Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis

Maciej Besta and Torsten Hoefler
Department of Computer Science, ETH Zurich

arXiv:2205.09702v2 [cs.LG] 20 May 2022

Abstract—Graph neural networks (GNNs) are among the most powerful tools in deep learning. They routinely solve complex problems on unstructured networks, such as node classification, graph classification, or link prediction, with high accuracy. However, both inference and training of GNNs are complex, and they uniquely combine the features of irregular graph processing with dense and regular computations. This complexity makes it very challenging to execute GNNs efficiently on modern massively parallel architectures. To alleviate this, we first design a taxonomy of parallelism in GNNs, considering data and model parallelism, and different forms of pipelining. Then, we use this taxonomy to investigate the amount of parallelism in numerous GNN models, GNN-driven machine learning tasks, software frameworks, or hardware accelerators. We use the work-depth model, and we also assess communication volume and synchronization. We specifically focus on the sparsity/density of the associated tensors, in order to understand how to effectively apply techniques such as vectorization. We also formally analyze GNN pipelining, and we generalize the established Message-Passing class of GNN models to cover arbitrary pipeline depths, facilitating future optimizations. Finally, we investigate different forms of asynchronicity, navigating the path for future asynchronous parallel GNN pipelines. The outcomes of our analysis are synthesized in a set of insights that help to maximize GNN performance, and a comprehensive list of challenges and opportunities for further research into efficient GNN computations. Our work will help to advance the design of future GNNs.

Index Terms—Parallel Graph Neural Networks, Distributed Graph Neural Networks, Parallel Graph Convolution Networks, Distributed Graph Convolution Networks, Parallel Graph Attention Networks, Distributed Graph Attention Networks, Parallel Message Passing Neural Networks, Distributed Message Passing Neural Networks, Asynchronous Graph Neural Networks.

1 Introduction

Graph neural networks (GNNs) are taking over the world of machine learning (ML) by storm [58], [223]. They have been used in a plethora of complex problems such as node classification, graph classification, or edge prediction [77], [110]. Example areas of application are social sciences (e.g., studying human interactions), bioinformatics (e.g., analyzing protein structures), chemistry (e.g., designing compounds), medicine (e.g., drug discovery), cybersecurity (e.g., identifying intruder machines), entertainment services (e.g., predicting movie popularity), linguistics (e.g., modeling relationships between words), transportation (e.g., finding efficient routes), and others [23], [49], [57], [58], [66], [89], [100], [108], [118], [223], [244], [251]. Some recent celebrated success stories are cost-effective and fast placement of high-performance chips [157], simulating complex physics [170], [177], guiding mathematical discoveries [70], or significantly improving the accuracy of protein folding prediction [120].

GNNs uniquely generalize both *traditional deep learning* [15], [95], [138] and *graph processing* [96], [152], [176]. Still, contrarily to the former, they do not operate on regular grids and highly structured data (such as, e.g., image processing); instead, the data in question is highly unstructured, irregular, and the resulting computations are data-driven and lacking straightforward spatial or temporal locality [152]. Moreover, contrarily to the latter, vertices and/or edges are associated with complex data and processing. For example, in many GNN models, each vertex i has an assigned k-dimensional *feature vector*, and each such vector is combined with the vectors of i's neighbors; this process is repeated iteratively. Thus, while the overall style of such GNN computations resembles label propagation algorithms such as PageRank [33], [167], it comes with additional complexity due to the high dimensionality of the vertex features.

Yet, this is only how *the simplest* GNN models, such as basic Graph Convolution Networks (GCN) [128], work. In many, if not most, GNN models, high-dimensional data may also be attached to every edge, and complex updates to the edge data take place at every iteration. For example, in the Graph Attention Network (GAT) model [202], to compute the scalar weight of a single edge (i, j), one must first concatenate linear transformations of the feature vectors of both vertices i and j, and then construct a dot product of such a resulting vector with a trained parameter vector. Other models come with even more complexity. For example, in Gated Graph ConvNet (G-GCN) [47] model, the edge weight may be a multidimensional vector.

At the same time, *parallel* and *distributed* processing have essentially become synonyms for computational efficiency. Virtually each modern computing architecture is parallel: cores form a socket while sockets form a non-uniform memory access (NUMA) compute node. Nodes may be further clustered into blades, chassis, and racks [25], [87], [180]. Numerous memory banks enable data *distribution*. All these parts of the architectural hierarchy run in parallel. Even a single sequential core offers parallelism in the form of *vectorization*, *pipelining*, or *instruction-level parallelism (ILP)*. On top of that, such architectures are often *heterogeneous*: Processing units can be CPUs or GPUs, Field Programmable Gate Arrays (FPGAs), or others. *How to harness all these rich features to achieve more performance in GNN workloads?*

To help answer this question, we systematically analyze different aspects of GNNs, focusing on *the amount of parallelism and distribution* in these aspects. We use fundamental theoretical parallel computing machinery, for example the

Structure of graph inputs	
$G = (V, E)$	A graph; V and E are sets of vertices and edges.
n, m	Numbers of vertices and edges in G; $\|V\| = n, \|E\| = m$.
$N(i), N^+(i), \widehat{N}(i)$	Neighbors of i, in-neighbors of i, and $\widehat{N}(i) = N(i) \cup \{i\}$.
d_i, d	The degree of a vertex i and the maximum degree in a graph.
$\mathbf{A}, \mathbf{D} \in \mathbb{R}^{n \times n}$	The graph adjacency and the degree matrices.
$\widetilde{\mathbf{A}}, \widetilde{\mathbf{D}}$	$\mathbf{A}$ and $\mathbf{D}$ matrices with self-loops ($\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$, $\widetilde{\mathbf{D}} = \mathbf{D} + \mathbf{I}$).
$\widehat{\mathbf{A}}, \overline{\mathbf{A}}$	Normalization: $\widehat{\mathbf{A}} = \widetilde{\mathbf{D}}^{-\frac{1}{2}} \widetilde{\mathbf{A}} \widetilde{\mathbf{D}}^{-\frac{1}{2}}$ and $\overline{\mathbf{A}} = \mathbf{D}^{-1}\mathbf{A}$ [223].
Structure of GNN computations	
L, k	The number of GNN layers and input features.
$\mathbf{X} \in \mathbb{R}^{n \times k}$	Input (vertex) feature matrix.
$\mathbf{Y}, \mathbf{H}^{(l)} \in \mathbb{R}^{n \times O(k)}$	Output (vertex) feature matrix, hidden (vertex) feature matrix.
$\mathbf{x}_i, \mathbf{y}_i, \mathbf{h}_i^{(l)} \in \mathbb{R}^n$	Input, output, and hidden feature vector of a vertex i (layer l).
$\mathbf{W}^{(l)} \in \mathbb{R}^{O(k) \times O(k)}$	A parameter matrix in layer l.
$\sigma(\cdot)$	Element-wise activation and/or normalization.
$\times, \odot$	Matrix multiplication and element-wise multiplication.

TABLE 1: The most important symbols used in the paper.

Work-Depth model [42], to reveal architecture independent insights. We put special focus on the linear algebra formulation of computations in GNNs, and we investigate the sparsity and density of the associated tensors. This offers further insights into performance-critical features of GNN computations, and facilitates applying parallelization mechanisms such as vectorization. *In general, our investigation will help to develop more efficient GNN computations.*

For a systematic analysis, *we propose an in-depth taxonomy of parallelism in GNNs*. The taxonomy identifies fundamental forms of parallelism in GNNs. While some of them have direct equivalents in traditional deep learning, we also illustrate others that are specific to GNNs.

To ensure wide applicability of our analysis, we cover a large number of different aspects of the GNN landscape. Among others, we consider different categories of GNN models (e.g., spatial, spectral, convolution, attentional, message passing), a large selection of GNN models (e.g., GCN [128], SGC [219], GAT [202], G-GCN [47]), parts of GNN computations (e.g., inference, training), building blocks of GNNs (e.g., layers, operators/kernels), programming paradigms (e.g., SAGA-NN [153], GReTA [126]), execution schemes behind GNNs (e.g., reduce, activate, different tensor operations), GNN frameworks (e.g., NeuGraph [153]), GNN accelerators (e.g., HyGCN [228]) GNN-driven ML tasks (e.g., node classification, edge prediction), mini-batching vs. full-batch training, different forms of sampling, and asynchronous GNN pipelines.

We finalize our work with *general insights* into parallel and distributed GNNs, and a set of *research challenges and opportunities*. Thus, our work can serve as a guide when developing parallel and distributed solutions for GNNs executing on modern architectures, and for choosing the next research direction in the GNN landscape.

1.1 Complementary Analyses

We discuss related works on the *theory* and *applications* of GNNs. There exist general GNN surveys [48], [49], [58], [100], [223], [241], [244], [251], works on theoretical aspects (spatial–spectral dichotomy [11], [63], the expressive power of GNNs [178], or heterogeneous graphs [224], [229]), analyzes of GNNs for specific applications (knowledge graph completion [5], traffic forecasting [119], [195], symbolic computing [137], recommender systems [220], text classification [115], or action recognition [3]), explainability of GNNs [232], and on software (SW) and hardware (HW) accelerators and SW/HW co-design [1]. We complement these works as we focus on parallelism and distribution of GNN workloads.

1.2 Scope of this Work & Related Domains

We focus on GNNs, but we also cover parts of the associated domains. In the graph **embeddings** area, one develops methods for finding low-dimensional representations of elements of graphs, most often vertices [68], [69], [210], [211]. As such, GNNs can be seen as a part of this area, because one can use a GNN to construct an embedding [223]. However, we exclude non-GNN related methods for constructing embeddings, such as schemes based on random walks [97], [168] or graph kernel designs [46], [131], [203].

2 Graph Neural Networks: Overview

We overview GNNs. Table 1 explains the most important notation. We first summarize a GNN computation and GNN-driven downstream ML tasks (§ 2.1). We then discuss different parts of a GNN computation in more detail, providing both the basic knowledge and general opportunities for parallelism and distribution. This includes the input GNN datasets (§ 2.2), the mathematical theory and formulations for GNN models that form the core of GNN computations (§ 2.3), GNN inference vs. GNN training (§ 2.4), and the programmability aspects (§ 2.5). We finish with a taxonomy of parallelism in GNNs (§ 2.6) and parallel & distributed theory used for formal analyses (§ 2.7).

2.1 GNN Computation: A High-Level Summary

We overview a **GNN computation** in Figure 1. The input is a *graph dataset*, which can be a single graph (usually a large one, e.g., a brain network), or several graphs (usually many small ones, e.g., chemical molecules). The input usually comes with *input feature vectors* that encode the semantics of a given task. For example, if the input nodes and edges model – respectively – papers and citations between these papers, then each node could come with an input feature vertex being a one-hot bag-of-words encoding, specifying

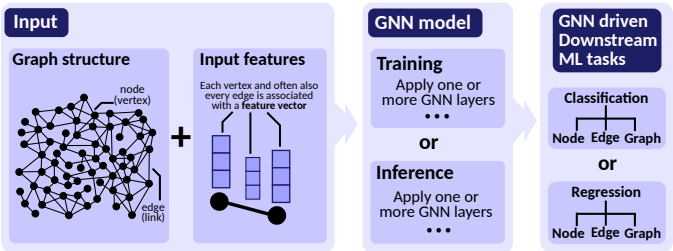

Fig. 1: **(§ 2.1) Overview of general GNN computation.** Input comprises the graph structure and the accompanying feature vectors (assigned to vertices/edges). The input is processed using a specific GNN model (training or inference). Output feature vectors are used in various downstream ML tasks.

the presence of words in the abstract of a given publication. Then, a *GNN model* – underlying the training and inference process – uses the graph structure and the input feature vectors to generate the *output feature vectors*. In this process, intermediate *hidden latent vectors* are often created. Note that hidden features may be updated *iteratively* more than once (we refer to a single such iteration, that updates all the hidden features, as a *GNN layer*). The output feature vectors are then used for the *downstream ML tasks* such as node classification or graph classification.

A single **GNN layer** is summarized in Figure 2. In general, one first applies a certain *graph-related* operation to the features. For example, in the GCN model [128], one aggregates the features of neighbors of each vertex v into the feature vector of v using summation. Then, a selected operation related to *traditional neural networks* is applied to the feature vectors. A common choice is an MLP or a plain linear projection. Finally, one often uses some form of non-linear activation (e.g., ReLU [128]) and/or normalization.

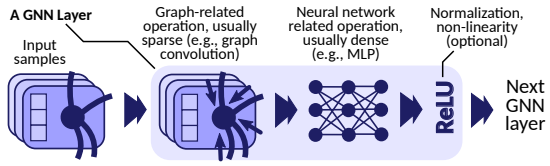

Fig. 2: **(§ 2.1) Overview of a single GNN layer.** The input set of samples (e.g., vertices or graphs) is processed with a graph-related operation such as graph convolution, followed by a neural network related operation such as an MLP, and finally (optionally) by a non-linearity such as ReLU, possibly combined with some normalization.

One key difference between GNNs and traditional deep learning are *possible dependencies between input data samples* which make the parallelization of GNNs much more challenging. We show **GNN data samples** in Figure 3. A single sample can be a node (a vertex), an edge (a link), a subgraph, or a graph itself. One may aim to classify samples (assign labels from a discrete set) or conduct regression (assign continuous values to samples). Both vertices and edges have inter-dependencies: vertices are connected with edges while edges share common vertices. The seminal work by Kipf and Welling [128] focuses on node classification. Here, one is given a single graph as input, data samples are single vertices, and the goal is to classify all unlabeled vertices.

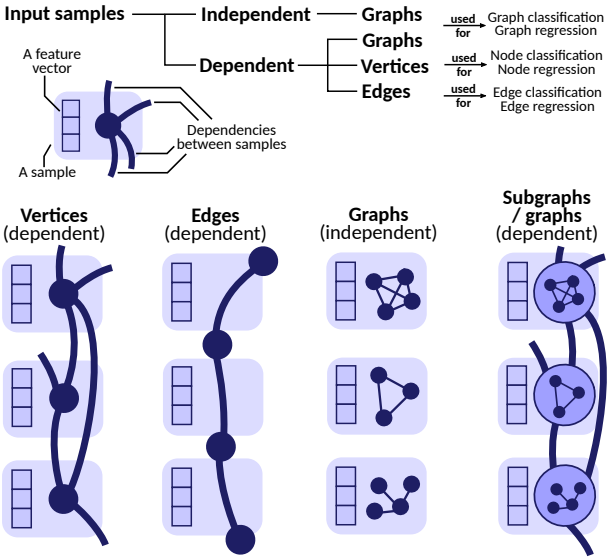

Fig. 3: **(§ 2.1) Overview of GNN samples.** GNN downstream ML tasks aim at classification or regression of vertices, edges, or graphs. While both vertex and edge samples virtually always have inter-sample dependencies, graphs may be both dependent and independent.

Graphs – when used as basic data samples – are usually independent [225], [231] (cf. Figure 3, 3rd column). An example use case is classifying chemical molecules. This setting resembles traditional deep learning (e.g., image recognition), where samples (single pictures) also have no explicit dependencies. Note that, as chemical molecules may differ in sizes, load balancing issues may arise. This also has analogies in traditional deep learning, e.g., sampled videos also may have varying sizes [143]. However, graph classification may also feature graph samples *with* inter-dependencies (cf. Figure 3, 4th column). This is useful when studying, for example, relations between network communities [142].

2.2 Input Datasets & Output Structures in GNNs

A GNN computation starts with the input graph G, modeled as a tuple (V, E); V is a set of vertices and $E \subseteq V \times V$ is a set of edges; $|V| = n$ and $|E| = m$. $N(v)$ denotes the set of vertices adjacent to vertex (node) v, d_v is v's degree, and d is the maximum degree in G (all symbols are listed in Table 1). The adjacency matrix (AM) of a graph is $\mathbf{A} \in \{0,1\}^{n \times n}$. $\mathbf{A}$ determines the connectivity of vertices: $\mathbf{A}(i,j) = 1 \Leftrightarrow (i,j) \in E$. The input, output, and hidden feature vector of a vertex i are denoted with, respectively, $\mathbf{x}_i, \mathbf{y}_i, \mathbf{h}_i$. We have $\mathbf{x}_i \in \mathbb{R}^k$ and $\mathbf{y}_i, \mathbf{h}_i \in \mathbb{R}^{O(k)}$, where k is the number of input features. These vectors can be grouped in matrices, denoted respectively as $\mathbf{X}, \mathbf{Y}, \mathbf{H} \in \mathbb{R}^{n \times k}$. If needed, we use the iteration index (l) to denote the latent features in an iteration (GNN layer) l ($\mathbf{h}_i^{(l)}$, $\mathbf{H}^{(l)}$). Sometimes, for clarity of equations, we omit the index (l).

2.3 GNN Mathematical Models

A GNN model defines a mathematical transformation that takes as input (1) the graph structure $\mathbf{A}$ and (2) the input features $\mathbf{X}$, and generates the output feature matrix $\mathbf{Y}$. Unless specified otherwise, $\mathbf{X}$ models vertex features. The

exact way of constructing $\mathbf{Y}$ based on $\mathbf{A}$ and $\mathbf{X}$ is an area of intense research. Here, hundreds of different GNN models have been developed [49], [58], [100], [223], [241], [244], [251]. We now discuss different categories of GNN models, see Figure 4 for a summary. *Importantly for parallel and distributed execution, one can formulate most GNN models using either the* ***local formulation (LC)*** *based on functions operating on single edges or vertices, or the* ***global formulation (GL)****, based on operations on matrices grouping all vertex- and edge-related vectors.*

2.3.1 Local (LC) GNN Formulations

In many GNN models, the latent feature vector $\mathbf{h}_i$ of a given node i is obtained by applying a *permutation invariant* aggregator function $\bigoplus$, such as sum or max, over the feature vectors of the neighbors $N(i)$ of i. Moreover, the feature vector of each neighbor of i may additionally be transformed by a function ψ. Finally, the outcome of $\bigoplus$ may be also transformed with another function ϕ. The sequence of these three transformations forms one *GNN layer*. We denote such a GNN model formulation (based on $\bigoplus, \psi, \phi$) as **local (LC)**. Formally, the equation specifying the feature vector $\mathbf{h}_i^{(l+1)}$ of a vertex i in the next GNN layer $l+1$ is as follows:

$$\mathbf{h}_i^{(l+1)} = \phi\left(\mathbf{h}_i^{(l)}, \bigoplus_{j \in N(i)} \psi\left(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}\right)\right) \tag{1}$$

Depending on the details of ψ, one can further distinguish three GNN classes [48]: *Convolutional GNNs* (C-GNNs), *Attentional GNNs* (A-GNNs), and *Message-Passing GNNs* (MP-GNNs). In short, in these three classes of models, ψ is – respectively – a fixed scalar coefficient (C-GNNs), a learnable function that returns a scalar coefficient (A-GNNs), or a learnable function that returns a vector coefficient (MP-GNNs).

As an example, consider the seminal GCN model by Kipf and Welling [128]. Here, $\bigoplus$ is a sum over $N(i) \cup \{i\} \equiv \widehat{N}(i)$, ψ acts on each neighbor j's feature vector by multiplying it with a scalar $1/\sqrt{d_i d_j}$, and ϕ is a linear projection with a trainable parameter matrix $\mathbf{W}$ followed by $ReLU$. Thus, the LC formulation is given by $\mathbf{h}_i^{(l+1)} = ReLU\left(\mathbf{W}^{(l)} \times \left(\sum_{j \in \widehat{N}(i)} \frac{1}{\sqrt{d_i d_j}} \mathbf{h}_j^{(l)}\right)\right)$. Note that each iteration may have different projection matrices $\mathbf{W}^{(l)}$.

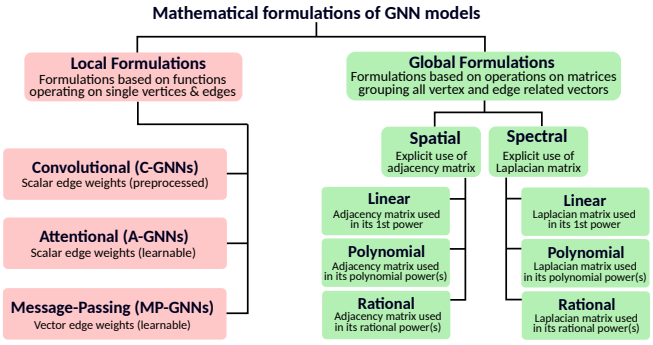

Fig. 4: **(§ 2.3) Categories of GNN models.** We propose a classification into **local** and **global** formulations of GNN models. Red/green refer to formulation details in Figure 5.

There are many ways in which one can parallelize GNNs in the LC formulation. Here, the first-class citizens are "fine-grained" functions being evaluated for vertices and edges. Thus, one could execute these functions in parallel over different vertices, edges, and graphs, parallelize a single function over the feature dimension or over the graph structure, pipeline a sequence of functions within a GNN layer or across GNN layers, or fuse parallel execution of functions. We discuss all these aspects in the following sections.

2.3.2 Global (GL) GNN Formulations

Many GNN models can also be formulated using operations on matrices $\mathbf{X}$, $\mathbf{H}$, $\mathbf{A}$, and others. We will refer to this approach as the **global (GL) linear algebraic** approach.

For example, the GL formulation of the GCN model is $\mathbf{H}^{(l+1)} = ReLU(\widehat{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)})$. $\widehat{\mathbf{A}}$ is the *normalized adjacency matrix with self loops* $\widetilde{\mathbf{A}}$ (cf. Table 1): $\widehat{\mathbf{A}} = \widetilde{\mathbf{D}}^{-\frac{1}{2}}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-\frac{1}{2}}$. This normalization incorporates coefficients $1/\sqrt{d_i d_j}$ shown in the LC formulation above (the original GCN paper gives more details about normalization).

Many GL models use higher powers of $\mathbf{A}$ (or its normalizations). Based on this criterion, GL models can be *linear (L)* (if only the 1st power of $\mathbf{A}$ is used), *polynomial (P)* (if a polynomial power is used), and *rational (R)* (if a rational power is used) [63]. This aspect impacts how to best parallelize a given model, as we illustrate in Section 4. For example, the GCN model [128] is linear.

Importantly, GNN computations involve both sparse and dense matrices. As the performance patterns of operations on such matrices differ vastly [123], [134]–[136], this comes with potential for different parallelization routines. We analyze this in more detail in Section 4.

2.4 GNN Inference vs. GNN Training

A series of GNN layers stacked one after another, as detailed in Figure 2 and in § 2.3, constitutes GNN inference. GNN training consists of three parts: forward pass, loss computation, and backward pass. The forward pass has the same structure as GNN inference. For example, in classification, the loss $\mathcal{L}$ is obtained as follows: $\mathcal{L} = \frac{1}{|\mathcal{Y}|}\sum_{i \in \mathcal{Y}} \text{loss}\left(\mathbf{y}_i, \mathbf{t}_i\right)$, where $\mathcal{Y}$ is a set of all the labeled samples, $\mathbf{y}_i$ is the final prediction for sample i, and $\mathbf{t}_i$ is the ground-truth label for sample i. In practice, one often uses the cross-entropy loss [65]; other functions may also be used [99].

Backpropagation outputs the gradients of all the trainable weights in the model. A standard chain rule is used to obtain mathematical formulations for respective GNN models. For example, the gradients for the first GCN layer, assuming a total of two layers ($L = 2$), are as follows [197]:

$$\nabla_{\mathbf{W}^{(0)}}\mathcal{L} = \left(\widehat{\mathbf{A}}\mathbf{X}\right)^T \left(\sigma'\left(\widehat{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)}\right) \odot \widehat{\mathbf{A}}^T \text{loss}\left(\mathbf{Y} - \mathbf{T}\right) \mathbf{W}^{(1)^T}\right)$$

where $\mathbf{T}$ is a matrix grouping all the ground-truth vertex labels, cf. Table 1 for other symbols. This equation reflects the forward propagation formula (cf. § 2.3.2); the main difference is using transposed matrices (because backward propagation involves propagating information in the reverse direction on the input graph edges) and the derivative of the non-linearity σ'.

The structure of backward propagation depends on whether full-batch or mini-batch training is used. Parallelizing mini-batch training is more challenging due to the inter-sample dependencies, we analyze it in Section 3.

2.5 GNN Programming Models and Operators

Recent works that originated in the systems community come with programming and execution models. These models facilitate GNN computations. In general, they each provide a set of programmable *kernels*, aka *operators* (also referred to as UDFs – User Defined Functions) that enable implementing the GNN functions both in the LC formulation ($\bigoplus, \psi, \phi$) and in the GL formulation (matrix products and others). Figure 5 shows both LC and GL formulations, and how they translate to the programming kernels.

The most widespread programming/execution model is SAGA [153] ("Scatter-ApplyEdge-Gather-ApplyVertex"), used in many GNN libraries [246]. In the Scatter operator, the feature vectors of the vertices u, v adjacent to a given edge (u, v) are processed (e.g., concatenated) to create the data specific to the edge (u, v). Then, in ApplyEdge, this data is transformed (e.g., passed through an MLP). Scatter and ApplyEdge together implement the ψ function. Then, Gather aggregates the outputs of ApplyEdge for each vertex, using a selected commutative and associative operation. This enables implementing the $\bigoplus$ function. Finally, ApplyVertex conducts some user specified operation on the aggregated vertex vectors (implementing ϕ).

Note that, to express the edge related kernels Scatter and UpdateEdge, the LC formulation provides a generic function ψ. On the other hand, to express these kernels in the GL formulation, one adds an element-wise product between the adjacency matrix $\mathbf{A}$ and some other matrix being a result of matrix operations that provide the desired effect. For example, to compute a "vanilla attention" model on graph edges, one uses a product of $\mathbf{H}^{(l)}$ with itself transposed.

Other operators, proposed in GReTA [126], FlexGraph [208], and others, are similar. For example, GReTA has one additional operator, Activate, which enables a separate specification of activation. On the other hand, GReTA does not provide a kernel for applying the ψ function.

We illustrate the relationships between operators and GNN functions from the LC and GL formulations, in Figure 5. Here, we use the name **Aggregate** instead of **Gather** to denote the kernel implementing the $\bigoplus$ function. This is because "Gather" has traditionally been used to denote bringing several objects together into an array [160][1].

Parallelism in these programming and execution models is tightly related to that of the associated GNN functions in LC and GL formulations; we discuss it in Section 4. We also analyze parallel and distributed frameworks and accelerators based on these models in Section 5.

2.6 Taxonomy of Parallelism in GNNs

In traditional deep learning, there are two fundamental ways to parallelize processing a neural network [16]: data parallelism and model parallelism that – respectively – partition data samples and neural weights among different workers. Model parallelism can further be divided into

[1] Another name sometimes used in this context is "Reduce"

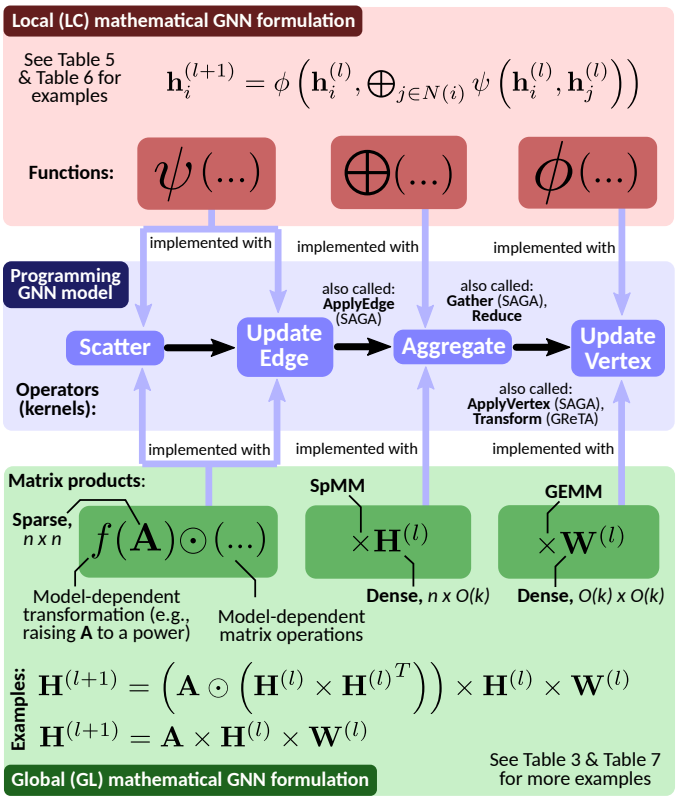

Fig. 5: **(§ 2.3–§ 2.5) Basic elements of GNN model formulations** (top part: the local (LC) approach, bottom part: the global (GL) approach), and **how they translate into GNN operators** (central part). SAGA [153], NAU [208], and GReTA [126] are GNN programming models. Red/green indicate formulations from Figure 4.

pipeline parallelism (different NN layers are processed in parallel) and operator parallelism (a single sample or neural activity is processed in parallel).

We overview the parallelism taxonomy in Figure 6, and show how it translates to parallelism in GNNs in Figure 7. It is similar to that of traditional DL, in that it also has *data parallelism* and *model parallelism*. However, there are certain differences that we identify and analyze.

For example, as we detail in Section 3, data parallelism in GNNs has two variants: *mini-batch parallelism* (when one parallelizes processing a mini-batch, *and updates the weights after each mini-batch*) and *graph [partition] parallelism* (when one parallelizes a batch due to the inability to store a given

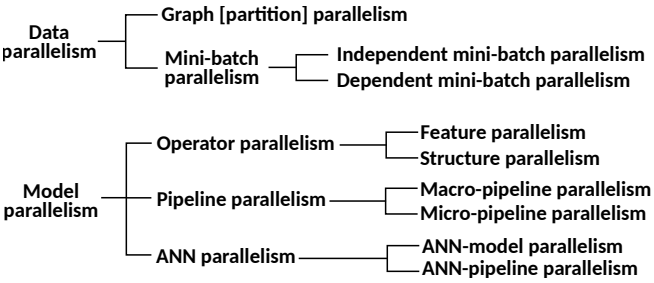

Fig. 6: **(§ 2.6) Parallelism taxonomy in GNNs.**

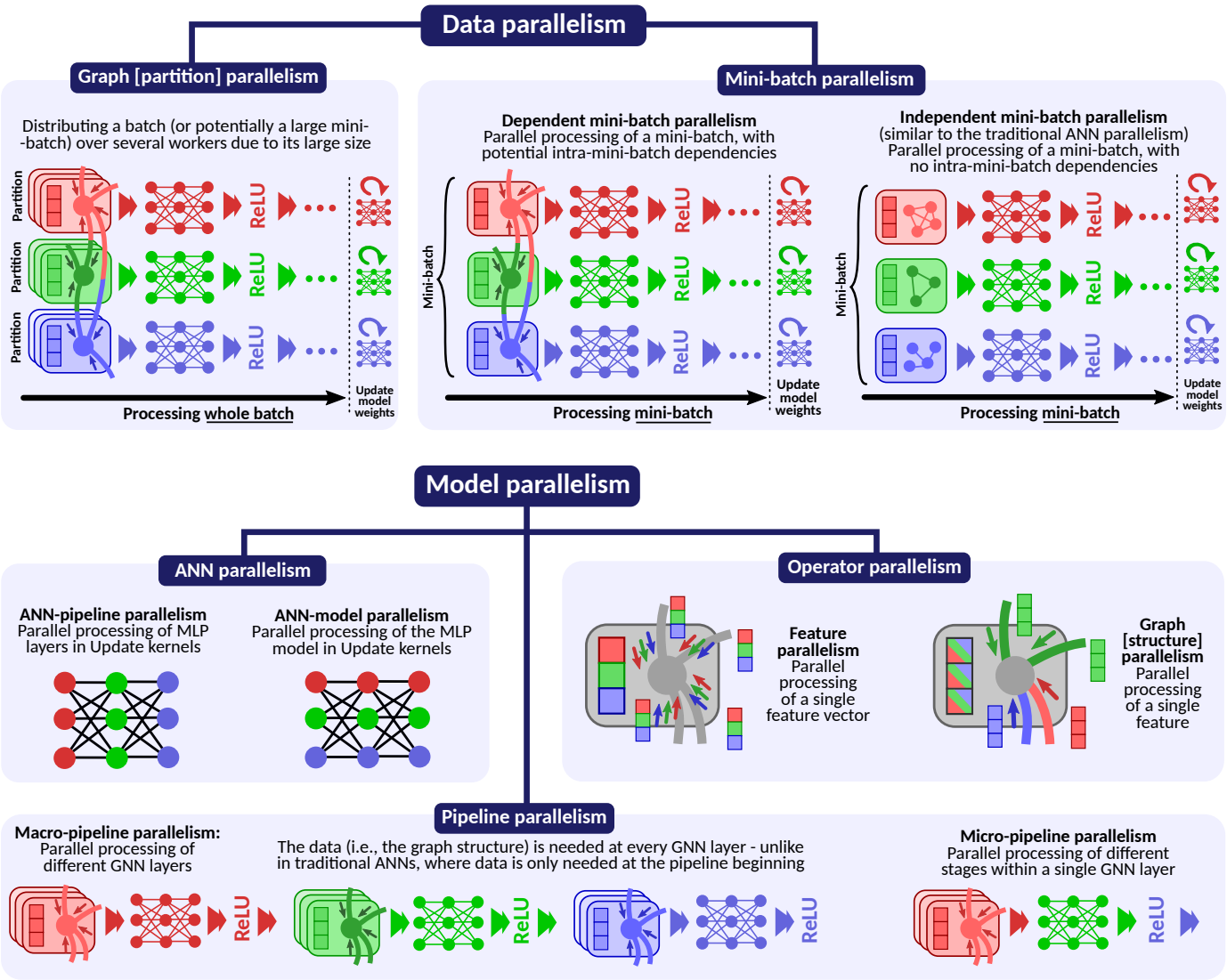

Fig. 7: **(§ 2.6) Overview of parallelism in GNNs. Different colors (red, green, blue) correspond to different workers.**

batch on one worker, *and only updates the weights after the whole batch*). Note that graph partition parallelism could also be applied to a large mini-batch, if that mini-batch cannot be stored on a single worker. Mini-batch parallelism further divides into *dependent mini-batch parallelism* (whenever samples have dependencies between one another) and *independent mini-batch parallelism* (no dependencies between samples). Graph partition parallelism and dependent mini-batch parallelism are much more challenging than their equivalent forms in traditional deep learning because of dependencies between data samples.

Model parallelism in GNNs also has several variants. First, in *pipeline parallelism*, we distinguish *macro-pipeline parallelism* (pipelining the actual GNN layers) and *micro-pipeline parallelism* (pipelining the processing of samples within a single GNN layer). Second, single operators can also be parallelized (*operator parallelism*) in different ways (*feature parallelism* when updating in parallel different features in a single feature vector, and *graph [structure] parallelism* when processing in parallel a single feature by assigning different workers to different neighbors of a given vertex). Finally, the UpdateEdge and UpdateVertex kernels come with dense neural network operations and thus one can apply traditional artificial neural network (ANN) parallelism to these kernels. We refer to this form in general as *ANN parallelism* and we further distinguish *ANN-pipeline parallelism* and *ANN-model parallelism*.

2.7 Parallel and Distributed Models and Algorithms

We use formal models for reasoning about parallelism. For a **single-machine** (**shared-memory**), we use the work-depth (WD) analysis, an established approach for bounding runtimes of parallel algorithms. The *work* of an algorithm is the total number of operations and the *depth* is defined as the longest sequential chain of execution in the algorithm (assuming infinite number of parallel threads executing the algorithm), and it forms the lower bound on the algorithm execution time [39], [42]. One usually wants to minimize depth while preventing work from increasing too much.

In **multi-machine** (**distributed-memory**) settings, one is often interested in understanding the algorithm cost in

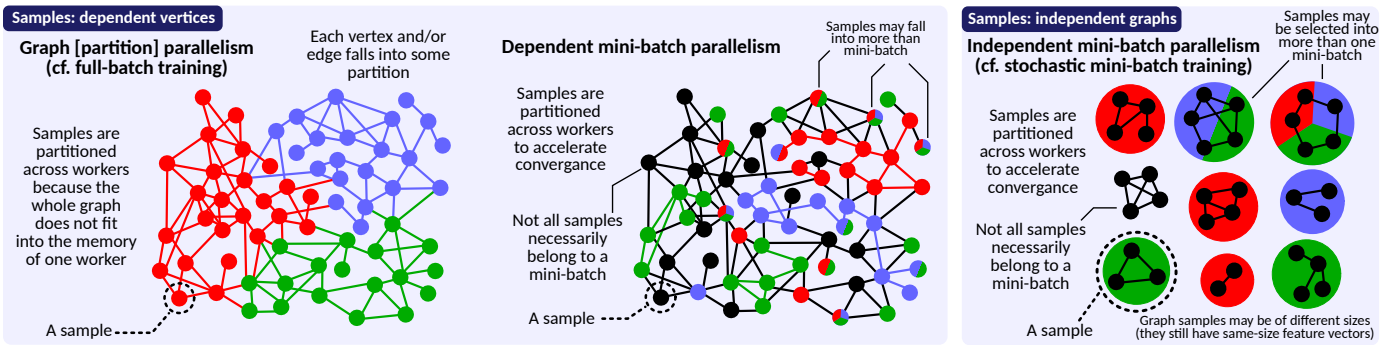

Fig. 8: **(§ 3.1, § 3.2) Graph partition parallelism vs. dependent and independent mini-batch parallelism in GNNs.** Different colors (**red**, **green**, **blue**) indicate different graph partitions or mini-batches, and the associated different workers. Black vertices do not belong to any mini-batch.

terms of the amount of *communication* (i.e., communicated data volume), *synchronization* (i.e., the number of global "supersteps"), and *computation* (i.e., work), and minimizing these factors. A popular model used in this setting is *Bulk Synchronous Parallel (BSP)* [200].

3 Data Parallelism

In traditional deep learning, a basic form of data parallelism is to parallelize the processing of input data samples within a mini-batch. Each worker processes its own portion of samples, computes partial updates of the model weights, and synchronizes these updates with other workers using established strategies such as parameter servers or allreduce [16]. As samples (e.g., pictures) are independent, it is easy to parallelize their processing, and synchronization is only required when updating the model parameters. In GNNs, mini-batch parallelism is more complex because very often, *there are dependencies between data samples* (cf. Figure 3 and § 2.1. Moreover, the input datasets as a whole are often massive. Thus, regardless of whether and how mini-batching is used, one is often forced to resort to graph partition parallelism because no single server can fit the dataset. We now detail both forms of GNN data parallelism. We illustrate them in Figure 8.

3.1 Graph Partition Parallelism

Some graphs may have more than 250 billion vertices and beyond 10 trillion edges [18], [147], and each vertex and/or edge may have a large associated feature vector [110]. Thus, one inevitably must distribute such graphs over different workers as they do not fit into one server memory. We refer to this form of GNN parallelism as the graph partition parallelism, because it is rooted in the established problem of graph partitioning [52], [122] and the associated mincut problem [84], [90], [122]. The main objective in graph partition parallelism is to distribute the graph across workers in such a way that both communication between the workers and work imbalance among workers are minimized.

We illustrate variants of graph partitioning in Figure 9. When distributing a graph over different workers and servers, one can specifically distribute vertices (*edge [structure] partitioning*, i.e., edges are partitioned), edges (*vertex [structure] partitioning*, i.e., vertices are partitioned), or edge and/or vertex input features (*edge/vertex [feature] partitioning*, i.e., edge and/or vertex input feature vectors are partitioned). Importantly, these methods can be combined, e.g., nothing prevents using both edge and feature vector partitioning together. Edge partitioning is probably the most widespread form of graph partitioning, but it comes with large communication and work imbalance when partitioning graphs with skewed degree distributions. Vertex partitioning alleviates it to a certain degree, but if a high-degree vertex is distributed among many workers, it may also lead to large overheads in maintaining a consistent distributed vertex state. Differences between edge and vertex partitioning are covered extensively in rich literature [9], [50]–[52], [55], [73], [73], [75], [94], [104], [121], [122], [124]. Feature vertex partitioning was not addressed in the graph pro-

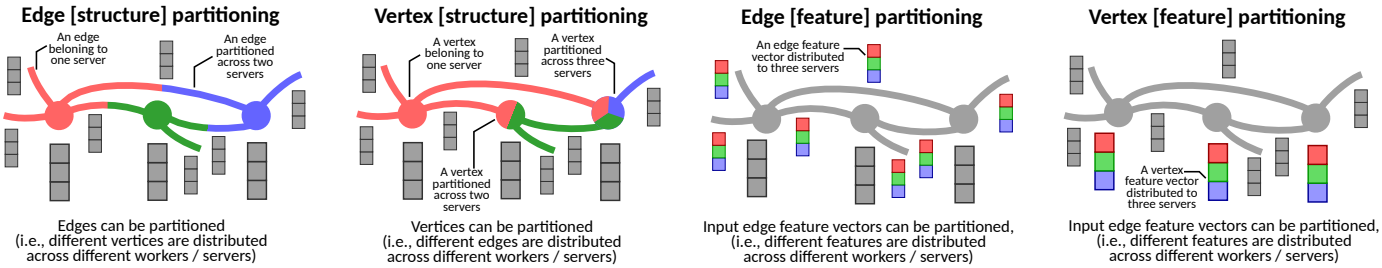

Fig. 9: **(§ 3.1) Different forms of graph partition parallelism.** Different colors (**red**, **green**, **blue**) indicate different graph partitions, and the associated different workers. The gray graph element is oblivious to a given form of partitioning. **Note that different partitioning schemes can be combined together.**

cessing area because usually traditional distributed graph algorithms, vertices and/or edges are associated with scalar values.

Partitioning entails communication when a given part of a graph depends on another part kept on a different server. This may happen during a graph related operator (Scatter, Aggregate) if edges or vertices are partitioned, and during a neural network related operator (UpdateEdge, UpdateVertex) if feature vectors are partitioned.

3.1.1 Full-Batch Training

Graph partition parallelism is commonly used to alleviate large memory requirements of full-batch training. In full-batch training, one must store *all the activations* for *each feature* in *each vertex* in *each GNN layer*). Thus, a common approach for executing and parallelizing this scheme is using distributed-memory large-scale clusters that can hold the massive input datasets in their combined memories, together with graph partition parallelism. Still, using such clusters may be expensive, and it still does not alleviate the slow convergence. Hence, mini-batching is often used.

3.2 Mini-Batch Parallelism

In GNNs, if data samples are independent graphs, then mini-batch parallelism is similar to traditional deep learning. First, one mini-batch is a set of such graph samples, with no dependencies between them. Second, samples (e.g., molecules) may have different sizes (causing potential **load imbalance**), similarly to, e.g., videos [143]. This setting is common in graph classification or graph regression. We illustrate this in Figure 8 (right), and we refer to it as *independent mini-batch parallelism*. Note that while such graph samples may have different sizes (e.g., molecules can have different counts of atoms and bonds), their corresponding feature vectors are of the same dimensionality.

Yet, in most GNN computations, mini-batch parallelism is much more challenging because of inter-sample dependencies (*dependent mini-batch parallelism*). As a concrete example, consider node classification. Similarly to graph partition parallelism, one may experience **load imbalance** issues, e.g., because vertices may differ in their degrees. Moreover, a key challenge in GNN mini-batching is the **information loss** when selecting the *target vertices* forming a given mini-batch. In traditional deep learning, one picks samples randomly. In GNNs, straightforwardly applying such a strategy would result in very low accuracy. This is because, when selecting a random subset of nodes, this subset may not even be connected, but most definitely it will be very sparse and due to the missing edges, a lot of information about the graph structure is lost during the Aggregate or Scatter operator. This information loss challenge was circumvented in the early GNN works with full-batch training [128], [223] (cf. § 3.1.1). Unfortunately, full-batch training comes with slow convergence (because the model is updated only once per epoch, which may require processing billions of vertices), and the above-mentioned large memory requirements. Hence, two more recent approaches that address specifically mini-batching were proposed *incorporating support vertices*, and *appropriately selecting target vertices*.

3.2.1 Support Vertices

In a line of works initiated by GraphSAGE [101], one adds some neighbors of sampled target vertices as so called *support vertices* to the mini-batch. These support vertices are only used to increase the accuracy of predictions for target vertices (i.e., they are not used as target vertices in that mini-batch). Specifically, when executing the Scatter and Aggregate kernels for each of target vertices in a mini-batch, one also considers the pre-selected support vertices. Hence, the results of Scatter and Aggregate are more accurate. Support vertices of each target vertex v usually come from not only 1-hop, but also from k-hop neighborhoods of v, where k may be as large as graph's diameter. The exact selection of support vertices depends on the details of each scheme. In GraphSAGE, they are sampled (for each target vertex) for each GNN layer before the actual training.

We illustrate support vertices in Figure 10. Here, note that the target vertices within each mini-batch may be clustered but may also be spread across the graph (depending on a specific scheme [60], [61], [101], [139]). Support vertices, indicated with darker shades of each mini-batch color, are located up to 2 hops away from their target vertices.

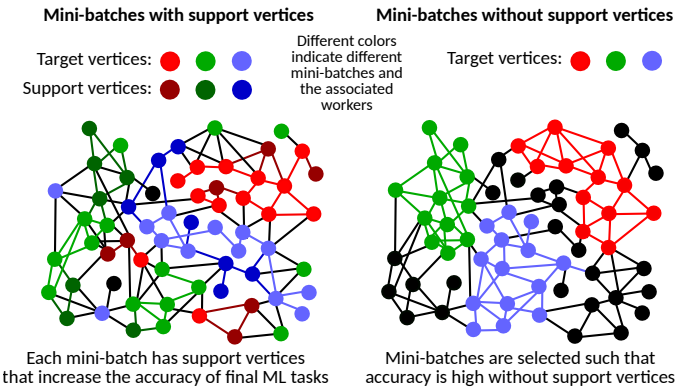

Fig. 10: **(§ 3.2) Two variants of mini-batching**: with and w/o support vertices. Light colors (**red**, **green**, **blue**) indicate different mini-batches (target vertices) and the associated workers. Dark colors (**dark red**, **dark green**, **dark blue**) indicate support vertices for each respective mini-batch. **Black** vertices do not belong to any mini-batch. Example schemes: GraphSAGE [101] (left), Cluster-GCN [65] (right).

One challenge related to support vertices is the **overhead of their pre-selection**. For example, in GraphSAGE, one has to – in addition to the forward and backward propagation passes – conduct as many sampling steps as there are layers in a GNN, to select support vertices for each layer and for each vertex. While this can be alleviated with parallelization schemes also used for forward and backward propagation, it inherently increases the depth of a GNN computation by a multiplicative constant factor.

Another associated challenge is called the **neighborhood explosion** and is related to the memory overhead due to maintaining potentially many such vertices. In the worst case, for each vertex in a mini-batch, assuming keeping all its neighbors up to H hops, one has to maintain $O(kd^H)$ state. Even if some of these vertices are target vertices in that mini-batch and thus are already maintained, when increas-

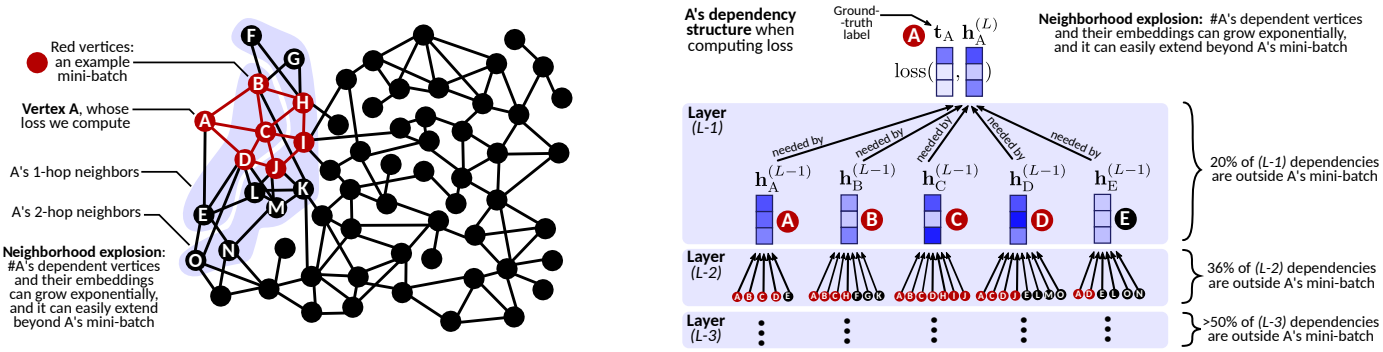

Fig. 11: **(§ 3.2) Neighborhood explosion** in mini-batching in GNNs.

ing H, their ratio becomes lower. GraphSAGE alleviates this by sampling a constant fraction of vertices from each neighborhood instead of keeping all the neighbors, but the memory overhead may still be large [234]. We show an example neighborhood explosion in Figure 11.

3.2.2 Appropriate Selection of Target Vertices

More recent GNN mini-batching works focus on the appropriate selection of target nodes included in mini-batches, such that support vertices are not needed for high accuracy. For example, Cluster-GCN first clusters a graph and then assign clusters to be mini-batches [65], [230]. This way, one reduces (but not eliminates) the loss of information because a mini-batch usually contains a tightly knit community of vertices. We illustrate this in Figure 10 (right). However, one has to additionally compute graph clustering as a form of preprocessing. This can be parallelized with one of many established parallel clustering routines [29], [30], [36], [122].

3.2.3 Discussion

We first observe that the **key difference between graph partition parallelism** and **mini-batch parallelism** is the *timing of updating model weights*. It takes place after the whole batch (for the former) and after each mini-batch (for the latter). Other differences are as follows. First, the primary objective when partitioning a graph is to minimize communication and work imbalance across workers. Contrarily, in mini-batching, one aims at a selection of target vertices that maximizes accuracy. Second, each vertex belongs to some partition(s), but not each vertex is necessarily included in a mini-batch. Third, while mini-batch parallelism has a variant with no inter-sample dependencies, graph partition parallelism nearly always deals with a connected graph and has to consider such dependencies.

We also note that one could consider the asynchronous execution of different mini-batches. This would entail **asynchronous GNN training**, with model updates being conducted asynchronously. Such a scheme could slow down convergence, but would offer potential for more parallelism.

3.3 Work-Depth Analysis: Full-Batch vs. Mini-Batch

We analyze work/depth of different GNN training schemes that use full-batch or mini-batch training, see Table 2.

First, all methods have a common term in work being $O(Lmk + Lnk^2)$ that equals the number of layers L times the number of operations conducted in each layer, which

Method	Work	Depth
Full-batch training schemes:		
Full-batch [128]	$O\left(Lmk + Lnk^2\right)$	$O\left(L\log k + L\log d\right)$
Weight-tying [139]	$O\left(Lmk + Lnk^2\right)$	$O\left(L\log k + L\log d\right)$
RevGNN [139]	$O\left(Lmk + Lnk^2\right)$	$O\left(L\log k + L\log d\right)$
Mini-batch training schemes:		
GraphSAGE [101]	$O\left(Lmk + Lnk^2 + c^L nk^2\right)$	$O\left(L\log k + L\log c\right)$
VR-GCN [60]	$O\left(Lmk + Lnk^2 + c^L nk^2\right)$	$O\left(L\log k + L\log c\right)$
FastGCN [61]	$O\left(Lmk + Lnk^2 + cLnk^2\right)$	$O\left(L\log k + L\log c\right)$
Cluster-GCN [65]	$O\left(W_{pre} + Lmk + Lnk^2\right)$	$O\left(D_{pre} + L\log k + L\log d\right)$
GraphSAINT [234]	$O\left(W_{pre} + Lmk + Lnk^2\right)$	$O\left(D_{pre} + L\log k + L\log d\right)$

TABLE 2: **(§ 3.3) Work-depth analysis of GNN training methods.** c is the number of vertices sampled per neighborhood or per GNN layer.

is mk for sparse graph operations (Aggregate) and nk^2 for dense neural network operations (UpdateVertex). This is the total work for full-batch methods. Mini-batch schemes have additional work terms. Schemes based on support vertices (GraphSAGE, VR-GCN, FastGCN) have terms that reflect how they pick these vertices. GraphSAGE and VR-GCN have a particularly high term $O(c^L nk^2)$ due to the neighborhood explosion (c is the number of vertices sampled per neighborhood). FastGCN alleviates the neighborhood explosion by sampling c vertices per whole layer, resulting in $O(cLnk^2)$ work. Then, approaches that focus on appropriately selecting target vertices (GraphSAINT, Cluster-GCN) do not have the work terms related to the neighborhood explosion. Instead, they have preprocessing terms indicated with W_{pre}. Cluster-GCN's W_{pre} depends on the selected clustering method, which heavily depends on the input graph size (n, m). GraphSAINT, on the other hand, does stochastic mini-batch selection, the work of which does not necessarily grow with n or m.

In terms of depth, all the full-batch schemes depend on the number of layers L. Then, in each layer, two bottleneck operations are the dense neural network operation (UpdateVertex, e.g., a matrix-vector multiplication) and the sparse graph operation (Aggregate). They take $O(\log k)$ and $O(\log d)$ depth, respectively. Mini-batch schemes are similar, with the main difference being the $O(\log c)$ instead of $O(\log d)$ term for the schemes based on support vertices. This is because Aggregate in these schemes is applied over c sampled neighbors. Moreover, in Cluster-GCN and Graph-

SAINT, the neighborhoods may have up to d vertices, yielding the $O(\log d)$ term. They however have the additional preprocessing depth term D_{pre} that depends on the used sampling or clustering scheme.

To summarize, full-batch and mini-batch GNN training schemes have similar depth. Note that this is achieved using graph partition parallelism in full-batch training methods, and mini-batch parallelism in mini-batching schemes. Contrarily, overall work in mini-batching may be larger due to the overheads from support vertices, or additional preprocessing when selecting target vertices using elaborate approaches. However, mini-batching comes with faster convergence and usually lower memory pressure.

3.4 Tradeoff Between Parallelism & Convergence

The *efficiency tradeoff* between the amount of parallelism in a mini-batch and the convergence speed, controlled with the mini-batch size, is an important part of parallel traditional ANNs [16]. In short, small mini-batches would accelerate convergence but may limit parallelism while large mini-batches may slow down convergence but would have more parallelism. In GNNs, finding the "right" mini-batch size is much more complex, because of the inter-sample dependencies. For example, a large mini-batch consisting of vertices that are not even connected, would result in very low accuracy. On the other hand, if a mini-batch is small but it consists of tightly connected vertices that form a cluster, then the accuracy of the updates based on processing that mini-batch can be high [65].

4 Model Parallelism

In traditional neural networks, models are often large. In GNNs, models ($\mathbf{W}$) are usually small and often fit into the memory of a single machine. However, numerous forms of model parallelism are heavily used to improve throughput; we provided an overview in § 2.6 and in Figure 7.

In the following model analysis, we often picture the used linear algebra objects and operations. For clarity, we indicate their shapes, densities, and dimensions, using small figures, see Table 3 for a list. Interestingly, GNN models in the LC formulations heavily use dense matrices and vectors with dimensionalities dominated by $O(k)$, and the associated operations. On the other hand, the GL formulations use both sparse and dense matrices of different shapes (square, rectangular, vectors), and the used matrix multiplications can be dense–dense (GEMM, GEMV), dense–sparse (SpMM), and sparse–sparse (SpMSpM). Other operations are elementwise matrix products or rational sparse matrix powers. This rich diversity of operations immediately illustrates a huge potential for parallel and distributed techniques to be used with different classes of models.

4.1 Operator Parallelism

When analyzing operator parallelism, we first focus on the LC formulations, and then proceed to the GL formulation.

4.1.1 Parallelism in LC Formulations of GNN Models

We illustrate generic work and depth equations of LC GNN formulations in Figure 12. Overall, work is the sum of any preprocessing costs W_{pre}, post-processing costs W_{post}, and work of a single GNN layer W_l times the number of layers L. In the considered generic formulation in Eq. (8), W_l equals to work needed to evaluate ψ for each edge (mW_ψ), $\bigoplus$ for each vertex ($nW_\oplus$), and ϕ for each vertex (nW_ϕ). Depth is analogous, with the main difference that the depth of a single GNN layer is a plain sum of depths of computing ψ, $\bigoplus$, and ϕ (each function is evaluated in parallel for each vertex and edge, hence no multiplication with n or m).

Symbol	Description	Used often in
	Matrices and vectors	
,	Dense vectors, **dimensions**: $O(k) \times 1$, $1 \times O(k)$	LC models
	Dense matrices, **dimensions**: $O(k) \times O(k)$	GL & LC models
,	Dense matrices, **dimensions**: $n \times O(k)$, $O(k) \times n$	GL models
	Sparse matrix, **dimensions**: $n \times n$	GL models
	Matrix multiplications (dimensions as stated above)	
×	GEMM, dense tall matrix × dense square matrix	GL models
×	GEMM, dense square matrix × dense square matrix	GL models
×	GEMM, dense tall matrix × dense tall matrix	GL models
×	GEMV, dense matrix × dense vector	LC models
×	SpMM, sparse matrix × dense matrix	GL models
	Elementwise matrix products and other operations	
$\odot$ (...)	Elementwise product of a sparse matrix and some object	GL models
^{x}, $x \in \mathbb{N}$	SpMSpM, sparse matrix × sparse matrix	GL models
^{x}, $x \in \mathbb{Z}$	Rational sparse matrix power	GL models
$\cdot$, $\odot$	Vector dot product, elementwise vector product	LC models
$\parallel$, $\sum$	Vector concatenation, sum of d vectors, $d \leq n$	LC models

TABLE 3: Important objects and operations from linear algebra used in GNNs. **We assign these operations to specific GNN models in Tables 5, 6, and 7**.

$$
\begin{aligned}
\text{Work (generic)} &= W_{pre} + LW_l + W_{post} \\
\text{Depth (generic)} &= D_{pre} + LD_l + D_{post} \\
\text{Work of Eq. (8)} &= W_{pre} + L \cdot \left(mW_\psi + nW_\oplus + nW_\phi\right) + W_{post} \\
\text{Depth of Eq. (8)} &= D_{pre} + L \cdot \left(D_\psi + D_\oplus + D_\phi\right) + D_{post}
\end{aligned}
$$

Fig. 12: Generic equations for work and depth in GNN LC formulations.

We now analyze work and depth of many specific GNN models, by focusing on the three functions forming these models: ψ, $\bigoplus$, and ϕ. The analysis outcomes are in Tables 5 and 6. We select the representative models based on a recent survey [63]. We also indicate whether a model belongs to the class of convolutional (C-GNN), attentional (A-GNN), or message-passing (MP-GNN) models [169] (cf. § 2.3.1).

Analysis of ψ We show the analysis results in Table 5. We provide the formulation of ψ for each model, and we

also illustrate all algebraic operations needed to obtain ψ. All C-GNN models have their ψ determined during preprocessing. This preprocessing corresponds to the adjacency matrix row normalization ($c_{ij} = 1/d_i$), the column normalization ($c_{ij} = 1/d_j$), or the symmetric normalization ($c_{ij} = 1/\sqrt{d_i d_j}$) [223]. In all these cases, their derivation takes $O(1)$ depth and $O(m)$ work. Then, A-GNNs and MP-GNNs have much more complex formulations of ψ than C-GNNs. Details depend on the model, but - importantly - nearly all the models have $O(k^2)$ work and $O(\log k)$ depth. The most computationally intense model, GAT, despite having its work equal to $O(dk^2)$, has also logarithmic depth of $O(\log k + \log d)$. This means that computing ψ in all the considered models can be effectively parallelized. As for the sparsity pattern and type of operations involved in evaluating ψ, most models use GEMV. All the considered A-GNN models also use transposition of dense vectors. GAT also uses vector concatenation and sum of up to d vectors. Finally, one considered MP-GNN model uses an elementwise MV product. In general, each considered GNN model uses dense matrix and vector operations to obtain ψ for each of the associated edges.

Analysis of $\bigoplus$ The aggregate operator $\bigoplus_{j \in N(i)}$ is almost always a commutative and associative operation such as min, max, or plain sum [79], [209]. While it operates on vectors $\mathbf{x}_j$ of dimensionality k, each dimension can be computed independently of others. Thus, to compute $\bigoplus_{j \in N(i)}$, one needs $O(\log d_i)$ depth and $O(kd_i)$ work, using established parallel tree reduction algorithms [41]. Hence, $\bigoplus$ is the bottleneck in depth in all the considered models. This is because d (maximum vertex degree) is usually much larger than k.

Analysis of ϕ The analysis of ϕ is shown in Table 6 (for the same models as in Table 5). We show the total model work and depth. All the models entail matrix-vector dense products and a sum of up to d dense vectors. Depth is logarithmic. Work varies, being the highest for GAT.

We also illustrate the operator parallelism in the LC formulation, focusing on the GNN programming kernels, in the top part of Figure 13. We provide the corresponding generic work-depth analysis in Table 4, and we also assess communication and synchronization (discussed separately in § 4.1.5). The four programming kernels follow the work and depth of the corresponding LC functions ($\psi, \oplus, \phi$).

Kernel	Work	Depth	Comm.	Sync.
Scatter (ψ)	$O(1)$	$O(1)$	$O(mk)$	$O(1)$
UpdateEdge (ψ)	$O(mW_\psi)$	$O(D_\psi)$	$O(1)$	$O(1)$
Aggregate ($\oplus$)	$O(nW_\oplus)$	$O(D_\oplus \log d)$	$O(mk)$	$O(1)$
UpdateVertex (ϕ)	$O(nW_\phi)$	$O(D_\phi)$	$O(1)$	$O(1)$

TABLE 4: **Work-depth analysis of GNN operators (kernels).**

4.1.2 Parallelism in GL Formulations of GNN Models

Parallelism in GL formulations is analyzed in Table 7. The models with both LC and FG formulations (e.g., GCN) have the same work and depth. Thus, fundamentally, they offer the same amount of parallelism. However, the GL formulations based on matrix operations come with potential for different parallelization approaches than the ones used for the LC formulations. For example, there are more opportunities to use vectorization, because one is not forced to vectorize the processing of feature vectors for each vertex or edge separately (as in the LC formulation), but instead one could vectorize the derivation of the whole matrix $\mathbf{H}$ [31].

There are also models designed in the GL formulations with no known LC formulations, cf. Tables 5–6. These are models that use polynomial and rational powers of the adjacency matrix, cf. § 2.3.2 and Figure 4. These models have only one iteration. They also offer parallelism, as indicated by the logarithmic depth (or square logarithmic for rational models requiring inverting the adjacency matrix [161]). While they have one iteration, making the L term vanish, they require deriving a given power x of the adjacency matrix $\mathbf{A}$ (or its normalized version). Importantly, as computing these powers is not interleaved with non-linearities (as is the case with many models that only use linear powers of $\mathbf{A}$), the increase in work and depth is *only logarithmic*, indicating more parallelism. Still, their representative power may be lower, due to the lack of non-linearities.

We overview two example GL models (GCN and vanilla graph attention) in Figure 13 (bottom). In this figure, we also indicate how the LC GNN kernels are reflected in the flow of matrix operations in the GL formulation.

4.1.3 Instantiation of Edge Feature Vectors

The LC formulations as specified by Eq. (8) (C-GNNs, A-GNNs, MP-GNNs) enable explicit instantiation of vertex feature vectors. However, in some cases, one may also want to explicitly instantiate edge feature vectors. Such instantiation would be used in, for example, edge classification or edge regression tasks. An example GNN formulation that enables this is Graph Networks by Battaglia et al. [14], also an LC formulation. The insights about parallelism in such a formulation are not different than the ones provided in this section; the central difference lies in the fact that, in MP-GNNs, edge feature vectors ψ are "transient" and used primarily as input for obtaining vertex feature vectors.

4.1.4 Feature vs. Structure vs. Model Weight Parallelism

Feature parallelism is straightforward in both LC and GL formulations (cf. Figure 13). In the former, one can execute binary tree reductions over different features in parallel (feature parallelism in $\bigoplus$), or update edge or vertex features in parallel (feature parallelism in ψ and ϕ). In the latter, one can multiply a row of an adjacency matrix with any column of the latent matrix $\mathbf{H}$ (corresponding to different features) in parallel. As feature vectors are dense, they can be stored contiguously in memory and easily used with vectorization.

Graph structure parallelism is also accessible in both LC and GL formulations. In the former, it is present via parallel execution of $\bigoplus$ (for a single specific feature). In the latter, one simply parallelizes the multiplication of a given adjacency matrix row with a given feature matrix column.

Traditional model weight parallelism, in which one partitions the weight matrix $\mathbf{W}$ across workers, is also possible in GNNs. Yet, due to the small sizes of weight matrices used so far in the literature [77], [110], it was not yet the focus of research. If this parallelism becomes useful in the feature, one could use traditional deep learning techniques to parallelize the model weight processing [16].

Reference	Class	Formulation for $\psi(\mathbf{h}_i, \mathbf{h}_j)$	Dimensions & density of one execution of $\psi(\mathbf{h}_i, \mathbf{h}_j)$	Pr?	Work & depth of one execution of $\psi(\mathbf{h}_i, \mathbf{h}_j)$	
GCN [128]	C-GNN	$\frac{1}{\sqrt{d_i d_j}}\mathbf{h}_j$	$c \cdot$	▬	$O(k)$	$O(1)$
GraphSAGE [101] (mean)	C-GNN	$\mathbf{h}_j$		▬	$O(1)$	$O(1)$
GIN [226]	C-GNN	$\mathbf{h}_j$		▬	$O(1)$	$O(1)$
CommNet [192]	C-GNN	$\mathbf{h}_j$		▬	$O(1)$	$O(1)$
Vanilla attention [201]	A-GNN	$\left(\mathbf{h}_i^T \cdot \mathbf{h}_j\right)\mathbf{h}_j$	$\left(\cdot \right) \cdot$	✖	$O(k)$	$O(\log k)$
MoNet [158]	A-GNN	$\exp\left(-\frac{1}{2}\left(\mathbf{h}_j - \mathbf{w}_j\right)^T \mathbf{W}_j^{-1}\left(\mathbf{h}_j - \mathbf{w}_j\right)\right)$	$\exp\left(\times \times \right)$	✖	$O\left(k^2\right)$	$O(\log k)$
GAT [202]	A-GNN	$\frac{\exp\left(\sigma\left(\mathbf{a}^T \cdot \left[\mathbf{W}\mathbf{h}_i \middle\Vert \mathbf{W}\mathbf{h}_j\right]\right)\right)}{\sum_{y \in \widehat{N}(i)} \exp\left(\sigma\left(\mathbf{a}^T \cdot \left[\mathbf{W}\mathbf{h}_i \middle\Vert \mathbf{W}\mathbf{h}_y\right]\right)\right)}\mathbf{h}_j$	$\frac{\exp\left(\cdot \left[\times \middle\Vert \times \right]\right)}{\sum \exp\left(\cdot \left[\times \middle\Vert \times \right]\right)} \cdot$	✖	$O\left(dk^2\right)$	$O\left(\log k + \log d\right)$
Attention-based GNNs [196]	A-GNN	$w\frac{\mathbf{h}_i^T \cdot \mathbf{h}_j}{\Vert\mathbf{h}_i\Vert\Vert\mathbf{h}_j\Vert}\mathbf{h}_j$	$\left(\cdot \right) \cdot$	✖	$O(k)$	$O(\log k)$
G-GCN [47]	MP-GNN	$\sigma\left(\mathbf{W}_1\mathbf{h}_i + \mathbf{W}_2\mathbf{h}_j\right) \odot \mathbf{h}_j$	$\left(\times \right) \odot$	✖	$O(k^2)$	$O(\log k)$
GraphSAGE [101] (pooling)	MP-GNN	$\sigma\left(\mathbf{W}\mathbf{h}_j + \mathbf{w}\right)$	$\times$	✖	$O(k^2)$	$O(\log k)$
EdgeConv [216] "choice 1"	MP-GNN	$\mathbf{W}\mathbf{h}_j$	$\times$	✖	$O(k^2)$	$O(\log k)$
EdgeConv [216] "choice 5"	MP-GNN	$\sigma\left(\mathbf{W}_1\left(\mathbf{h}_j - \mathbf{h}_i\right) + \mathbf{W}_2\mathbf{h}_i\right)$	$\times$	✖	$O(k^2)$	$O(\log k)$

TABLE 5: **Comparison of local (LC) formulations of GNN models with respect to the inner function $\psi(\mathbf{h}_i, \mathbf{h}_j)$.** For clarity and brevity of equations, when it is obvious, we omit the matrix multiplication symbol $\times$ and the indices of a given iteration (GNN layer) number (l). "**Class**": class of a GNN model with respect to the complexity of ψ, details are in Section 2. All models considered in this table feature aggregations over 1-hop neighbors ("Type L", details are in Section 2). "**Dimensions & density**": dimensions and density of the most important tensors and tensor operations when computing $\psi(\mathbf{h}_i, \mathbf{h}_j)$ in a given model. "**Pr**": can coefficients in ψ be preprocessed (▬), or do they have to be learnt (✖)? When listing the most important tensor operations, we focus on multiplications.

Reference	Class	Formulation of ϕ for $\mathbf{h}_i^{(l)}$; $\psi(\mathbf{h}_i, \mathbf{h}_j)$ are stated in Table 5	Dimensions & density of computing $\phi(\cdot)$, excluding $\psi(\cdot)$	Work & depth (a whole training iteration or inference, including ψ from Table 5)	
GCN [128]	C-GNN	$\mathbf{W} \times \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_j\right)\right)$	$\times \sum$	$O(Lmk + Lnk^2)$	$O(L\log d + L\log k)$
GraphSAGE [101] (mean)	C-GNN	$\mathbf{W} \times \left(\frac{1}{d_i} \cdot \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_j\right)\right)\right)$	$\times \sum$	$O(Lmk + Lnk^2)$	$O(L\log d + L\log k)$
GIN [226]	C-GNN	$\mathrm{MLP}\left((1+\epsilon)\mathbf{h}_i + \sum_{j \in N(i)} \psi\left(\mathbf{h}_j\right)\right)$	$\overbrace{\times \ldots \times}^{K\text{ times}} \times \sum$	$O(Lmk + LKnk^2)$	$O(L\log d + LK\log k)$
CommNet [192]	C-GNN	$\mathbf{W}_1\mathbf{h}_i + \mathbf{W}_2 \times \left(\sum_{j \in N^+(i)} \psi\left(\mathbf{h}_j\right)\right)$	$\times \sum$	$O(Lmk + Lnk^2)$	$O(L\log d + L\log k)$
Vanilla attention [201]	A-GNN	$\mathbf{W} \times \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)\right)$	$\times \sum$	$O(Lmk + Lnk^2)$	$O(L\log d + L\log k)$
GAT [202]	A-GNN	$\mathbf{W} \times \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)\right)$	$\times \sum$	$O(Lmdk^2 + Lnk^2)$	$O(L\log d + L\log k)$
Attention-based GNNs [196]	A-GNN	$\mathbf{W} \times \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)\right)$	$\times \sum$	$O(Lmk + Lnk^2)$	$O(L\log d + L\log k)$
MoNet [158]	A-GNN	$\mathbf{W} \times \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_j\right)\right)$	$\times \sum$	$O(Lmk^2 + Lnk^2)$	$O(L\log d + L\log k)$
G-GCN [47]	MP-GNN	$\mathbf{W} \times \left(\sum_{j \in N^+(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)\right)$	$\times \sum$	$O(Lmk^2 + Lnk^2)$	$O(L\log d + L\log k)$
GraphSAGE [101] (pooling)	MP-GNN	$\left(\mathbf{W} \times \left(\mathbf{h}_i \middle\Vert \left(\max_{j \in N(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)\right)\right)\right)$	$\times \left(\middle\Vert \left(\times \sum \right)\right)$	$O(Lmk^2 + Lnk^2)$	$O(L\log d + L\log k)$
EdgeConv [216] "choice 1"	MP-GNN	$\sum_{j \in N^+(i)} \psi\left(\mathbf{h}_j\right)$	$\sum$	$O(Lmk^2 + Lnk^2)$	$O(L\log d + L\log k)$
EdgeConv [216] "choice 5"	MP-GNN	$\max_{j \in N^+(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)$	$\sum$	$O(Lmk^2 + Lnk^2)$	$O(L\log d + L\log k)$

TABLE 6: **Comparison of local (LC) formulation of GNN models with respect to the outer function ϕ.** For clarity and brevity of equations, when it is obvious, we omit the matrix multiplication symbol $\times$ and the indices of a given iteration number (l); we also omit activations from the formulations (these are elementwise operations, not contributing to work or depth). "**Class**": class of a GNN model with respect to the complexity of ψ, details are in Section 2. All models considered in this table feature aggregations over 1-hop neighbors ("Type L", details are in Section 2). "**Dimensions & density**": dimensions and density of the most important tensors and tensor operations in a given model when computing $\mathbf{h}_i^{(l)}$. When listing the most important tensor operations, we focus on multiplications.

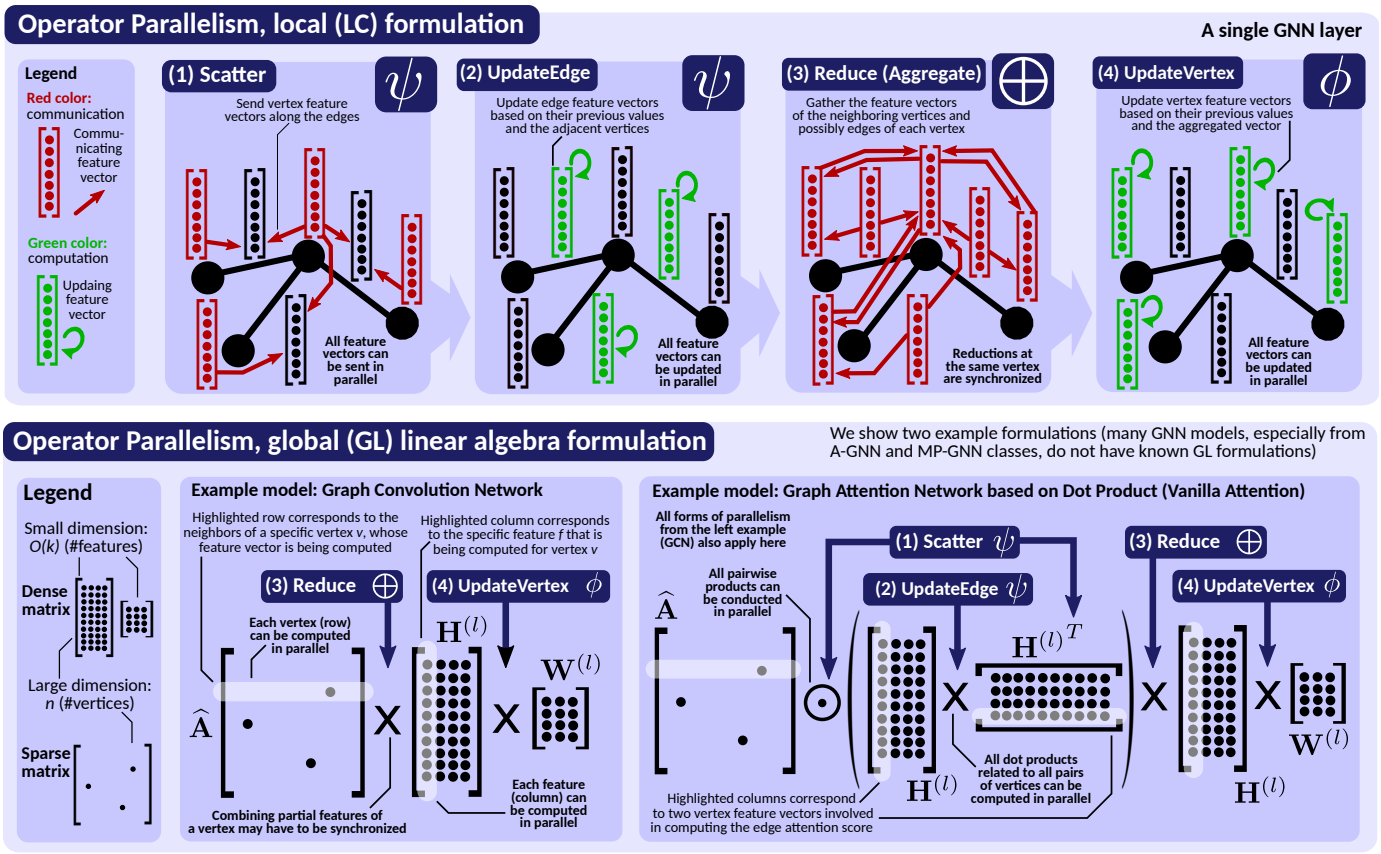

Fig. 13: **Operator parallelism in GNNs** for LC formulations (top) and GL formulations (bottom).

Reference	Type	Algebraic formulation for $\mathbf{H}^{(l+1)}$	Dimensions & density of deriving $\mathbf{H}^{(l+1)}$	#I	Work & depth (one whole training iteration or inference)	
GCN [128]	L	$\widehat{\mathbf{A}}\mathbf{H}\mathbf{W}$	$\times\ \times$	L	$O(mkL + Lnk^2)$	$O(L\log k + L\log d)$
GraphSAGE [101] (mean)	L	$\widehat{\mathbf{A}}\mathbf{H}\mathbf{W}$	$\times\ \times$	L	$O(mkL + Lnk^2)$	$O(L\log k + L\log d)$
GIN [226]	L	$\text{MLP}\left(((1+\epsilon)\mathbf{I} + \widehat{\mathbf{A}})\mathbf{H}\right)$	$\times\ \times\ \times\ \ldots\ \times$	L	$O(mkL + KLnk^2)$	$O(LK\log k + LK\log d)$
CommNet [192]	L	$\mathbf{A}\mathbf{H}\mathbf{W}_2 + \mathbf{H}\mathbf{W}_1$	$\times\ \times\ +\ \times$	L	$O(mkL + Lnk^2)$	$O(L\log k + L\log d)$
Dot Product [201]	L	$(\mathbf{A} \odot (\mathbf{H}\mathbf{H}^T))\,\mathbf{H}\mathbf{W}$	$\odot\ (\times\ \times)\ \times\ \times$	L	$O(Lmk + Lnk^2)$	$O(L\log k + L\log d)$
EdgeConv [216] "choice 1"	L	$\mathbf{A}\mathbf{H}\mathbf{W}$	$\times\ \times$	L	$O(mkL + Lnk^2)$	$O(L\log k + L\log d)$
SGC [219]	P	$\widehat{\mathbf{A}}^s\mathbf{H}\mathbf{W}$	$^s\ \times\ \times$	1	$O(mn\log s + nk^2)$	$O(\log k + \log s\log d)$
DeepWalk [168]	P	$\left(\sum_{s=0}^{T}\overline{\mathbf{A}}^s\right)\mathbf{H}\mathbf{W}$	$(\ ^0 + \ldots + \ ^T)\ \times\ \times$	1	$O(mn\log T + nk^2)$	$O(\log k + \log T\log d)$
ChebNet [72]	P	$\left(\sum_{s=0}^{T}\theta_s\overline{\mathbf{A}}^s\right)\mathbf{H}\mathbf{W}$	$(\ ^0 + \ldots + \ ^T)\ \times\ \times$	1	$O(mn\log T + nk^2)$	$O(\log k + \log T\log d)$
DCNN [6], GDC [130]	P	$\left(\sum_{s=1}^{T}w_s\overline{\mathbf{A}}^s\right)\mathbf{H}\mathbf{W}$	$(\ ^1 + \ldots + \ ^T)\ \times\ \times$	1	$O(mn\log T + nk^2)$	$O(\log k + \log T\log d)$
Node2Vec [97]	P	$\left(\frac{1}{p}\mathbf{I} + \left(1-\frac{1}{q}\right)\overline{\mathbf{A}} + \frac{1}{q}\overline{\mathbf{A}}^2\right)\mathbf{H}\mathbf{W}$	$(\ ^0 + \ldots + \ ^2)\ \times\ \times$	1	$O(mn + nk^2)$	$O(\log k + \log d)$
LINE [148], SDNE [207]	P	$\left(\overline{\mathbf{A}} + \theta\overline{\mathbf{A}}^2\right)\mathbf{H}\mathbf{W}$	$(\ + \ ^2)\ \times\ \times$	1	$O(mn + nk^2)$	$O(\log k + \log d)$
Auto-Regress [250], [256]	R	$\left((1+\alpha)\mathbf{I} - \alpha\widehat{\mathbf{A}}\right)^{-1}\mathbf{H}\mathbf{W}$	$^{-1}\ \times\ \times$	1	$O(n^3 + nk^2)$	$O(\log^2 n + \log k)$
PPNP [43], [129], [230]	R	$\alpha\left(\mathbf{I} - (1-\alpha)\widehat{\mathbf{A}}\right)^{-1}\mathbf{H}\mathbf{W}$	$^{-1}\ \times\ \times$	1	$O(n^3 + nk^2)$	$O(\log^2 n + \log k)$
ARMA [38], ParWalks [221]	R	$b\left(\mathbf{I} - a\widehat{\mathbf{A}}\right)^{-1}\mathbf{H}\mathbf{W}$	$^{-1}\ \times\ \times$	1	$O(n^3 + nk^2)$	$O(\log^2 n + \log k)$

TABLE 7: **Comparison of global (GL) linear algebra formulations of GNN models.** For clarity and brevity of equations, when it is obvious, we omit the matrix multiplication symbol $\times$ and the indices of a given iteration number (l); we also omit activations from the formulations (these are elementwise operations, not contributing to work or depth). "**Type**": type of a GNN model with respect to the scope of accessed vertex neighbors, details are in Section 2 ("**L**": adjacency matrix is used in its 1st power, "**P**": adjacency matrix is used in its polynomial power, "**R**": adjacency matrix is used in its rational power). "**#I**": the number of GNN layers (GNN iterations). "**Dimensions & density**": dimensions and density of the most important tensors and tensor operations in a given model. When listing the most important tensor operations, we focus on multiplications.

4.1.5 Communication & Synchronization

Communication in the LC formulations takes place in the Scatter kernel (a part of ψ) if vertex feature vectors are communicated to form edge feature vectors; transferred data amounts to $O(mk)$. Similarly, during the Aggregate kernel ($\bigoplus$), there can also be $O(mk)$ data moved. Both UpdateEdge (ψ) and UpdateVertex (ϕ) do not explicitly move data. However, they may be associated with communication intense operations; especially A-GNNs and MP-GNNs often have complex processing associated with ψ and ϕ, cf. Tables 5 and 6. While this processing entails matrices of dimensions of up to $O(k) \times O(k)$, which easily fit in the memory of a single machine, this may change in the future, *if* the feature dimensionality k is increased in future GNN models.

In the default synchronized variants of GNN, computing all kernels of the same type must be followed by global synchronization, to ensure that all data has been received by respective workers (after Scatter and Aggregate) or that all feature vectors have been updated (after UpdateEdge and UpdateVertex). In Section 4.2.3, we discuss how this requirement can be relaxed by allowing asynchronous execution.

Communication and synchronization in the GL formulations heavily depend on the used matrix representations and operations. Specifically, there have been a plethora of works into communication reduction in matrix operations, for example targeting dense matrix multiplications [86], [134]–[136] or sparse matrix multiplications [92], [187], [188], [190]. They could be used with different GNN operations (cf. Table 3) and different models (cf. Table 7). The exact bounds would depend on the selected schemes. Importantly, many works propose to trade more storage for less communication by different forms of input matrix replication [188]. This could be used in GNNs for more performance.

4.2 Pipeline Parallelism

Pipelining has two general benefits. First, it increases the throughput of a computation, lowering the overall processing runtime. This is because more operations can finish in a time unit. Second, it reduces memory pressure in the computation. Specifically, one can divide the input dataset into chunks, and process these chunks separately via pipeline stages, having to keep a fraction of the input in memory at a time. In GNNs, pipelining is often combined with graph partition parallelism, with partitions being such chunks. We distinguish two main forms of GNN pipelines: micro-pipelines and macro-pipelines, see Figure 7 and § 2.6.

4.2.1 Micro-Pipeline Parallelism

In micro-pipeline parallelism, the pipeline stages correspond to the operations within a GNN layer. Here, for simplicity, we consider a graph operation followed by a neural operation, followed by a non-linearity, cf. Figure 2. One can equivalently consider kernels (Scatter, UpdateEdge, Aggregate, UpdateVertex) or the associated functions ($\psi, \oplus, \phi$). Such pipelining enables reducing the length of the sequence of executed operators by up to 3×, effectively forming a 3-stage operator micro-pipeline. There have been several practical works into micro-pipelining GNN operators, especially using HW accelerators; we discuss them in Section 5.

We show an example micro-pipeline (synchronous) in the top panel of Figure 14. Observe that each neural operation must wait for *all* graph operations to finish, because – in the worst case – in each partition, there may be vertices with edges to all other partitions. This is an important difference to traditional deep learning (and to a GNN setting with independent graphs, cf. Figure 3), where chunks have no inter-chunk dependencies, and thus neural processing of P1

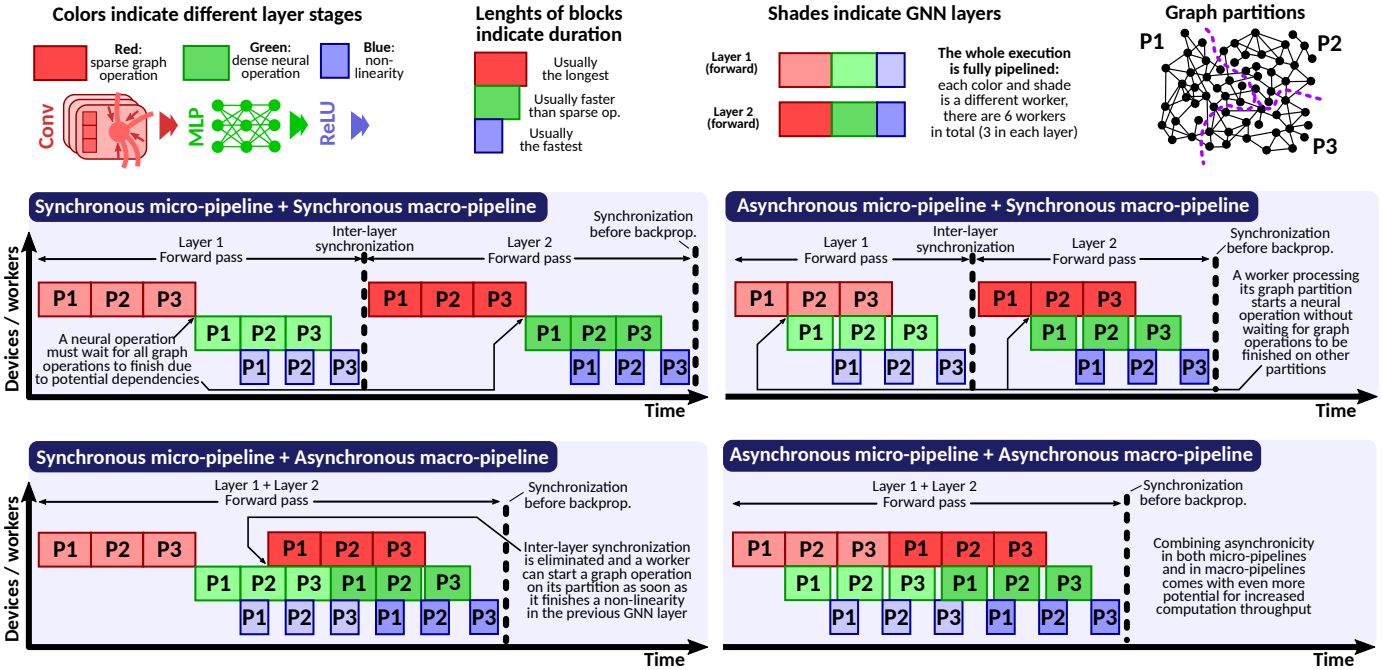

Fig. 14: **(§ 4.2) Overview of pipelining combined with graph partition parallelism** (top-left panel), and of **(§ 4.2.3) asynchronous pipelined execution (other panels)**. Each of four example GNN executions processes three graph partitions (P1, P2, P3) on three stages: **red** (a sparse graph operation such as convolution), **green** (a dense neural operation such as MLP), and **blue**) (a non-linearity such as ReLU). We use two GNN layers (shades indicate layers). The whole execution is fully pipelined, i.e., there are 6 workers in total (three workers for each stage in each layer).

could start right after finishing the graph operation on P1.

The exact benefits from micro-pipelining in depth depend on a concrete GNN model. Assuming a simple GCN, the four operations listed above take, respectively, $O(\log d)$, $O(1)$, and $O(1)$ depth. Thus, as Aggregate takes asymptotically more time, one could replicate the remaining stages, in order to make the pipeline balanced.

4.2.2 Macro-Pipeline Parallelism

In macro-pipeline parallelism, pipeline stages are GNN layers. Such pipelines are subject to intense research in traditional deep learning, with designs such as GPipe [116], PipeDream [164], or Chimera [144]. However, pipelining GNN layers is more difficult because of dependencies between data samples, and it is only in its early development stage [197]. In Figure 14, the execution is fully pipelined, i.e., all layers are processed by different workers.

There is an interesting **difference between GNN macro-pipelines and the traditional ANN pipelines**. Specifically, in the latter, the data is only needed at the pipeline beginning. In the former, the data (i.e., the graph structure) is needed *at every GNN layer*.

4.2.3 Asynchronous Pipelining

In asynchronous pipelining, pipeline stages proceed without waiting for the previous stages to finish [164]. This notion can be applied to both micro- and macro-pipelines in GNNs. First, in **asynchronous micro-pipelines**, a worker processing its graph partition starts a neural operation without waiting for graph operations to be finished on other partitions (Figure 14, top-right panel). Second, in **asynchronous macro-pipelines**, the inter-layer synchronization is eliminated and a worker can start a graph operation on its partition as soon as it finishes a non-linearity in the previous GNN layer (Figure 14, bottom-left panel). Finally, **both forms can be combined**, see Figure 14 (bottom-right).

Note that asynchronous pipelining can be used with *both graph partitions* (i.e., asynchronous processing of different graph partitions) *and with mini-batches* (i.e., asynchronous processing of different mini-batches).

4.2.4 Theoretical Formulation of Arbitrarily Deep Pipelines

To understand GNN pipelining better, we first provide a variant of Eq. (8), namely Eq. (2), which defines a *synchronous Message-Passing GNN execution with graph partition parallelism*. In this equation, we explicitly illustrate that, when computing Aggregation ($\oplus$) of a given vertex i, some of the aggregated neighbors may come from "remote" graph partitions, where i does not belong; such i's neighbors form a set $N^{\mathcal{R}}(i)$. Other neighbors come from the same "local" graph partition, forming a set $N^{\mathcal{L}}(i)$. Note that $N^{\mathcal{R}}(i) \cup N^{\mathcal{L}}(i) = N(i)$. Moreover, in Eq. (2), we also explicitly indicate the current training iteration t in addition to the current layer l by using a double index (t, l). Overall, Eq. (2) describes a synchronous standard execution because, to obtain a feature vector in the layer l and in the training iteration t, all used vectors come from the previous layer $l-1$, in the same training iteration t.

Different forms of staleness and asynchronicity can be introduced by modifying the layer indexes so that they "point more to the past", i.e., use stale feature vectors from past layers. For this, we generalize Eq. (2) into Eq. (3) by incorporating parameters to fully control the scope of such staleness. These parameters are L_ϕ (controlling the staleness of i's own previous feature vector), $L_\psi^{\mathcal{L}}$ (controlling the staleness of feature vectors coming from i's local neighbors from i's partition), and $L_\psi^{\mathcal{R}}$ (controlling the staleness of feature vectors coming from i's remote neighbors in other partitions). Moreover, to also *allow for staleness and asynchronicity with respect to training iterations*, we introduce the analogous parameters $T_\phi, T_\psi^{\mathcal{L}}, T_\psi^{\mathcal{R}}$. We then define the behavior of Eq. (3) such that these six parameters *upper bound the maximum allowed staleness*, i.e., Eq. (3) can use feature vectors from past layers/iterations *at most* as old as controlled by the given respective index parameters.

Now, first observe that when setting $L_\phi = L_\psi^{\mathcal{L}} = L_\psi^{\mathcal{R}} = 1$ and $T_\phi = T_\psi^{\mathcal{L}} = T_\psi^{\mathcal{R}} = 0$, we obtain the standard synchronous equation (cf. Figure 14, top-left panel). Setting any of these parameters to be larger than this introduces staleness. For example, PipeGCN [206] proposes to pipeline communication and computation *between training iterations* in the GCN model [128] by using $T_\psi^{\mathcal{R}} = 1$ (all other parameters are zero). This way, the model is allowed to use stale feature vectors coming from *remote partitions in previous training iterations*, enabling communication-computation overlap (at the cost of somewhat longer convergence). Another option would be to *only* set $L_\psi^{\mathcal{R}} = 2$ (or to a higher value). *This would enable asynchronous macro-pipelining*, because one does not have to wait for the most recent GNN layer to finish processing other graph partitions to start processing its own feature vector. We leave the exploration of other asynchronous designs based on Eq. (3) for future work.

Finally, we also obtain the equivalent formulations for

Standard computation with graph partition parallelism:

$$\mathbf{h}_i^{(t,l)} = \phi\left(\mathbf{h}_i^{(t,l-1)}, \bigoplus_{j \in N^{\mathcal{L}}(i)} \psi\left(\mathbf{h}_i^{(t,l-1)}, \mathbf{h}_j^{(t,l-1)}\right) \bigoplus_{j \in N^{\mathcal{R}}(i)} \psi\left(\mathbf{h}_i^{(t,l-1)}, \mathbf{h}_j^{(t,l-1)}\right)\right) \tag{2}$$

Using bounded stale feature vectors with graph partition parallelism (worst case):

$$\mathbf{h}_i^{(t,l)} = \phi\left(\mathbf{h}_i^{(t-T_\phi,l-L_\phi)}, \bigoplus_{j \in N^{\mathcal{L}}(i)} \psi\left(\mathbf{h}_i^{(t-T_\phi,l-L_\phi)}, \mathbf{h}_j^{\left(t-T_\psi^{\mathcal{L}},l-L_\psi^{\mathcal{L}}\right)}\right) \bigoplus_{j \in N^{\mathcal{R}}(i)} \psi\left(\mathbf{h}_i^{(t-T_\phi,l-L_\phi)}, \mathbf{h}_j^{\left(t-T_\psi^{\mathcal{R}},l-L_\psi^{\mathcal{R}}\right)}\right)\right) \tag{3}$$

Fig. 15: **(§ 4.2.3) Generalization of the Message Passing GNN formulation** that includes graph partition parallelism combined with fully synchronous (top) and potentially stale asynchronous computation (bottom).

Standard computation:

$$\nabla \mathbf{h}_j^{(t,l)} = \sum_{\substack{i \in V: \\ j \in N^{\mathcal{L}}(i)}} \nabla \mathbf{h}_i^{(t,l+1)} + \sum_{\substack{i \in V: \\ j \in N^{\mathcal{R}}(i)}} \nabla \mathbf{h}_i^{(t,l+1)} \tag{4}$$

Using bounded stale gradients (worst case):

$$\nabla \mathbf{h}_j^{(t,l)} = \sum_{\substack{i \in V: \\ j \in N^{\mathcal{L}}(i)}} \nabla \mathbf{h}_i^{(t-T^{\mathcal{L}},l+L^{\mathcal{L}})} + \sum_{\substack{i \in V: \\ j \in N^{\mathcal{R}}(i)}} \nabla \mathbf{h}_i^{(t-T^{\mathcal{R}},l+L^{\mathcal{R}})} \tag{5}$$

Fig. 16: **(§ 4.2.3) Generalization of computing gradients in GNNs** to include graph partition parallelism combined with fully synchronous (top) and potentially stale asynchronous computation (bottom).

the asynchronous computation of *stale gradients*, see Figure 16. This establishes a similar approach for optimizing backward propagation passes.

4.2.5 Beyond Micro- and Macro-Pipelining

We note that the above two forms of pipelining do not necessarily exhaust all opportunities for pipelined execution in GNNs. For example, there is extensive work on parallel pipelined reduction trees [107] that could be used to further accelerate the Aggregate operator ($\bigoplus$).

4.3 Artificial Neural Network (ANN) Parallelism

Finally, note that in some GNN models, for example GIN [226], the dense UpdateVertex or UpdateEdge kernels may be MLPs. These MLPs can be parallelized using the traditional deep learning approaches [16]. Specifically, we distinguish two such approaches: **ANN-model parallelism** (parallel processing of MLP parameters within one layer) and **ANN-pipeline parallelism** (parallel pipelined processing of consecutive MLP layers), cf. Figure 7.

4.4 Other Forms of Parallelism in GNNs

One could identify other forms of parallelism in GNNs. First, by combining model and data parallelism, one obtains – as in traditional deep learning – *hybrid parallelism* [132]. More elaborate forms of model parallelism are also possible. An example is Mixture of Experts (MoE) [154], in which different models could be evaluated in parallel. Currently, MoE usage in GNNs is in its infancy [109], [252].

5 Frameworks, Accelerators, Techniques

We finally analyze existing GNN SW frameworks and HW accelerators[2]. For this, we first describe parallel and distributed architectures used by these systems.

5.1 Parallel and Distributed Computing Architectures

There are both single-machine (single-node, often shared-memory) or multi-machine (multi-node, often distributed-memory) GNN systems.

5.1.1 Single-Machine Architectures

Multi- or manycore parallelism is usually included in general-purpose CPUs. Graphical Processing Units (GPUs) offer massive amounts of parallelism in a form of a large number of simple cores. However, they often require the compute problems to be structured so that they fit the "regular" GPU hardware and parallelism. Moreover, Field Programmable Gate Arrays (FPGAs) are well suited for problems that easily form pipelines. Finally, novel proposals include processing-in-memory (PIM) [29], [163] that brings computation closer to data.

GNNs feature both irregular operations that are "sparse" (i.e., entailing many random memory accesses), such as reductions over neighborhoods, and regular "dense" operations, such as transformations of feature vectors, that are usually dominated by sequential memory access patterns [1]. The latter are often suitable for effective GPU processing while the former are easier to be processed effectively on the CPU. Thus, both architectures are highly relevant in the context of GNNs. Our analysis (Table 8, the top part) indicates that they are both supported by more than 50% of the available GNN processing frameworks. We observe that most of these designs focus on executing regular dense GNN operations on GPUs, leaving the irregular sparse computations for the CPU. While being an effective approach, we note that GPUs were successfully used to achieve very high performance in irregular graph processing [185], and they thus have high potential for also accelerating sparse GNN operations.

There is also interest in HW accelerators for GNNs (Table 8, the bottom part). Most are ASIC proposals (some are evaluated using FPGAs); several of them incorporate PIM. With today's significance of heterogeneous computing, developing GNN-specific accelerators and using them in tandem with mainstream architectures is an important thread of work that, as we predict, will only gain more significance in the foreseeable future.

5.1.2 Multi-Machine Parallelism

While shared-memory systems are sufficient for processing many datasets, a recent trend in GNNs is to increase the size of input graphs [110], which often requires multi-node settings to avoid expensive I/Os. We observe (Table 8, the top part) that different GNN software frameworks support distributed-memory environments. However, the majority of them focus on training, leaving much room for developing efficient distributed-memory frameworks and techniques for GNN inference. We also note high potential in incorporating high-performance interconnect related mechanisms such as Remote Direct Memory Access (RDMA) [87], SmartNICs [28], [74], [106], or novel network topologies and routing [26], [34] into the GNN domain.

5.2 General Categories of Systems & Design Decisions

The first systems supporting GNNs usually belonged to one of two categories. The first category are systems constructed on top of graph processing frameworks and paradigms that add neural network processing capabilities (e.g., Neugraph [153] or Gunrock [213]). On the contrary, systems in the second category (e.g., the initial PyG design [79]) start with deep learning frameworks, and extend it towards graph processing capabilities. These two categories usually focus on – respectively – the LC and GL formulations and associated execution paradigms.

[2] We encourage participation in this analysis. In case the reader possesses additional relevant information, such as important details of systems not mentioned in the current paper version, the authors would welcome the input.

Reference	Arch.	Ds?	T?	I?	Op?	mp?	Mp?	Dp?	Dpp	PM	Remarks
[SW] PipeGCN [206]	CPU+GPU	■	■ (fb)	✖	■	■	✖	■	sh	LC	
[SW] BNS-GCN [205]	GPU	■	■ (fb)	✖	?	?	✖	■	sh		
[SW] PaSca [243]	GPU	■	■	■	?	?	?	■			
[SW] Marius++ [204]	CPU	✖	■ (mb)	✖	■ (v)	?	✖	■		LC (SU)	Focus on using disk
[SW] BGL [151]	GPU	■	■ (mb)	✖	■	■	✖	■	sh	—	
[SW] DistDGLv2 [248]	CPU+GPU	■	■ (mb)	✖	■	■	✖	■	sh	—	
[SW] SAR [159]	CPU	■	■ (fb)	✖	✖	✖	✖	■			
[SW] DeepGalois [105]	CPU	■	■ (fb)	✖	?	✖	✖	■ (v)	sh	LC (AU)	
[SW] DistGNN [155]	CPU	■	■ (fb)	✖	?	✖	✖	■ (v)	sh	LC(AU) ?	
[SW] DGCL [54]	GPU	▪*	■	✖	?	✖	✖	■ (v)		LC (AU)	*Only two servers used.
[SW] Seastar [222]	GPU	✖	■	✖	■ (f)	✖	✖	■ (v, t)		LC (VC)	
[SW] Chakaravarthy [56]	GPU	■	■ (fb)	✖	✖	✖	✖	■ (v, sn)		✖	
[SW] Zhou et al. [249]	CPU	✖	✖	■	■ (f)	✖	✖	?		✖	
[SW] MC-GCN [12]	GPU	▪*	■ (fb)	✖	■ (f)	✖	✖	■ (v)		GL	*Multi-GPU within one node.
[SW] Dorylus [197]	CPU	■	■ (fb)	✖	✖	✖	■	■ (v)		LC (SAGA)	
[SW] Min et al. [156]	GPU	▪*	■ (mb)	✖	?	✖	■	■ (v)		GL	*Multi-GPU within one node.
[SW] GNNAdvisor [215]	GPU	✖	■	■	■ (f, s)	✖	✖	?		GL, LC	
[SW] AliGraph [255]	CPU	■	■	■	?	✖	?	■		LC (NAU)	
[SW] FlexGraph [208]	CPU	■	■ (fb)	✖	■ (s)	?	✖	■		LC (NAU)	
[SW] Kim et al. [125]	CPU+GPU	✖	■ (mb)	✖	■ (s)	?	✖	■		LC (AU)	
[SW] AGL [237]	CPU	■	■ (mb)	■	✖	■	■	■		MapReduce	
[SW] ROC [117]	CPU+GPU	■	■ (fb)	■	✖	✖	✖	■		✖	
[SW] DistDGL [247]	CPU	■	■ (mb)	✖	?	✖	✖	■		✖	
[SW] PaGraph [10], [149]	GPU	▪*	■ (mb)	✖	?	✖	✖	■		✖	*Multi-GPU within one node.
[SW] 2PGraph [240]	GPU	■	■ (mb)	✖	?	■	?	■		—	
[SW] GMLP [242]	GPU	■	■ (mb)	✖	?	✖	?	■		LC	
[SW] fuseGNN [64]	GPU	✖	■	✖	■	✖	✖	■		LC (AU)*	*Two aggregation schemes are used.
[SW] P^3 [81]	CPU+GPU	■	■ (mb)	✖	■	■	✖	■		LC (SAGA)*	*A variant called P-TAGS
[SW] QGTC [214]	GPU	✖	■	✖	?	✖	▪	■	sh	GL	
[SW] CAGNET [199]	CPU+GPU	■	■ (fb)	✖	■ (f, s)	✖	✖	■ (v, e)	sh+rep	GL	
[SW] PCGCN [198]	CPU+GPU	✖	■	✖	✖	✖	✖	■ (e)	sh	—	
[SW] FeatGraph [111]	CPU, GPU	✖	■ (fb)	■	■ (f, s)	✖	✖	■ (v)	sh	GL	
[SW] G^3 [150]	GPU	✖	■	■	?	?	?	?		GL	
[SW] NeuGraph [153]	GPU	■	■	✖	▪	■	✖	■ (v, e)	sh	LC (SAGA)	
[SW] PyTorch-Direct [79]	GPU	■	■	■	■	?	?	■ (v, e)		GL, LC	
[SW] PyG [79]	CPU, GPU	■	■*	■	■	?	?	■ (v, e)		GL, LC	*Mini-batching for graph components
[SW] DGL [209]	CPU, GPU	■	■*	■	■	?	▪	■		GL, LC	*Mini-batching for graph components
[HW] ZIPPER [245]	new	✖	✖	■	▪	■	✖	■ (v, e)	sh	GL, LC	
[HW] GCNear [253]	new (PIM)	✖	■ (fb)	✖	■	■	✖	■ (v, e)	sh	LC (AU)	
[HW] BlockGNN [254]	new	✖	■	■	▪	■	✖	■		—	
[HW] TARe [103]	new (ReRAM)	✖	✖	■	▪	■	✖	?		GL	
[HW] Rubik [62]	new	✖	■ (mb)	✖	▪	?	✖	■ (v, e)	sh	LC (AU)	
[HW] GCNAX [141]	new	✖	✖	■	▪	?	✖	?		GL	
[HW] Li et al. [140]	new	✖	✖	■	■	■	■	■		LC (AU)	
[HW] GReTA [126]	new	✖	✖	■	?	?	?	■	sh	LC (GReTA)	
[HW] GNN-PIM [217]	new (PIM)	✖	✖	■	?	■	✖	■ (v)	sh	LC (SAGA)	
[HW] EnGN [145]	new	✖	✖	■	▪	?	✖	■ (v, e)	sh	LC (AU + "feature extraction" stage)	
[HW] HyGCN [228]	new	✖	✖	■	■	■	✖	■ (v, e)	sh	LC (AU)	
[HW] AWB-GCN [85]	new	✖	✖	■	▪	■	■	■ (v, e)	sh	GL	
[HW] GRIP [127]	new	✖	✖	■	?	■	▪	■ (v, e)	sh	GL, LC (GReTA)	
[HW] Zhang et al. [235]	new	✖	✖	■	■	■	▪	■ (v, e)	sh	GL	
[HW] GraphACT [233]	new	✖	■ (mb)	✖	▪	■	✖	?		GL	
[HW] Auten et al. [7]	new	✖	✖	■	?	?	?	?		LC (AU)	

TABLE 8: Comparison of GNN processing frameworks and analyses of GNN implementations. They are grouped by the targeted architecture. Within each group, the systems are sorted by publication date. "**[SW]**": a software framework or package, "**[HW]**": a hardware accelerator. "—": not relevant for a given system. "**Ds?**" (distributed memory): does a design target distributed environments such as clusters, supercomputers, or data centers? "**Arch.**": targeted architecture. "**new**": a new proposed architecture. "**T**": Focus on training? (if more details are provided, we distinguish between "**fb**": full batch, and "**mb**": mini batch). "**I**": Focus on inference? "**Op**": Support for operator parallelism ("**f**": feature parallelism, "**s**": structure parallelism)? "**mp**": Support for micro-pipeline parallelism? "**Mp**": Support for macro-pipeline parallelism? "**Dp**": Support for graph partition parallelism (if more details are provided, we distinguish between "**v**": vertex partitioning (also called 1D partitioning), "**e**": edge partitioning (also called 2D partitioning), "**t**": type partitioning (in heterogeneous GNNs, where a vertex can have multiple types), or "**sn**": snapshot partitioning (in dynamic GNNs, where a graph can be stored in multiple snapshots)). "**Dpp**": data partitioning policy (if more details are provided, we distinguish between "**sh**": sharding, and "**rep**": replication). "**PM**": Used programming model or paradigm: "**GL**": Global, linear algebra based, focusing on operations on matrices such as SpMM or GEMM, "**LC**": Fine Grained, focusing on operations of graph elements, such as neighborhood aggregation. If more details are provided, we further distinguish "**AU**": Aggregate + Update. "**NAU**": Neighborhood selection + Aggregate + Update. "**SAGA**": Scatter + ApplyEdge (i.e., Update Edge) + Gather + ApplyVertex (i.e., Update Vertex). "**GReTA**": Gather + Reduce (i.e., Aggregate) + Transform (i.e., Update) + Activate. "■": Support. "▪": Partial / limited support. "✖": No support. "?": Unknown.

The third, most recent, category of GNN systems, does not start from either traditional graph processing or deep learning. Instead, they target GNN computations from scratch, focusing on GNN-specific workload characteristics and design decisions [113], [171], [172], [215]. For example, Zhang et al. [238] analyze the computational graph of GNNs, and propose optimizations tailored specifically for GNNs.

5.3 Parallelism in GNN Systems

The most commonly supported form of parallelism is **graph partition parallelism**. Here, one often reuses a plethora of established techniques from the graph processing domain [52]. Unsurprisingly, most schemes used for graph partitioning are based on assigning vertices to workers ("1D partitioning"). This is easy to develop, but comes with challenges related to load balancing. Better load balancing properties can often be obtained when also partitioning each neighborhood among workers ("2D partitioning"). While some frameworks support this variant, there is potential for more development into this direction. We also observe that most systems support sharding, attacking a node or edge classification/regression scenario with a single large input graph. Here, CAGNET [199] combines sharding with replication, in order to accelerate GNN training by communication avoidance [189], a technique that have been used to speed up many linear algebra based computations [30].

The majority of works use graph partition parallelism on its own, to alleviate large sizes of graph inputs (by distributing it over different memory resources) and to accelerate GNN computations (by parallelizing the execution of one GNN layer). Some systems ZIPPER [245] combine graph partition parallelism with pipelining, offering further speedups and reductions in used storage resources.

Operator parallelism is supported by the majority of systems. Both **feature parallelism** and **structure parallelism** have been investigated, and there are systems supporting both (FeatGraph [111], GNNAdvisor [215], CAGNET [199]). Most of these systems target these forms of parallelism explicitly (e.g., by programming binary reduction trees processing Reduce). For example, Seastar [222] focuses on combining three types of parallel execution (feature-level, vertex-level, edge-wise). On the other hand, CAGNET is an example design that supports operator parallelism implicitly, by incorporating 2D and 3D distributed-memory matrix products and the appropriate partitioning of $\mathbf{A}$ and $\mathbf{H}$ matrices. We note that existing works often refer to operator parallelism differently (e.g., intra-phase dataflow" [82]).

Micro-pipeline parallelism is widely supported by HW accelerators. We conjecture this is because it is easier to use a micro-pipeline in the HW setting, where there already often exist such pipeline dedicated HW resources.

Macro-pipeline parallelism is least often supported. This is caused by its complexity: one must consider how to pipeline the processing of interrelated nodes, edges, or graphs, across GNN layers. While it is relatively straightforward to use pipeline parallelism when processing graph samples in the context of graph classification or regression, most systems focus on the more complex node/edge classification or regression. Here, two examples are GRIP [127] and work by Zhang et al. [235], where pipelining is enabled by simply loading the weights of the next GNN layer, while the current layer is being processed.

5.4 Optimizations in GNN Systems

We also summarize parallelism related optimizations.

Frameworks that enable data parallelism use different forms of **graph partitioning**. The primary goal is to minimize the edges crossing graph partitions in order to reduce communication. Here, some designs (e.g., DeepGalois [105], DistGNN [155]) use vertex cuts. Others (e.g., DGCL [54], QGTC [214]) incorporate METIS partitioning [122]. ROC [117] proposes an interesting scheme, in which it uses online linear regression to effectively repartition the graph across different training iterations, minimizing the execution time.

There are numerous other schemes used for **reducing communication volume**. For example, DeepGalois [105] and DGCL [54] use message combination and aggregation, DGCL also balances loads on different interconnect links, DistGNN [155] delays updates to optimize communication, Min et al. [156] use zero copy in the CPU–GPU setting (GPU threads access host memory without requiring much CPU interaction) and computation & communication overlap, and GNNAdvisor [215] and 2PGraph [240] renumber nodes to achieve more locality [194]. GNNAdvisor also provides locality-aware task scheduling [114]. 2PGraph also increases the amount of locality with cluster based mini-batching (it increases the number of sampled vertices that belong to the same neighborhood in a mini-batch), a scheme similar to the approach taken by the Cluster-GCN model [65].

Moreover, there are many interesting **operator related optimizations**. A common optimization is operator fusion (sometimes referred to as "kernel fusion" [64]), in which one merges the execution of different operators, for example Aggregate and UpdateVertex [54], [64], [209], [222]. This enables better on-chip data reuse and reduces the number of invoked operators. While most systems fuse operators within a single GNN layer, QGTC offers operator fusion also across different GNN layers [214]. Other interesting schemes include operator reorganization [238], in which one ensures that operators first perform neural operations (e.g., MLP) and only then the updated feature vectors are propagated. This limits redundant computation.

Many systems come with optimizations related to the used **graph representation**. For example, PCGCN [198] provides a hybrid representation, in which it uses CSR to maintain sparse parts of the adjacency matrix, and dense bitmaps for dense edge blocks. Further, many designs focus on ensuring cache friendlines: DistGNN [155] uses tiling (referred to as "blocking"), PaGraph [149] and 2PGraph [240] provide effective feature caching, and Lin et al. [146] propose lossy compression to fit into GPU memory.

Some systems **hybridize** various aspects of GNN computing. For example, Dorylus [197] executes graph-related sparse operators (e.g., Aggregate) on CPUs, while dense tensor related operators (e.g., UpdateVertex) run on Lambdas. fuseGNN [64] dynamically switches between incorporated execution paradigms: dense operators such as UpdateVertex are executed with the GL paradigm and the corresponding GEMM kernels, while sparse computations such as Aggregate use the graph-related paradigms such as SAGA.

Many other schemes exist. For example, Zhang et al. [238] reduce memory consumption (intermediate data are recomputed in the backward pass, instead of being cached), while He et al. [102] incorporate serverless computing [67] for more efficient GNNs.

5.5 Analyses and Evaluations of GNN Systems

Finally, there are several works dedicated to analyses and evaluations of various techniques. Garg et al. [82] investigate different strategies for mapping sparse and dense GNN operators on various HW accelerators. Serafini et al. [181] compare sample-based and full-graph trainig. Yan et al. [227] analyze the performance of GCNs on GPUs. Wang et al. [218] and Zhang et al. [246] extend this analysis to a wider choice of GNN models. Baruah et al. [13] introduce GNNMark, a benchmarking suite for GNNs on GPUs. Tailor et al [193] analyze the performance of point cloud GNNs. Qiu et al. [171] analyze the impact of sparse matrix data format on the performance of GNN computations.

6 Other Categories & Aspects of GNNs

We also briefly discuss other categories, variants, and aspects of GNN models.

6.1 Classes of Local GNN Models

Depending on the details of $\bigoplus$, ψ, and ϕ, one distinguishes three **GNN classes** [169]: *Convolutional GNNs* (C-GNNs), *Attentional GNNs* (A-GNNs), and *Message-Passing GNNs* (MP-GNNs). Each class is defined by an equation specifying the feature vector $\mathbf{h}_i^{(l+1)}$ of a vertex i in the next GNN layer $l+1$. The three equations to update a vertex are as follows:

$$\mathbf{h}_i^{(l+1)} = \phi\left(\mathbf{h}_i^{(l)}, \bigoplus_{j \in N(i)} c_{ij} \cdot \psi\left(\mathbf{h}_j^{(l)}\right)\right) \quad \text{(C-GNNs)} \tag{6}$$

$$\mathbf{h}_i^{(l+1)} = \phi\left(\mathbf{h}_i^{(l)}, \bigoplus_{j \in N(i)} a\left(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}\right) \cdot \psi\left(\mathbf{h}_j^{(l)}\right)\right) \quad \text{(A-GNNs)} \tag{7}$$

$$\mathbf{h}_i^{(l+1)} = \phi\left(\mathbf{h}_i^{(l)}, \bigoplus_{j \in N(i)} \psi\left(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}\right)\right) \quad \text{(MP-GNNs)} \tag{8}$$

The overall difference between these classes lies in the expressiveness of operations acting on edges. C-GNNs enable fixed precomputed *scalar* edge weights c_{ij}, A-GNNs enable *scalar* edge weights that may be the result of arbitrarily complex operations on learnable parameters $a(\mathbf{h}_i, \mathbf{h}_j)$, and MP-GNNs enable arbitrarily complex edge weights $\psi(\mathbf{h}_i, \mathbf{h}_j)$.

Specifically, in **C-GNNs**, Eq. (6), the feature vector $\mathbf{h}_j^{(l)}$ of each neighbor j is first transformed using a function ψ. The outcome is then multiplied by a scalar c_{ij}. Importantly, c_{ij} is a value that is fixed, i.e., it is known upfront (before the GNN computation starts), and it does not change across iterations. Example GNN models that fall into the class of C-GNNs are GCN [128] or SGC [219].

In **A-GNNs**, Eq. (7), $\mathbf{h}_j^{(l)}$ is also first transformed with a function ψ. However, the outcome is then multiplied by a value $a\left(\mathbf{h}_i^{(l)}, \mathbf{h}_j^{(l)}\right)$ which is no longer precomputed. Instead, a is a parameterized function of the model and is obtained during training. Importantly, while the derivation of this weight can be arbitrarily complex (e.g., the weight can be computed via attention), the edge weight itself is a scalar. Overall, A-GNNs come with richer expressive capabilities and can be used with models such as GAT [202] or Gated Attention Network (GaAN) [239].

Finally, **MP-GNNs** are the most complex LC class, with edge weights ψ that can be arbitrary vectors or even more complex objects, as specified by ψ. Example MP-GNN models are G-GCN [47] or different models under the umbrella of the Message Passing paradigm (MPNN) [91].

Explicit division of GNNs models into three classes (C-GNNs, A-GNNs, MP-GNNs) has its benefits, as it makes it easier to apply various optimizations. For example, whenever handling a C-GNN model, one can simply precompute edge weights instead of deriving them in each GNN layer.

We illustrate generic work and depth equations in Figures 17-18. Overall, work is the sum of any preprocessing costs W_{pre}, post-processing costs W_{post}, and work of a single GNN layer W_l times the number of layers L. In the MP-GNN generic formulation, W_l equals to work needed to evaluate ψ for each edge (mW_ψ), $\bigoplus$ for each vertex ($nW_\oplus$), and ϕ for each vertex (nW_ϕ). In C-GNNs, the preprocessing cost belongs to W_{pre} and D_{pre}. In A-GNNs, one needs to also consider the work and depth needed to obtain the coefficients a. Depth is analogous, with the main difference that the depth of a single GNN layer is a plain sum of depths of computing ψ, $\bigoplus$, and ϕ (each function is evaluated in parallel for each vertex and edge, hence no multiplication with n or m).

$$\begin{aligned}
\text{Generic} &= W_{pre} + LW_l + W_{post} \\
\text{MP-GNNs} &= W_{pre} + L \cdot \left(mW_\psi + nW_\oplus + nW_\phi\right) + W_{post} \\
\text{C-GNN} &= W_{pre} + L \cdot \left(mW_\psi + nW_\oplus + nW_\phi\right) + W_{post} \\
\text{A-GNNs} &= W_{pre} + L \cdot \left(mW_\psi + mW_a + nW_\oplus + nW_\phi\right) + W_{post}
\end{aligned}$$

Fig. 17: Generic equations for work in GNN model classes.

$$\begin{aligned}
\text{Generic} &= D_{pre} + LD_l + D_{post} \\
\text{MP-GNNs} &= D_{pre} + L \cdot \left(D_\psi + D_\oplus + D_\phi\right) + D_{post} \\
\text{C-GNN} &= D_{pre} + L \cdot \left(D_\psi + D_\oplus + D_\phi\right) + D_{post} \\
\text{A-GNNs} &= D_{pre} + L \cdot \left(D_\psi + D_a + D_\oplus + D_\phi\right) + D_{post}
\end{aligned}$$

Fig. 18: Generic equations for depth in GNN model classes.

6.2 Reductions over H-Hop Neighborhoods

The LC formulations described by Eq. (6)–(8) enable reductions over H-hop neighborhoods, where $H > 1$. This would enable expressing models such as MixHop [2] or SGC [219]. Such H-hop reductions are similar to using polynomial powers of the adjacency matrix in the GL formulation, i.e., they also enable using the knowledge from regions of the graph located further than the direct 1-hop neighbors, within one GNN layer. In terms of parallelization, the main difference from reductions over 1-hop neighborhoods is that the number of vertices being reduced may be much larger (up to n), which means that the work and depth of such a reduction become $O(n)$ and $O(\log n)$, respectively.

Simultaneously, such reductions would require large preprocessing costs, i.e., one needs to derive (and maintain) the information of H-hop neighbors for each vertex.

6.3 Imposing Vertex Order

Nearly all considered GNN models assume that $\bigoplus$ is permutation invariant. However, some GNN models, such as PATCHY-SAN [165], impose explicit vertex ordering. In such models, the outcome of $\bigoplus$ would be different based on the order of the input vertices. This still enables parallelism: analogously to the established parallel prefix sum problem [40], one could compute such a $\bigoplus$ in $O(\log d)$ depth and $O(d)$ work (for d input vertices) – assuming that $\bigoplus$ is associative. However, there may be constant overhead of $2\times$ for both depth and work, compared to the case where $\bigoplus$ is permutation-invariant.

6.4 Heterogeneous GNNs

Heterogeneous graphs (HetG) [32], [53], [184], [211], [229] generalize simple graphs defined as a tuple (V, E) (cf. Section 2) in that vertices and edges may have arbitrary *types* (e.g., person, paper, journal) and *attributes* (e.g., age, page count, ranking). There has been research into HetG learning, with recent models such as Heterogeneous Graph Neural Network [236], Heterogeneous Graph Attention Network [212], or Heterogeneous Graph Transformer [112]. Computations on such models can benefit from forms of parallelism discussed in this work. However, in addition to this, they also come with potential for novel forms of parallelism. For example, in HetG, one often uses different adjacency matrices to model edges of different types. Such matrices could be processed in parallel, introducing **type parallelism**. Similarly, different attributes can also be processed in parallel, offering **attribute parallelism**.

6.5 Dynamic and Temporal GNNs

Another line of works addresses dynamic graph neural networks [223] with potential for **snapshot parallelism**. Specifically, one may parallelize the processing of different *snapshots* of a graph, taken at different time points [56]. Such snapshots can be processed with both the FG and the GL approach. This introduces potential for new optimizations, such as using different approaches with different snapshots based on their sparsity properties.

6.6 Hierarchical GNNs

Some GNN models are *hierarchical*, i.e., one deals with different input granularities in the same training and inference process. For example, in the SEAL-C and SEAL-AI models [142], one is primarily interested in graph classification. However, before running training and inference using graphs as data samples, the authors first execute GNNs separately on each graph, using nodes as data samples. In this process, each graph obtains an embedding vector, that is then used as input for the training process in graph classification. Such hierarchical approach comes with opportunities for more elaborate forms of parallelism, for example parallel pipelining of stages of computations belonging to the different hierarchy levels.

6.7 Spectral GNN Models

GL formulations of GNNs can be *spatial* or *spectral*; the difference is in using either the adjacency or the Laplacian matrix. As Chen et al. [63] shows, one can transform a spatial formulation into a spectral one, and vice versa. For simplicity, we focus on spacial formulations.

6.8 Preprocessing in GNNs

There are different forms of preprocessing in GNNs. First, different GNN models often require **preprocessing the adjacency matrix** and the degree matrix: by *incorporating self-loops* ($\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}, \widetilde{\mathbf{D}} = \mathbf{D} + \mathbf{I}$), with *symmetric normalization* ($\mathbf{A}' = \widetilde{\mathbf{D}}^{-\frac{1}{2}}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-\frac{1}{2}}$), or with *random-walk normalization* ($\overline{\mathbf{A}} = \mathbf{D}^{-1}\mathbf{A}$). Second, some spectral GNN models often require a **spectral decomposition** of the Laplacian matrix [63]. Third, many GNN works propose to reduce or even eliminate using several GNN layers, and instead **push the bulk of computation to preprocessing**. This could involve explicitly considering *multihop neighborhoods* [2], [80] or *polynomial* as well as *rational powers of the adjacency matrix* [63], [80], [219]. All these preprocessing routines come with potential for parallel and distributed execution.

6.9 Global vs. Local Approach for GNNs

On one hand, there exist many GNN models defined using the GL approach, that cannot be easily expressed with the LC formulation. Examples are numerous GNN models that use rational powers of the adjacency or Laplacian matrix, such as Auto-Regress [17], [250], [256], PPNP [43], [129], [230], ARMA [38], ParWalks [221], or RationalNet [63]. On the other hand, many GNN models that were defined with the LC approach, have no known GL formulations, for example GAT [202], MoNet [158], G-GCN [47], or EdgeConv [216].

7 Selected Insights

We now summarize our insights into parallel and distributed GNNs, pointing to parts with more details.

- **GNNs come with new forms of parallelism** Graph partition parallelism (Section 3), closely related to the graph partitioning problem in graph processing, is more challenging than equivalent forms of parallelism in traditional deep learning, due to the dependencies between data samples. Another form, characteristic to GNNs, is graph structure parallelism (§ 4.1.4, § 5.3).
- **GNNs come with rich diversity of tensor operations** Different GNN models use a large variety of tensor operations. While today's works focus on the C-GNN style of computations, that only uses two variants of matrix products, there are numerous others, listed in Table 3 and assigned to models in Tables 5, 6, and 7.
- **Even local GNN formulations heavily use tensor operations** One could conclude that efficient tensor operations are crucial only for the global GNN formulations, as these formulations focus on expressing the whole GNN model with matrices and operations on matrices (cf. Table 7). However, many local GNN formulations have complex tensor operations associated with UpdateEdge or UpdateVertex operators, see Tables 5 and 6.
- **Both local and global GNN formulations are important** There are many GNN models formulated using the local

approach, that have no known global formulation, and vice versa. Thus, effective parallel and distributed execution is important in both formulations.

- **Local and global GNN formulations welcome different optimizations** While having similar, or often the same, work and depth, local and global GNN formulations have potential for different types of optimizations. For example, global GNN models focus on operations on large matrices, which immediately suggests optimizations related to – for example – communication-avoiding linear algebra [136], [188], [189]. On the other hand, local GNN models use operators as the "first class citizens", suggesting that an example important line of work would be operator parallelism such as the one proposed by the Galois framework [133].

8 Challenges & Opportunities

Many of the considered parts of the parallel and distributed GNN landscape were not thoroughly researched. Some were not researched at all. We now list such challenges and opportunities for future research.

- **Efficient GNN inference** Most GNN frameworks focus on training, but fewer of them also target inference. There is a large potential for developing high-performance schemes targeting fast inference.
- **Advanced mini-batching in GNNs** There is very little work on advanced mini-batch training and GNN layer pipelining. Mini-batch training of GNNs is by nature complex, due to the dependencies between node, edge, or graph samples. While the traditional deep learning saw numerous interesting works such as GPipe [116], PipeDream [164], or Chimera [144], that investigate mechanisms such as asynchronous or bi-directional mini-batching, such works are yet to be developed for GNNs.
- **Asynchronous GNNs** The landscape of asynchronous GNNs is largely unexplored. While we outline the overall framework defined in Eq. (2), (3), (4), and (5), implementations, optimizations, and analyses are missing. There could be a plethora of works improving the GNN performance by incorporating different forms of asynchrony.
- **More performance in GNNs via replication** Many GNN works have explored graph partitioning. However, very few (e.g., CAGNET [199]) uses replication for more performance (e.g., through 2.5D & -3D matrix multiplications).
- **Parallelization of GNN models beyond simple C-GNNs** There is very little work on parallel and distributed GNN models beyond the simple seminal ones from the C-GNN or A-GNN classes, such as GCN [128], GAT [202], or GraphSAGE [101]. One would welcome works on more complex models from the MP-GNN class, cf. Tables 5-6.
- **Parallelization of GNN models beyond linear ones** Virtually no research exists on parallel and distributed GNN models of polynomial and rational types, cf. Table 7.
- **Parallelization of other GNN settings** Besides very few attempts [255], there is no work on parallelizing heterogeneous GNNs [236], dynamic and temporal GNNs [223], or hierarchical GNNs [142]. We predict that parallel and distributed schemes targeting these works will come with a large number of research opportunities, due to the rich diversity of these GNN models and the associated graph related problems. One example idea would be to use the available techniques from dynamic and streaming graph processing [22] for GNNs.
- **Achieving large scales** A large challenge is to further push the scale of GNN computations. When comparing the scale and parameter counts of models such as CNNs or Transformers with GNNs, it can be seen that there is a large gap and a lot of research opportunities.
- **New forms of parallelism** It would be interesting to investigate effective utilization of other forms of parallelism in GNNs, for example using Mixture of Experts [154].
- **Incorporating new HW architectures** While some initial works exist, there are not many designs on using GNNs with architectures such as FPGAs [35], [71], transactional processing [27], or processing in memory [4], [29], [88], [93], [98], [162], [163], [166], [182], [183].
- **Incorporating high-performance distributed-memory capabilities** CAGNET [199] illustrated how to scalably execute GNN training across many compute nodes. It would be interesting to push this direction and use high-performance distributed-memory developments and interconnects, and the associated mechanisms for more performance of distributed-memory GNN computations, using – for example – RDMA and RMA programming [87], [179], SmartNICs [28], [74], serverless computing [67], of high-performance networking architectures [20], [24], [26], [34]. Such techniques have been successfully used to accelerate the related graph processing field [191].
- **Incorporating techniques from graph processing** There is more potential in using graph processing techniques for GNNs. While many such schemes have already been incorporated, there are many more to be tried, for example sketching and sampling [21], [37], [90] or various forms of approximate graph processing [8], [19], [44], [45], [59], [76], [78], [83], [173]–[175], [186].
- **Incorporating techniques from linear algebra computations** A lot of work has been done into optimization algebraic operations such as matrix products [86], [92], [134]–[136], [188], [190]. Many of them could be applied in the GNN setting, especially in the context of GL GNN formulations.

9 Conclusion

Graph neural networks (GNNs) are one of the most important and fastest growing parts of machine learning. Parallel and distributed execution of GNNs is a key to GNNs achieving large scales, high performance, and possibly accuracy. In this work, we conduct an in-depth analysis of parallelism and distribution in GNNs. We provide a taxonomy of parallelism, use it to analyze a large number of GNN models, and synthesise the outcomes in a set of insights as well as opportunities for future research. Our work will propel the development of next-generation GNN computations.

References

[1] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *ACM CSUR*, 2021.

[2] S. Abu-El-Haija, B. Perozzi, A. Kapoor, N. Alipourfard, K. Lerman, H. Harutyunyan, G. Ver Steeg, and A. Galstyan. Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *international conference on machine learning*, pages 21–29. PMLR, 2019.

[3] T. Ahmad, L. Jin, X. Zhang, L. Lin, and G. Tang. Graph convolutional neural network for action recognition: A comprehensive survey. *IEEE TAI*, 2021.

[4] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *ISCA*, 2015.

[5] S. Arora. A survey on graph neural networks for knowledge graph completion. *arXiv preprint arXiv:2007.12374*, 2020.

[6] J. Atwood and D. Towsley. Diffusion-convolutional neural networks. In *Advances in neural information processing systems*, pages 1993–2001, 2016.

[7] A. Auten, M. Tomei, and R. Kumar. Hardware acceleration of graph neural networks. In *ACM/IEEE DAC*, 2020.

[8] D. A. Bader et al. Approximating betweenness centrality. In *Algorithms and Models for the Web-Graph*, pages 124–137. Springer, 2007.

[9] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. *Graph partitioning and graph clustering*, volume 588. American Mathematical Society Providence, RI, 2013.

[10] Y. Bai, C. Li, Z. Lin, Y. Wu, Y. Miao, Y. Liu, and Y. Xu. Efficient data loader for fast sampling-based gnn training on large graphs. *IEEE TPDS*, 2021.

[11] M. Balcilar, G. Renton, P. Héroux, B. Gauzere, S. Adam, and P. Honeine. Bridging the gap between spectral and spatial domains in graph neural networks. *arXiv preprint arXiv:2003.11702*, 2020.

[12] M. F. Balın et al. Mg-gcn: Scalable multi-gpu gcn training framework. *arXiv preprint arXiv:2110.08688*, 2021.

[13] T. Baruah, K. Shivdikar, S. Dong, Y. Sun, S. A. Mojumder, K. Jung, J. L. Abellán, Y. Ukidave, A. Joshi, J. Kim, et al. Gnnmark: A benchmark suite to characterize graph neural network training on gpus. In *IEEE ISPASS*, 2021.

[14] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[15] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler. A modular benchmarking infrastructure for high-performance and reproducible deep learning. In *IEEE IPDPS*, 2019.

[16] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM CSUR*, 2019.

[17] Y. Bengio, O. Delalleau, and N. Le Roux. 11 label propagation and quadratic criterion. 2006.

[18] M. Besta. *Enabling High-Performance Large-Scale Irregular Computations*. PhD thesis, ETH Zurich, 2021.

[19] M. Besta, A. Carigiet, Z. Vonarburg-Shmaria, K. Janda, L. Gianinazzi, and T. Hoefler. High-performance parallel graph coloring with strong guarantees on work, depth, and quality. In *ACM/IEEE Supercomputing*, 2020.

[20] M. Besta, J. Domke, M. Schneider, M. Konieczny, S. Di Girolamo, T. Schneider, A. Singla, and T. Hoefler. High-performance routing with multipathing and path diversity in ethernet and hpc networks. *IEEE Transactions on Parallel and Distributed Systems*, 32(4):943–959, 2020.

[21] M. Besta, M. Fischer, T. Ben-Nun, D. Stanojevic, J. D. F. Licht, and T. Hoefler. Substream-centric maximum matchings on fpga. *ACM TRETS*, 2020.

[22] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *IEEE TPDS*, 2020.

[23] M. Besta, R. Grob, C. Miglioli, N. Bernold, G. Kwasniewski, G. Gjini, R. Kanakagiri, S. Ashkboos, L. Gianinazzi, N. Dryden, et al. Motif prediction with graph neural networks. In *ACM KDD*, 2022.

[24] M. Besta, S. M. Hassan, S. Yalamanchili, R. Ausavarungnirun, O. Mutlu, and T. Hoefler. Slim noc: A low-diameter on-chip network topology for high energy efficiency and scalability. In *ACM ASPLOS*, 2018.

[25] M. Besta and T. Hoefler. Fault tolerance for remote memory access programming models. In *ACM HPDC*, 2014.

[26] M. Besta and T. Hoefler. Slim fly: A cost effective low-diameter network topology. In *ACM/IEEE Supercomputing*, 2014.

[27] M. Besta and T. Hoefler. Accelerating irregular computations with hardware transactional memory and active messages. In *ACM HPDC*, 2015.

[28] M. Besta and T. Hoefler. Active access: A mechanism for high-performance distributed data-centric computations. In *ACM ICS*, 2015.

[29] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, et al. Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems. In *IEEE/ACM MICRO*, 2021.

[30] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik. Communication-efficient jaccard similarity for high-performance distributed genome comparisons. In *IEEE IPDPS*, 2020.

[31] M. Besta, F. Marending, E. Solomonik, and T. Hoefler. Slimsell: A vectorizable graph representation for breadth-first search. In *IEEE IPDPS*, 2017.

[32] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *arXiv preprint arXiv:1910.09017*, 2019.

[33] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *ACM HPDC*, 2017.

[34] M. Besta, M. Schneider, K. Cynk, M. Konieczny, E. Henriksson, S. Di Girolamo, A. Singla, and T. Hoefler. Fatpaths: Routing in supercomputers and data centers when shortest paths fall short. *ACM/IEEE Supercomputing*, 2020.

[35] M. Besta, D. Stanojevic, J. D. F. Licht, T. Ben-Nun, and T. Hoefler. Graph processing on FPGAs: Taxonomy, survey, challenges. *arXiv preprint arXiv:1903.06697*, 2019.

[36] M. Besta, Z. Vonarburg-Shmaria, Y. Schaffner, L. Schwarz, G. Kwasniewski, L. Gianinazzi, J. Beranek, K. Janda, T. Holenstein, S. Leisinger, et al. Graphminesuite: Enabling high-performance and programmable graph mining algorithms with set algebra. *VLDB*, 2021.

[37] M. Besta, S. Weber, L. Gianinazzi, R. Gerstenberger, A. Ivanov, Y. Oltchik, and T. Hoefler. Slim graph: Practical lossy graph compression for approximate graph processing, storage, and analytics. In *ACM/IEEE Supercomputing*, 2019.

[38] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi. Graph neural networks with convolutional arma filters. *IEEE TPAMI*, 2021.

[39] G. Bilardi and A. Pietracaprina. *Models of Computation, Theoretical*. Springer US, 2011.

[40] G. E. Blelloch. Pre x sums and their applications. Technical report, Citeseer, 1990.

[41] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[42] G. E. Blelloch and B. M. Maggs. *Parallel Algorithms*. 2010.

[43] A. Bojchevski, J. Klicpera, B. Perozzi, A. Kapoor, M. Blais, B. Rózemberczki, M. Lukasik, and S. Günnemann. Scaling graph neural networks with approximate pagerank. In *ACM KDD*, 2020.

[44] P. Boldi, M. Rosa, and S. Vigna. Hyperanf: Approximating the neighbourhood function of very large graphs on a budget. In *Proceedings of the 20th international conference on World wide web*, pages 625–634. ACM, 2011.

[45] M. Borassi and E. Natale. Kadabra is an adaptive algorithm for betweenness via random approximation. *arXiv preprint arXiv:1604.08553*, 2016.

[46] K. M. Borgwardt. *Graph kernels*. PhD thesis, lmu, 2007.

[47] X. Bresson and T. Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

[48] M. M. Bronstein, J. Bruna, T. Cohen, and P. Veličković. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *arXiv preprint arXiv:2104.13478*, 2021.

[49] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[50] A. Buluç and E. G. Boman. Towards scalable parallel hypergraph partitioning. 2008.

[51] A. Buluc and K. Madduri. Graph partitioning for scalable distributed graph computations. *Graph Partitioning and Graph Clustering*, 588:83, 2013.

[52] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.

[53] H. Cai, V. W. Zheng, and K. C.-C. Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications.

IEEE Transactions on Knowledge and Data Engineering, 30(9):1616–1637, 2018.

[54] 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.

[55] Ü. V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. In *IPDPS*, 2001.

[56] V. T. Chakaravarthy, S. S. Pandian, S. Raje, Y. Sabharwal, T. Suzumura, and S. Ubaru. Efficient scaling of dynamic graph neural networks. In *ACM/IEEE Supercomputing*, 2021.

[57] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM computing surveys (CSUR)*, 38(1):2, 2006.

[58] I. Chami, S. Abu-El-Haija, B. Perozzi, C. Ré, and K. Murphy. Machine learning on graphs: A model and comprehensive taxonomy. *arXiv preprint arXiv:2005.03675*, 2020.

[59] B. Chazelle, R. Rubinfeld, and L. Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM Journal on computing*, 34(6):1370–1379, 2005.

[60] J. Chen et al. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568*, 2017.

[61] J. Chen et al. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv:1801.10247*, 2018.

[62] X. Chen, Y. Wang, X. Xie, X. Hu, A. Basak, L. Liang, M. Yan, L. Deng, Y. Ding, Z. Du, et al. Rubik: A hierarchical architecture for efficient graph neural network training. *IEEE TCAD*, 2021.

[63] Z. Chen, F. Chen, L. Zhang, T. Ji, K. Fu, L. Zhao, F. Chen, L. Wu, C. Aggarwal, and C.-T. Lu. Bridging the gap between spatial and spectral domains: A unified framework for graph neural networks. *arXiv preprint arXiv:2107.10234*, 2021.

[64] Z. Chen, M. Yan, M. Zhu, L. Deng, G. Li, S. Li, and Y. Xie. fusegnn: accelerating graph convolutional neural network training on gpgpu. In *IEEE/ACM ICCAD*, 2020.

[65] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *ACM KDD*, 2019.

[66] D. J. Cook and L. B. Holder. *Mining graph data*. John Wiley & Sons, 2006.

[67] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *ACM Middleware*, 2021.

[68] P. Cui, X. Wang, J. Pei, and W. Zhu. A survey on network embedding. *IEEE Transactions on Knowledge and Data Engineering*, 31(5):833–852, 2018.

[69] Y. Dai, S. Wang, N. N. Xiong, and W. Guo. A survey on knowledge graph embedding: Approaches, applications and benchmarks. *Electronics*, 9(5):750, 2020.

[70] A. Davies, P. Veličković, L. Buesing, S. Blackwell, D. Zheng, N. Tomašev, R. Tanburn, P. Battaglia, C. Blundell, A. Juhász, et al. Advancing mathematics by guiding human intuition with ai. *Nature*, 600(7887):70–74, 2021.

[71] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE TPDS*, 2021.

[72] M. Defferrard et al. Convolutional neural networks on graphs with fast localized spectral filtering. *NeurIPS*, 2016.

[73] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *IEEE IPDPS*, 2006.

[74] S. Di Girolamo, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, J. Beránek, M. Besta, L. Benini, D. Roweth, and T. Hoefler. Network-accelerated non-contiguous memory transfers. In *ACM/IEEE Supercomputing*, 2019.

[75] C. H. Ding, X. He, H. Zha, M. Gu, and H. D. Simon. A min-max cut algorithm for graph partitioning and data clustering. In *Proceedings 2001 IEEE international conference on data mining*, pages 107–114. IEEE, 2001.

[76] S. Dumbrava, A. Bonifati, A. N. R. Diaz, and R. Vuillemot. Approximate evaluation of label-constrained reachability queries. *arXiv preprint arXiv:1811.11561*, 2018.

[77] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

[78] G. ECHBARTHI and H. KHEDDOUCI. Lasas: an aggregated search based graph matching approach. In *The 29th International Conference on Software Engineering and Knowledge Engineering*, 2017.

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

[80] F. Frasca, E. Rossi, D. Eynard, B. Chamberlain, M. Bronstein, and F. Monti. Sign: Scalable inception graph neural networks. *arXiv preprint arXiv:2004.11198*, 2020.

[81] S. Gandhi and A. P. Iyer. P3: Distributed deep graph learning at scale. In *USENIX OSDI*, 2021.

[82] R. Garg, E. Qin, F. Muñoz-Martínez, R. Guirado, A. Jain, S. Abadal, J. L. Abellán, M. E. Acacio, E. Alarcón, S. Rajamanickam, et al. Understanding the design space of sparse/dense multiphase dataflows for mapping graph neural networks on spatial accelerators. *arXiv preprint arXiv:2103.07977*, 2021.

[83] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 90–100. Society for Industrial and Applied Mathematics, 2008.

[84] B. Geissmann and L. Gianinazzi. Parallel minimum cuts in near-linear work and low depth. *arXiv preprint arXiv:1807.09524*, 2018.

[85] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *IEEE/ACM MICRO*, 2020.

[86] E. Georganas, J. Gonzalez-Dominguez, E. Solomonik, Y. Zheng, J. Tourino, and K. Yelick. Communication Avoiding and Overlapping for Numerical Linear Algebra. In *ACM/IEEE Supercomputing*, 2012.

[87] R. Gerstenberger et al. Enabling highly-scalable remote memory access programming with mpi-3 one sided. In *ACM/IEEE Supercomputing*, 2013.

[88] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu. The processing-in-memory paradigm: Mechanisms to enable adoption. In *Beyond-CMOS Technologies for Next Generation Computer Design*, pages 133–194. Springer, 2019.

[89] L. Gianinazzi, M. Fries, N. Dryden, T. Ben-Nun, and T. Hoefler. Learning combinatorial node labeling algorithms. *arXiv preprint arXiv:2106.03594*, 2021.

[90] L. Gianinazzi, P. Kalvoda, A. De Palma, M. Besta, and T. Hoefler. Communication-avoiding parallel minimum cuts and connected components. *ACM SIGPLAN Notices*, 2018.

[91] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272. PMLR, 2017.

[92] N. Gleinig, M. Besta, and T. Hoefler. I/o-optimal cache-oblivious sparse matrix-sparse matrix multiplication. 2022.

[93] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. *arXiv preprint arXiv:2105.03814*, 2021.

[94] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[95] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[96] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. *POOSC*, 2005.

[97] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *ACM KDD*, 2016.

[98] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu. Simdram: a framework for bit-serial simd processing using dram. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–345, 2021.

[99] W. L. Hamilton. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning*, 14(3):1–159, 2020.

[100] W. L. Hamilton et al. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[101] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, 2017.

[102] C. He, E. Ceyani, K. Balasubramanian, M. Annavaram, and S. Avestimehr. Spreadgnn: Serverless multi-task federated learning for graph neural networks. *arXiv preprint arXiv:2106.02743*, 2021.

[103] Y. He, Y. Wang, C. Liu, H. Li, and X. Li. Tare: Task-adaptive in-situ reram computing for graph learning. In *ACM/IEEE DAC*, 2021.

[104] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.

[105] L. Hoang, X. Chen, H. Lee, R. Dathathri, G. Gill, and K. Pingali. Efficient distribution for deep learning on large graphs, 2021.

[106] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell. spin: High-performance streaming processing in the network. In *ACM/IEEE Supercomputing*, 2017.

[107] T. Hoefler and D. Moor. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Supercomputing frontiers and innovations*, 1(2):58–75, 2014.

[108] T. Horváth, T. Gärtner, and S. Wrobel. Cyclic pattern kernels for predictive graph mining. In *ACM KDD*, 2004.

[109] F. Hu, L. Wang, S. Wu, L. Wang, and T. Tan. Graph classification by mixture of diverse experts. *arXiv preprint arXiv:2103.15622*, 2021.

[110] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[111] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang. Featgraph: A flexible and efficient backend for graph neural network systems. In *ACM/IEEE Supercomputing*, 2020.

[112] Z. Hu, Y. Dong, K. Wang, and Y. Sun. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*, pages 2704–2710, 2020.

[113] G. Huang, G. Dai, Y. Wang, and H. Yang. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *ACM/IEEE Supercomputing*, 2020.

[114] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen. Understanding and bridging the gaps in current gnn performance optimizations. In *ACM PPoPP*, 2021.

[115] L. Huang, D. Ma, S. Li, X. Zhang, and H. Wang. Text level graph neural network for text classification. *arXiv preprint arXiv:1910.02356*, 2019.

[116] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *NeurIPS*, 2019.

[117] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. *MLSys*, 2020.

[118] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(1):75–105, 2013.

[119] W. Jiang and J. Luo. Graph neural network for traffic forecasting: A survey. *arXiv preprint arXiv:2101.11174*, 2021.

[120] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[121] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Supercomputing'95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, pages 29–29. IEEE, 1995.

[122] G. Karypis and V. Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[123] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, and H. Meyerhenke. Mathematical foundations of the graphblas. In *IEEE HPEC*, 2016.

[124] M. Kim and K. S. Candan. Sbv-cut: Vertex-cut based graph partitioning using structural balance vertices. *Data & Knowledge Engineering*, 72:285–303, 2012.

[125] T. Kim, C. Hwang, K. Park, Z. Lin, P. Cheng, Y. Miao, L. Ma, and Y. Xiong. Accelerating gnn training with locality-aware partial execution. In *ACM SIGOPS APSys*, 2021.

[126] K. Kiningham, P. Levis, and C. Ré. Greta: Hardware optimized graph processing for gnns. In *ReCoML 2020*, 2020.

[127] K. Kiningham, C. Re, and P. Levis. Grip: a graph neural network accelerator architecture. *arXiv preprint arXiv:2007.13828*, 2020.

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

[129] J. Klicpera et al. Predict then propagate: Graph neural networks meet personalized pagerank. *arXiv preprint arXiv:1810.05997*, 2018.

[130] J. Klicpera, S. Weißenberger, and S. Günnemann. Diffusion improves graph learning. *NeurIPS*, 2019.

[131] N. M. Kriege, F. D. Johansson, and C. Morris. A survey on graph kernels. *Applied Network Science*, 5(1):1–42, 2020.

[132] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[133] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *ACM SIGPLAN Notices*, 42(6):211–222, 2007.

[134] G. Kwasniewski, T. Ben-Nun, L. Gianinazzi, A. Calotoiu, T. Schneider, A. N. Ziogas, M. Besta, and T. Hoefler. Pebbles, graphs, and a pinch of combinatorics: Towards tight i/o lower bounds for statically analyzable programs. *ACM SPAA*, 2021.

[135] G. Kwasniewski, T. Ben-Nun, A. N. Ziogas, T. Schneider, M. Besta, and T. Hoefler. On the parallel i/o optimality of linear algebra kernels: near-optimal lu factorization. In *ACM PPoPP*, 2021.

[136] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *ACM/IEEE Supercomputing*, 2019.

[137] L. C. Lamb, A. Garcez, M. Gori, M. Prates, P. Avelar, and M. Vardi. Graph neural networks meet neural-symbolic computing: A survey and perspective. *arXiv preprint arXiv:2003.00330*, 2020.

[138] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[139] G. Li, M. Müller, B. Ghanem, and V. Koltun. Training graph neural networks with 1000 layers. In *International conference on machine learning*, pages 6437–6449. PMLR, 2021.

[140] H. Li, M. Yan, X. Yang, L. Deng, W. Li, X. Ye, D. Fan, and Y. Xie. Hardware acceleration for gcns via bidirectional fusion. *IEEE Computer Architecture Letters*, 20(1):66–4, 2021.

[141] J. Li, A. Louri, A. Karanth, and R. Bunescu. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *IEEE HPCA*, 2021.

[142] J. Li, Y. Rong, H. Cheng, H. Meng, W. Huang, and J. Huang. Semi-supervised graph classification: A hierarchical graph perspective. In *The World Wide Web Conference*, pages 972–982, 2019.

[143] S. Li, T. Ben-Nun, S. D. Girolamo, D. Alistarh, and T. Hoefler. Taming unbalanced training workloads in deep learning with partial collective operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 45–61, 2020.

[144] S. Li and T. Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *ACM/IEEE Supercomputing*, 2021.

[145] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE TOC*, 2020.

[146] C.-Y. Lin, L. Luo, and L. Ceze. Accelerating spmm kernel with cache-first edge sampling for gnn inference. *arXiv preprint arXiv:2104.10716*, 2021.

[147] H. Lin et al. Shentu: processing multi-trillion edge graphs on millions of cores in seconds. In *ACM/IEEE Supercomputing*, 2018.

[148] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu. Learning entity and relation embeddings for knowledge graph completion. In *AAAI*, 2015.

[149] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *ACM SoCC*, 2020.

[150] H. Liu, S. Lu, X. Chen, and B. He. G3: when graph neural networks meet parallel graph processing systems on gpus. *VLDB*, 2020.

[151] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541*, 2021.

[152] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[153] 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*, 2019.

[154] S. Masoudnia and R. Ebrahimpour. Mixture of experts: a literature survey. *Artificial Intelligence Review*, 42(2):275–293, 2014.

[155] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha. Distgnn: Scalable distributed training for large-scale graph neural networks. *arXiv preprint arXiv:2104.06700*, 2021.

[156] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu. Large graph convolutional network training with gpu-oriented data communication architecture. *arXiv preprint arXiv:2103.03330*, 2021.

[157] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[158] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *IEEE CVPR*, 2017.

[159] H. Mostafa. Sequential aggregation and rematerialization: Distributed full-batch training of graph neural networks on large graphs. *arXiv preprint arXiv:2111.06483*, 2021.

[160] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3, Sept. 2012. available at: http://www.mpi-forum.org (Sep. 2012).

[161] K. Mulmuley. A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. In *ACM STOC*, 1986.

[162] O. Mutlu et al. Processing Data Where It Makes Sense: Enabling In-Memory Computation. *MicPro*, 2019.

[163] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun. A modern primer on processing in memory. *arXiv preprint arXiv:2012.03112*, 2020.

[164] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *ACM SOSP*, 2019.

[165] M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. In *ICML*, 2016.

[166] G. F. Oliveira, J. Gómez-Luna, L. Orosa, S. Ghose, N. Vijaykumar, I. Fernandez, M. Sadrosadati, and O. Mutlu. Damov: A new methodology and benchmark suite for evaluating data movement bottlenecks. *arXiv preprint arXiv:2105.03725*, 2021.

[167] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[168] B. Perozzi et al. Deepwalk: Online learning of social representations. In *ACM KDD*, 2014.

[169] Petar Veličković. Theoretical foundations of graph neural networks. https://petar-v.com/talks/GNN-Wednesday.pdf, 2021.

[170] T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, and P. W. Battaglia. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409*, 2020.

[171] S. Qiu, Y. Liang, and Z. Wang. Optimizing sparse matrix multiplications for graph neural networks. *arXiv preprint arXiv:2111.00352*, 2021.

[172] M. K. Rahman, M. H. Sujon, and A. Azad. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks. In *IEEE IPDPS*, 2021.

[173] M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. *Data Mining and Knowledge Discovery*, 30(2):438–475, 2016.

[174] M. Riondato and E. Upfal. Abra: Approximating betweenness centrality in static and dynamic graphs with rademacher averages. *ACM TKDD*, 12(5):61, 2018.

[175] L. Roditty and V. Vassilevska Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 515–524. ACM, 2013.

[176] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz, et al. The future is big graphs: a community view on graph processing systems. *Communications of the ACM*, 64(9):62–71, 2021.

[177] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. Battaglia. Learning to simulate complex physics with graph networks. In *ICML*, 2020.

[178] R. Sato. A survey on the expressive power of graph neural networks. *arXiv preprint arXiv:2003.04078*, 2020.

[179] P. Schmid, M. Besta, and T. Hoefler. High-performance distributed rma locks. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 19–30. ACM, 2016.

[180] H. Schweizer et al. Evaluating the cost of atomic operations on modern architectures. In *ACM/IEEE PACT*, 2015.

[181] M. Serafini and H. Guan. Scalable graph neural network training: The case for sampling. *ACM SIGOPS Operating Systems Review*, 55(1):68–76, 2021.

[182] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, and M. A. Kozuch. Rowclone: fast and energy-efficient in-dram bulk data copy and initialization. In *IEEE/ACM MICRO*, 2013.

[183] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *IEEE/ACM MICRO*, 2017.

[184] C. Shi, Y. Li, J. Zhang, Y. Sun, and S. Y. Philip. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering*, 29(1):17–37, 2016.

[185] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua. Graph processing on gpus: A survey. *ACM Computing Surveys (CSUR)*, 50(6):1–35, 2018.

[186] G. M. Slota and K. Madduri. Complex network analysis using parallel approximate motif counting. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 405–414. IEEE, 2014.

[187] E. Solomonik, M. Besta, F. Vella, and T. Hoefler. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *ACM/IEEE Supercomputing*, 2017.

[188] E. Solomonik, E. Carson, N. Knight, and J. Demmel. Tradeoffs between synchronization, communication, and work in parallel linear algebra computations. Technical report, 2014.

[189] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *Euro-Par*, 2011.

[190] E. Solomonik and T. Hoefler. Sparse Tensor Algebra as a Parallel Programming Model. *CoRR*, abs/1512.00066, 2015.

[191] A. Strausz, F. Vella, S. Di Girolamo, M. Besta, and T. Hoefler. Asynchronous distributed-memory triangle counting and lcc with rma caching. 2022.

[192] S. Sukhbaatar, R. Fergus, et al. Learning multiagent communication with backpropagation. *NeurIPS*, 2016.

[193] S. A. Tailor, R. de Jong, T. Azevedo, M. Mattina, and P. Maji. Towards efficient point cloud graph neural networks through architectural simplification. In *IEEE/CVF ICCV*, 2021.

[194] A. Tate, A. Kamil, A. Dubey, A. Größlinger, B. Chamberlain, B. Goglin, C. Edwards, C. J. Newburn, D. Padua, D. Unat, et al. Programming abstractions for data locality. PADAL Workshop 2014, 2014.

[195] D. A. Tedjopurnomo, Z. Bao, B. Zheng, F. Choudhury, and A. Qin. A survey on modern deep neural network for traffic prediction: Trends, methods and challenges. *IEEE TKDE*, 2020.

[196] K. K. Thekumparampil, C. Wang, S. Oh, and L.-J. Li. Attention-based graph neural network for semi-supervised learning. *arXiv preprint arXiv:1803.03735*, 2018.

[197] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, et al. Dorylus: Affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads. In *USENIX OSDI*, 2021.

[198] C. Tian, L. Ma, Z. Yang, and Y. Dai. Pcgcn: Partition-centric processing for accelerating graph convolutional network. In *IEEE IPDPS*, 2020.

[199] A. Tripathy, K. Yelick, and A. Buluç. Reducing communication in graph neural network training. In *ACM/IEEE Supercomputing*, 2020.

[200] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[201] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

[202] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[203] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11:1201–1242, 2010.

[204] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman. Marius++: Large-scale training of graph neural networks on a single machine. *arXiv preprint arXiv:2202.02365*, 2022.

[205] C. Wan, Y. Li, A. Li, N. S. Kim, and Y. Lin. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-

parallelism and random boundary node sampling sampling. *MLSys*, 2022.

[206] C. Wan, Y. Li, C. R. Wolfe, A. Kyrillidis, N. S. Kim, and Y. Lin. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. *arXiv preprint arXiv:2203.10428*, 2022.

[207] D. Wang, P. Cui, and W. Zhu. Structural deep network embedding. In *ACM KDD*, 2016.

[208] L. Wang, Q. Yin, C. Tian, J. Yang, R. Chen, W. Yu, Z. Yao, and J. Zhou. Flexgraph: a flexible and efficient distributed framework for gnn training. In *EuroSys*, 2021.

[209] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv:1909.01315*, 2019.

[210] Q. Wang, Z. Mao, B. Wang, and L. Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE TKDE*, 2017.

[211] X. Wang, D. Bo, C. Shi, S. Fan, Y. Ye, and P. S. Yu. A survey on heterogeneous graph embedding: methods, techniques, applications and sources. *arXiv:2011.14867*, 2020.

[212] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu. Heterogeneous graph attention network. In *The world wide web conference*, pages 2022–2032, 2019.

[213] Y. Wang et al. Gunrock: A high-performance graph processing library on the GPU. In *ACM PPoPP*, 2016.

[214] Y. Wang, B. Feng, and Y. Ding. Qgtc: Accelerating quantized gnn via gpu tensor core. *arXiv preprint arXiv:2111.09547*, 2021.

[215] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding. Gnnadvisor: An efficient runtime system for gnn acceleration on gpus. *arXiv preprint arXiv:2006.06608*, 2020.

[216] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon. Dynamic graph cnn for learning on point clouds. *Acm Transactions On Graphics (tog)*, 38(5):1–12, 2019.

[217] Z. Wang, Y. Guan, G. Sun, D. Niu, Y. Wang, H. Zheng, and Y. Han. Gnn-pim: A processing-in-memory architecture for graph neural networks. In *Springer ACA*, 2020.

[218] Z. Wang, Y. Wang, C. Yuan, R. Gu, and Y. Huang. Empirical analysis of performance bottlenecks in graph neural network training and inference with gpus. *Neurocomputing*, 2021.

[219] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger. Simplifying graph convolutional networks. In *International conference on machine learning*, pages 6861–6871. PMLR, 2019.

[220] S. Wu, F. Sun, W. Zhang, and B. Cui. Graph neural networks in recommender systems: a survey. *arXiv preprint arXiv:2011.02260*, 2020.

[221] X.-M. Wu, Z. Li, A. M.-C. So, J. Wright, and S.-F. Chang. Learning with partially absorbing random walks. In *NIPS*, volume 25, pages 3077–3085, 2012.

[222] 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.

[223] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE TNNLS*, 2020.

[224] Y. Xie, B. Yu, S. Lv, C. Zhang, G. Wang, and M. Gong. A survey on heterogeneous network representation learning. *Pattern Recognition*, 116:107936, 2021.

[225] Z. Xinyi and L. Chen. Capsule graph neural network. In *International conference on learning representations*, 2018.

[226] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[227] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. Characterizing and understanding gcns on gpu. *IEEE Computer Architecture Letters*, 19(1):22–25, 2020.

[228] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. Hygcn: A gcn accelerator with hybrid architecture. In *IEEE HPCA*, 2020.

[229] C. Yang, Y. Xiao, Y. Zhang, Y. Sun, and J. Han. Heterogeneous network representation learning: A unified framework with survey and benchmark. *IEEE TKDE*, 2020.

[230] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *ACM KDD*, 2018.

[231] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling. *arXiv preprint arXiv:1806.08804*, 2018.

[232] H. Yuan, H. Yu, S. Gui, and S. Ji. Explainability in graph neural networks: A taxonomic survey. *arXiv preprint arXiv:2012.15445*, 2020.

[233] H. Zeng and V. Prasanna. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In *ACM/SIGDA FPGA*, 2020.

[234] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

[235] B. Zhang, H. Zeng, and V. Prasanna. Hardware acceleration of large scale gcn inference. In *IEEE ASAP*, 2020.

[236] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla. Heterogeneous graph neural network. In *ACM KDD*, 2019.

[237] D. Zhang, X. Huang, Z. Liu, Z. Hu, X. Song, Z. Ge, Z. Zhang, L. Wang, J. Zhou, Y. Shuang, et al. Agl: a scalable system for industrial-purpose graph machine learning. *arXiv preprint arXiv:2003.02454*, 2020.

[238] H. Zhang, Z. Yu, G. Dai, G. Huang, Y. Ding, Y. Xie, and Y. Wang. Understanding gnn computational graph: A coordinated computation, io, and memory perspective. *arXiv preprint arXiv:2110.09524*, 2021.

[239] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D.-Y. Yeung. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018.

[240] L. Zhang, Z. Lai, S. Li, Y. Tang, F. Liu, and D. Li. 2pgraph: Accelerating gnn training over large graphs on gpu clusters. In *IEEE CLUSTER*, 2021.

[241] S. Zhang, H. Tong, J. Xu, and R. Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1):1–23, 2019.

[242] W. Zhang, Y. Shen, Z. Lin, Y. Li, X. Li, W. Ouyang, Y. Tao, Z. Yang, and B. Cui. Gmlp: Building scalable and flexible graph neural networks with feature-message passing. *arXiv:2104.09880*, 2021.

[243] W. Zhang, Y. Shen, Z. Lin, Y. Li, X. Li, W. Ouyang, Y. Tao, Z. Yang, and B. Cui. Pasca: A graph neural architecture search system under the scalable paradigm. In *ACM TheWebConf*, 2022.

[244] Z. Zhang, P. Cui, and W. Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[245] Z. Zhang, J. Leng, S. Lu, Y. Miao, Y. Diao, M. Guo, C. Li, and Y. Zhu. Zipper: Exploiting tile-and operator-level parallelism for general and scalable graph neural network acceleration. *arXiv preprint arXiv:2107.08709*, 2021.

[246] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, and M. Guo. Architectural implications of graph neural networks. *IEEE Computer architecture letters*, 19(1):59–62, 2020.

[247] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. Distdgl: distributed graph neural network training for billion-scale graphs. In *IEEE/ACM IA3*, 2020.

[248] D. Zheng, X. Song, C. Yang, D. LaSalle, Q. Su, M. Wang, C. Ma, and G. Karypis. Distributed hybrid cpu and gpu training for graph neural networks on billion-scale graphs. *arXiv:2112.15345*, 2021.

[249] A. Zhou, J. Yang, Y. Gao, T. Qiao, Y. Qi, X. Wang, Y. Chen, P. Dai, W. Zhao, and C. Hu. Optimizing memory efficiency of graph neural networks on edge computing platforms. *arXiv preprint arXiv:2104.03058*, 2021.

[250] D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf. Learning with local and global consistency. In *Advances in neural information processing systems*, pages 321–328, 2004.

[251] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[252] X. Zhou and Y. Luo. Explore mixture of experts in graph neural networks, 2019.

[253] Z. Zhou, C. Li, X. Wei, and G. Sun. Gcnear: A hybrid architecture for efficient gcn training with near-memory processing. *arXiv preprint arXiv:2111.00680*, 2021.

[254] Z. Zhou, B. Shi, Z. Zhang, Y. Guan, G. Sun, and G. Luo. Blockgnn: Towards efficient gnn acceleration using block-circulant weight matrices. In *ACM/IEEE DAC*, 2021.

[255] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou. Aligraph: a comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730*, 2019.

[256] X. Zhu, Z. Ghahramani, and J. D. Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *ICML*, 2003.