient than kernel fusion operations in DGL. In addition, FlexGraph benefits from efficient SIMD instructions to process multi-dimensional data, as explained in GCN.

We should remark that not all INFA and INHA models can be "simulated" by Pre+DGL. For example, if a GNN model is performed on a dynamic graph (*i.e.,* the graph structure evolves over time), the expanded graph cannot be pre-computed in advance. Instead, the flexible interfaces of NAU allow users to easily handle such situation.

7.3 Performance on Multiple Machines

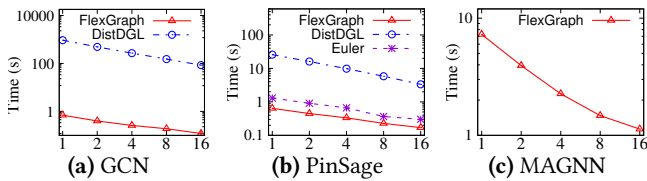We compared FlexGraph with DistDGL and Euler in a distributed environment on the Reddit dataset. Note that we

Figure 13. End-to-end performance on multiple machines.

Table 4. Breakdown of 3 stages on Twitter (in seconds).

Model	Nbr.Selection	Aggregation	Update
GCN	0 (0%)	159.3 (98%)	3.2 (2%)
PinSage	27.4 (42.2%)	34.4 (53%)	3.1 (4.8%)
MAGNN	101.8 (43.5%)	130 (55.5%)	2.3 (1%)

only report the result of MAGNN in FlexGraph, as it cannot be implemented in DistDGL and Euler.

(1) As shown in Figure 13, on all 3 GNN modes, FlexGraph nearly achieves linear speedup as the increase of machines.

(2) On GCN, FlexGraph outperforms DistDGL by 1021.76× on average. We do not report the performance of GCN on Euler, since it cannot finish the training process of one epoch in half an hour. The reason is the mini-batch strategy they adopt, as we explained in Section 7.1.

(3) On PinSage, FlexGraph outperforms Euler and DistDGL by up to 2.04× and 40.34×, respectively. The speedup over Euler is mainly because of the hybrid aggregate execution scheme and pipeline processing. In contrast, Euler adopts a dataflow-based framework, and starts the AGGREGATE operation after all required features are synchronized from other machines, using sparse tensor operations. Euler performs much better than DistDGL, and we find that conducting random walks in Euler is more efficient than DistDGL.

7.4 Breakdown Analysis

Table 4 displays the stage-level time breakdown of the selected models based on the Twitter dataset in a single machine environment, from which we can observe that the stage time distribution varies greatly from model to model. Next we present a detailed stage-by-stage analysis as follows.

(1) Stage *NeighborSelection* varies according to the complexity of UDFs to select "neighbors" (see Figure 5). For DNFA models (*e.g.*, GCN), the neighborhood is defined as 1-hop neighbors. Therefore, the input graph structure can capture dependencies among vertices, and we do not need to build HDGs explicitly. In contrast, this stage in INFA and INHA models usually involves many graph-related operations (*e.g.*, conducting random walks and finding metapath instances). Over 40% of end-to-end time is used in this stage, even we have integrated an efficient parallel graph processing library into FlexGraph. This result clearly demonstrates the necessity of efficient graph processing for a GNN framework.

(2) Stage *Aggregation* is executed based on HDGs, and its execution time is affected by both the size of HDGs and the complexity of the aggregation function for each level of
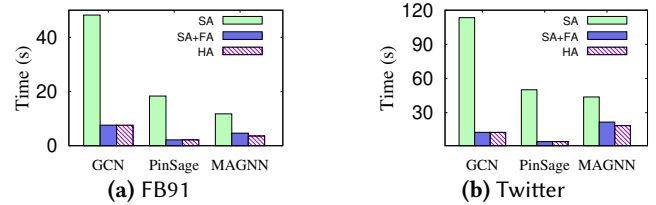

Figure 14. Effectiveness of hybrid aggregation.

HDGs. GCN and PinSage have the same aggregation function. Compared with GCN, each vertex in PinSage has a relatively small size of neighbors. Therefore, the time spent in this stage for PinSage is much shorter than GCN. MAGNN also spends a quite long time to execute the aggregation operation, although it has smaller sizes of neighbors than GCN, as its aggregation function is more complex than GCN.

(3) Stage *Update* takes a relatively small percentage of end-to-end time. This stage usually only involves dense NN operations on each vertex, which can be efficiently executed in current deep learning frameworks.

7.5 Hybrid Aggregation

We next evaluated the effectiveness of the hybrid aggregation strategy of FlexGraph on datasets FB91 and Twitter. Denote by SA the strategy that uses only sparse scatter operations in aggregation (*e.g.*, Figure 8), by SA+FA the strategy that exploits both sparse tensor and feature fusion operations for aggregation, and by HA the hybrid aggregation strategy (*i.e.*, (SA+FA)+dense tensor operations). Figure 14 shows the performance of the *Aggregation* stage in FlexGraph that adopts these aggregation strategies with partition number $k = 8$.

(1) Overall, HA achieves on average a 6.73× speedup (up to 10.99×) compared with SA in the *Aggregation* stage.

(2) Dense tensor operations cannot be applied in GCN and PinSage, as their schema trees keep a flat structure and do not involve aggregation operations, while dense tensor operations only are only used for aggregations in the schema trees. Therefore, HA and SA+FA report the same performance in these two models. On both GNN models, feature fusion operations improve the aggregation performance by on average 7.63× and 9.73×, respectively. This verifies the effectiveness of embracing parallel graph processing into NN workloads.

(3) On MAGNN, HA beats SA by 2.81× on average. We observe that in HA, dense tensor operation contributes 12.61% speedup over SA on average. For example, by disabling dense tensor operation of HA, a round of aggregation of MAGNN increases from 18.63s to 21.74s over dataset Twitter.

7.6 Workload Balancing

We next evaluated the impact of application-driven workload balancing strategy ADB of FlexGraph. We compared it with the classical Hash partitioning and a state-of-the-art graph partitioner PULP [32]. Figure 15a shows the performance of stage *Aggregation* of GCN, PinSage and MAGNN over Twitter with partition number $k = 8$. Our strategy ADB beats Hash
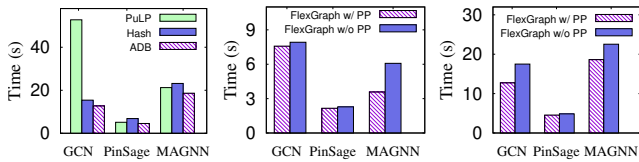
(a) WB. over Twitter **(b)** PP. over FB91 **(c)** PP. over Twitter

Figure 15. Effectiveness of workload balancing (WB.) and pipeline processing (PP.)

and PULP by 23.41% and 33.03% on average, respectively. The improvement over PULP is larger, since the partitions generated by PULP are more skewed than those of Hash.

7.7 Pipeline Processing

We also evaluated the effectiveness of pipeline processing. Figures 15b and 15c show the performance of stage *Aggregation* of FlexGraph for one epoch for all 3 GNN models over FB91 and Twitter, with/without pipeline processing. The partition number k is 8. On average, with pipeline processing, the performance of FlexGraph improves by 15.75%, 5.72%, 29.23% on GCN, PinSage, and MAGNN, up to 27.2% and 6.17%, 41.11% respectively. This verifies the effectiveness of pipeline processing in a distributed setting. PinSage achieves the smallest improvement, as each vertex in PinSage has a relatively small size of neighbors, thus communication sizes among different machines are also relatively small.

7.8 Memory Consumption of HDGs

We last evaluated the memory consumption of HDGs, comparing to the size of the input graph G (see Table 5). FlexGraph does not construct extra HDGs for GCN, since the input graph serves the desired purpose well. Compared with PinSage, memory consumption of HDGs for MAGNN is relatively large, since the size of neighbor for each vertex is relatively large, and each neighbor contains multiple vertices from G. With the optimized storage strategy, HDGs of MAGNN take space at most 1.28× of the input graph. Once the HDGs for INFA and INHA models are constructed, the input graph G can be swapped to disk to save memory if necessary, since the aggregation and update operations are executed directly upon HDGs, not the input graph.

8 Related Work

It is challenging to train GNN models on large-scale graphs. In traditional deep learning frameworks, training samples are mutually independent, and developers usually adopt a mini-batch strategy to train large-scale data. However, graphs inherently represent the dependencies among training samples (*i.e.,* vertices), and each vertex in GNN models must incorporate its depending samples (*i.e.,* neighbors). Existing solutions to support GNN training at scale can be roughly divided into two categories. The first is to deal with all vertices/edges of the entire large graph simultaneously on multiple devices/-machines, and typical solutions include NeuGraph [26] and ROC [20]. NeuGraph first splits a large graph into multiple

Table 5. Mem. footprint of HDGs *w.r.t.* input graphs.

	Reddit	FB91	Twitter
PinSage	2.03%	11.99%	28.37%
MAGNN	27.74%	107.27%	128.42%

chunks, using a 2-D graph partitioning; it then processes one chunk each time where a GAS-like abstraction (*i.e.,* SAGA-NN) is applied on each chunk and the intermediate result of each chunk is stored; and finally it combines all intermediate results after all chunks are processed. The second category is to adopt various neighbor sampling strategies to obtain full or partial neighbors within k hops of each vertex for a GNN model with k layers. Then the training process takes a batch of vertices as well as their neighbors within k hops in a mini-batch. Note that in each GNN layer, each vertex still only aggregates features of its sampled 1-hop neighbors. AliGraph [50] and Euler [13] fall into this category. To allow users to succinctly express various sampling strategies, AliGraph adopts Gremlin as the high-level query language for its sampling engine, which translates every sampling query into a distributed execution plan that runs across multiple machines. Note that both aforementioned distributed training strategies are designed for DNFA models only.

9 Conclusion and Future Work

GNN models vary in both the definitions of "neighbors" and neighborhood aggregation schemes, which are beyond the reach of existing GNN frameworks. To alleviate the expressivity challenge, we propose a new GNN programming model NAU, which utilizes hierarchical dependency graphs HDGs to express hierarchical dependencies among vertices. Based on NAU, we present a distributed GNN framework FlexGraph, which adopts several optimization strategies from different aspects to improve its performance.

Currently, GPUs have been widely used in deep learning tasks due to their much higher massive parallelism and memory access bandwidth than CPUs, and there also exist some solutions focusing on exploiting GPUs for GNN performance acceleration [20, 25, 26]. These solutions are orthogonal approaches to support efficient GNN training. We believe that the NAU programming abstraction and proposed optimization strategies can be applied into GPUs, and take the support of GPUs as one potential future direction.

Acknowledgement

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[2] Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. 2020. Spectral clustering with graph neural networks for graph pooling. In *Proceedings of the 37th international conference on Machine learning. ACM*.

[3] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 233–244.

[4] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*.

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Neural Information Processing Systems, Workshop on Machine Learning Systems*.

[6] Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Xiaojian Luo, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2020. Application Driven Graph Partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. International Foundation for Autonomous Agents and Multiagent Systems, Portland OR USA, 1765–1779. https://doi.org/10/gg6rb2

[7] Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Chao Tian, and Jingren Zhou. 2020. Capturing associations in graphs. *Proceedings of the VLDB Endowment* 13, 12 (2020), 1863–1876.

[8] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing sequential graph computations. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 1–39.

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

[10] Xinyu Fu, Jiani Zhang, Ziqiao Meng, and Irwin King. 2020. MAGNN: Metapath Aggregated Graph Neural Network for Heterogeneous Graph Embedding. In *Proceedings of The Web Conference 2020*. 2331–2341.

[11] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 1263–1272.

[12] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 17–30.

[13] Euler graph library. 2020. https://github.com/alibaba/euler

[14] Fangda Guo, Ye Yuan, Guoren Wang, Lei Chen, Xiang Lian, and Zimeng Wang. 2019. Cohesive group nearest neighbor queries over road-social networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 434–445.

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*. 1024–1034.

[16] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687* (2020).

[17] Ziniu Hu, Yuxiao Dong, Kuansan Wang, Kai-Wei Chang, and Yizhou Sun. 2020. GPT-GNN: Generative Pre-Training of Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1857–1867.

[18] IMDB. 2020. http://www.imdb.com/

[19] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1317–1328. https://doi.org/10/gf9tg4

[20] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems (MLSys)* (2020), 187–198.

[21] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR)*.

[22] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. 2018. Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting. In *International Conference on Learning Representations*.

[23] libgrape lite. 2020. https://github.com/alibaba/libgrape-lite

[24] PGL library. 2020. https://github.com/PaddlePaddle/PGL

[25] Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. 2020. G3: when graph neural networks meet parallel graph processing systems on GPUs. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2813–2816.

[26] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: parallel deep neural network computation on large graphs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*. 443–457.

[27] Diego Marcheggiani and Ivan Titov. 2017. Encoding Sentences with Graph Convolutional Networks for Semantic Role Labeling. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 1506–1515.

[28] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford InfoLab.

[29] Shichao Pei, Lu Yu, Robert Hoehndorf, and Xiangliang Zhang. 2019. Semi-supervised entity alignment via knowledge graph embedding with awareness of degree difference. In *The World Wide Web Conference*. 3130–3136.

[30] PyTorch. 2020. http://pytorch.org

[31] PyTorch Scatter. 2020. https://github.com/rusty1s/pytorch_scatter

[32] George M. Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. 2016. Partitioning Trillio-Edge Graphs in Minutes. *IPDPS* (2016).

[33] Jianing Sun, Wei Guo, Dengcheng Zhang, Yingxue Zhang, Florence Regol, Yaochen Hu, Huifeng Guo, Ruiming Tang, Han Yuan, Xiuqiang He, et al. 2020. A framework for recommending accurate and diverse items using bayesian graph convolutional neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2030–2039.

[34] Jianing Sun, Yingxue Zhang, Wei Guo, Huifeng Guo, Ruiming Tang, Xiuqiang He, Chen Ma, and Mark Coates. 2020. Neighbor Interaction Aware Graph Convolution Networks for Recommendation. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1289–1298.

[35] Twitter. 2010. http://twitter.com/

[36] Lei Wang, Jianwei Niu, Xuefeng Liu, and Kaili Mao. 2019. The Silent Majority Speaks: Inferring Silent Users' Opinions in Online Social Networks. In *The World Wide Web Conference*. 3321–3327.

[37] Lei Wang, Jianwei Niu, and Shui Yu. 2019. SentiDiff: Combining textual information and sentiment diffusion patterns for Twitter sentiment analysis. *IEEE Transactions on Knowledge and Data Engineering* 32, 10 (2019), 2026–2039.

[38] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019). https://arxiv.org/abs/1909. 01315

[39] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan. 2019. Session-based recommendation with graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 346–353.

[40] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[41] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.

[42] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation Learning on Graphs with Jumping Knowledge Networks. In *International Conference on Machine Learning*. 5453–5462.

[43] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th*

[44] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. Gnnexplainer: Generating explanations for graph neural networks. In *Advances in neural information processing systems*. 9244–9255.

[45] Jiaxuan You, Rex Ying, and Jure Leskovec. 2019. Position-aware Graph Neural Networks. In *International Conference on Machine Learning*. 7134–7143.

[46] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*. 5165–5175.

[47] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2020. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[48] Zhihui Zhang, Jingwen Leng, Lingxiao Ma, Youshan Miao, Chao Li, and Minyi Guo. 2020. Architectural Implications of Graph Neural Networks. *IEEE Computer Architecture Letters* 19, 1 (2020), 59–62.

[49] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. *arXiv preprint arXiv:2010.05337* (2020).

[50] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.