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

Maciej Besta and Torsten Hoefler

***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*—Deep learning, parallel processing, parallel algorithms.**

I. Introduction

GRAPH neural networks (GNNs) are taking over the world of machine learning (ML) by storm [1]. They have been used in a plethora of complex problems such as node classification, graph classification, or edge prediction. Example areas of application are social sciences, bioinformatics, chemistry, medicine, cybersecurity, linguistics, transportation, and others [1]. Some recent celebrated success stories are cost-effective and fast placement of high-performance chips [2], guiding mathematical discoveries [3], or significantly improving the accuracy of protein folding prediction [4].

GNNs generalize both *traditional deep learning (DL)* and *graph processing*. Contrarily to the former, they do not operate on regular grids and highly structured data (such as, e.g., image processing); instead, the data is highly unstructured, irregular, and the resulting computations are data-driven and lacking straightforward spatial or temporal locality [5]. 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 [6], 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) [7], 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 [8], 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) [9] 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. 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?*

Manuscript received 23 May 2022; revised 20 July 2023; accepted 27 July 2023. Date of publication 22 February 2024; date of current version 3 April 2024. This project received funding from the European Research Council (Project PSAP, No. 101002047), and the European High-Performance Computing Joint Undertaking (JU) under Grant 955513 (MAELSTROM). This project was supported by the ETH Future Computing Laboratory (EFCL), financed by a donation from Huawei Technologies. This project received funding from the European Union.s HE research and innovation programme under Grant 101070141 (Project GLACIATION). Recommended for acceptance by Y. Sun. *(Corresponding author: Maciej Besta.)*

The authors are with the Department of Computer Science, ETH Zurich, 8092 Zürich, Switzerland (e-mail: maciej.besta@inf.ethz.ch; htor@inf.ethz.ch).

Digital Object Identifier 10.1109/TPAMI.2023.3303431

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 Work-Depth model [10], 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 [7], SGC [11], GAT [8], G-GCN [9]), parts of GNN computations (e.g., inference, training), building blocks of GNNs (e.g., layers, operators/kernels), programming paradigms (e.g., SAGA-NN [12], GReTA [13]), execution schemes behind GNNs (e.g., reduce, activate, different tensor operations), GNN frameworks (e.g., NeuGraph [12]), GNN accelerators (e.g., HyGCN [14]) GNN-driven ML tasks (e.g., node classification, edge prediction), mini-batching versus 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.

Overall, the central contributions of our work are:

- We identify and analyze fundamental forms of parallelism in GNNs, and we illustrate that they – to some degree – match those in traditional DL. This will foster designing future GNN systems more effectively, by empowering system designers with a clear view of the space of parallelization approaches that they could use, and how these approaches can be combined. Moreover, it will facilitate reusing existing large-scale DL frameworks.
- We analyze a broad spectrum of GNN models formally (covering all major classes of models, i.e., Convolution, Attention, Message-Passing, and Linear/Polynomial/Rational ones), for a total of 23 models, investigating how parallelizable they are, and identifying their bottlenecks and the associated tradeoffs. This will facilitate scaling these models to much larger sizes than what is done today. It is an important factor in making them more powerful, as indicated by the recent successes of large NLPs.

TABLE I
MOST IMPORTANT SYMBOLS USED IN THE PAPER

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 $\overline{\mathbf{A}} = \mathbf{D}^{-1}\mathbf{A}$ [131].
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.

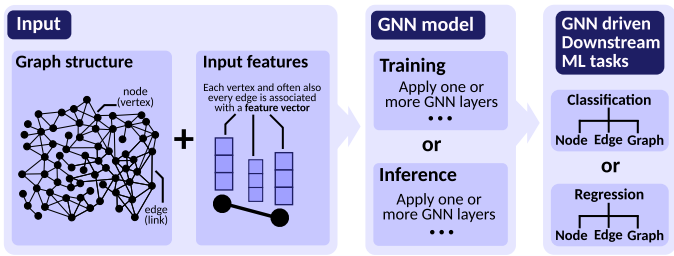


Fig. 1. *(Section II-A) Overview of general GNN computation.* Input comprises the graph structure and the accompanying feature vectors (assigned to vertices/edges). The input is processed using a specific GNN model (training or inference). Output feature vectors are used in various downstream ML tasks.

- We design a broad theoretical framework for asynchronous GNNs, which will serve as a blueprint for novel GNN models and implementations that will further push the scalability and performance of GNNs.
- We review challenges and opportunities, which will facilitate future research into large-scale GNNs.

II. GRAPH NEURAL NETWORKS: OVERVIEW

We first overview GNNs; Table I provides notation.

A. GNN Computation: A High-Level Summary

We overview a *GNN computation* in Fig. 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 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

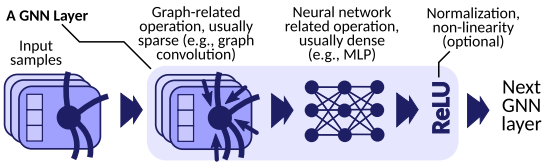


Fig. 2. *(Section II-A) Overview of one GNN layer.* The input samples (e.g., vertices or graphs) are processed with a graph-related operation such as graph convolution, followed by a neural network related operation such as an MLP, then optionally by a non-linearity such as ReLU, and potentially by some normalization.

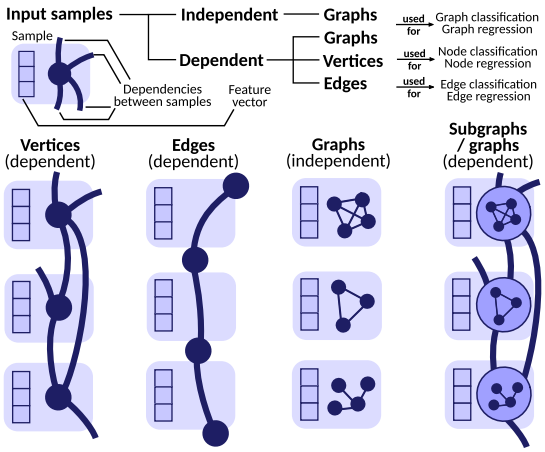


Fig. 3. *(Section II-A) Overview of GNN samples.* GNN downstream ML tasks aim at classification or regression of vertices, edges, or graphs. While both vertex and edge samples virtually always have inter-sample dependencies, graphs may be both dependent and independent.

(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 Fig. 2. In general, one first applies a certain *graph-related* operation to the features. For example, in the GCN model [7], 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 [7]) and/or 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 Fig. 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 [7] 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.

Graphs – when used as basic data samples – are usually independent [15] (cf. Fig. 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 (we discuss it in Section III-B). This also has analogies in traditional deep learning, e.g., sampled videos also may have varying sizes [16]. Graph classification may also feature graph samples *with* inter-dependencies (cf. Fig. 3, 4th column). This is useful when studying, for example, relations between network communities [17]; see Section III-B for details.

B. 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 I). 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 $\mathbf{X}, \mathbf{Y}, \mathbf{H} \in \mathbb{R}^{n \times O(k)}$. 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).

C. 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 [1], [18]. 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*.

1) GNN Formulations: Local (LC) versus Global (GL): We explicitly distinguish LC and GL formulations because they have different potential for performance optimizations. GL formulations can harness techniques from linear algebra and matrix computations, such as communication avoidance [19], [20]. They also offer more potential for vectorization, as one operates on whole feature and adjacency matrices and not on individual feature vectors. LC formulations also have potential advantages. For example, functions operating on single vertices/edges can be programmed more effectively and scheduled more flexibly on low-end compute resources such as serverless functions. Moreover, the fine-grained perspective facilitates integration with vertex-centric graph processing paradigms, benefiting from established parallel frameworks such as Galois [21].

TABLE II
(SECTION III-D) WORK-DEPTH ANALYSIS OF GNN TRAINING METHODS

Method	Work & depth in one training iteration	
Full-batch training schemes:		
Full-batch [7]	$O\left(Lmk + Lnk^2\right)$	$O\left(L\log k + L\log d\right)$
Weight-tying [61]	$O\left(Lmk + Lnk^2\right)$	$O\left(L\log k + L\log d\right)$
RevGNN [61]	$O\left(Lmk + Lnk^2\right)$	$O\left(L\log k + L\log d\right)$
Mini-batch training schemes:		
GraphSAGE [52]	$O\left(Lmk + Lnk^2 + c^L nk^2\right)$	$O\left(L\log k + L\log c\right)$
VR-GCN [62]	$O\left(Lmk + Lnk^2 + c^L nk^2\right)$	$O\left(L\log k + L\log c\right)$
FastGCN [63]	$O\left(Lmk + Lnk^2 + cLnk^2\right)$	$O\left(L\log k + L\log c\right)$
Cluster-GCN [28]	$O\left(W_{pre} + Lmk + Lnk^2\right)$	$O\left(D_{pre} + L\log k + L\log d\right)$
GraphSAINT [53]	$O\left(W_{pre} + Lmk + Lnk^2\right)$	$O\left(D_{pre} + L\log k + L\log d\right)$

c is the number of vertices sampled per neighborhood or per GNN layer.

TABLE III
IMPORTANT OBJECTS AND OPERATIONS FROM LINEAR ALGEBRA USED IN GNNS

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

We assign these operations to specific GNN models in tables V, VI, and VII.

It is often highly non-trivial to provide both an LC and a GL variant of a GNN model. While some models have both formulations (e.g., GCN, GIN, Vanilla Attention, CommNet; cf. Table V), for most models, this is not the case. Many models only have known local formulations (e.g., MoNet, GAT, AGNN, G-GCN, the "pooling" variant of GraphSAGE, EdgeConv "choice 5"; cf. Table V) or global ones (e.g., SGC, ChebNet, DCNN, GDC, LINE, PPNP; cf. Table VII). Very often, developing an LC variant of a GL model is hard, e.g., for the PPNP model, it would require finding the LC equivalent of inverting the adjacency matrix. On the other hand, complex operations used to compute a score for an edge in many LC formulations (e.g., in MoNet or G-GCN) are challenging to express in GL formulations. Hence, it is important to investigate both types of formulations to ensure all these models can benefit from efficient parallel and distributed execution.

TABLE IV
WORK-DEPTH ANALYSIS OF GNN OPERATORS (KERNELS)

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)	$O(nW_\phi)$	$O(D_\phi)$	$O(1)$	$O(1)$

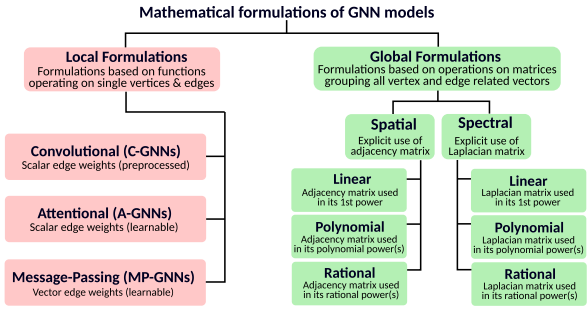


Fig. 4. *(Section II-C) Categories of GNN models.* We classify the GNN model formulations into *local* and *global*. Red/green refer to formulation details in Fig. 5.

Fig. 4 shows the taxonomy of GNN formulations. The LC sub-categories were proposed by Bronstein et al. [22]; the GL sub-categories are described by Chen et al. [23].

2) Local GNN Formulations: Details: 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 ($N(i)$ is defined as the 1-hop neighborhood) [22]. 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:

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

As an example, consider the seminal GCN model by Kipf and Welling [7]. 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 $\mathbf{h}_i^{(l+1)} = ReLU(\mathbf{W}^{(l)} \times (\sum_{j\in\widehat{N}(i)} \frac{1}{\sqrt{d_i d_j}} \mathbf{h}_j^{(l)}))$.

TABLE V
COMPARISON OF LOCAL (LC) FORMULATIONS OF GNN MODELS WITH RESPECT TO THE INNER FUNCTION $\psi(\mathbf{h}_i, \mathbf{h}_j)$

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 [7]	C-GNN	$\frac{1}{\sqrt{d_i d_j}}\mathbf{h}_j$	$c \cdot$ [▮]	🔋	$O(k)$	$O(1)$
GraphSAGE [52] (mean)	C-GNN	$\mathbf{h}_j$	[▮]	🔋	$O(1)$	$O(1)$
GIN [67]	C-GNN	$\mathbf{h}_j$	[▮]	🔋	$O(1)$	$O(1)$
CommNet [68]	C-GNN	$\mathbf{h}_j$	[▮]	🔋	$O(1)$	$O(1)$
Vanilla attention [69]	A-GNN	$\left(\mathbf{h}_i^T \cdot \mathbf{h}_j\right)\mathbf{h}_j$	$\left([\cdot\cdot] \cdot [▮]\right) \cdot [▮]$	✖	$O(k)$	$O(\log k)$
MoNet [70]	A-GNN	$\exp\left(-\frac{1}{2}\left(\mathbf{h}_j - \mathbf{w}_j\right)^T \mathbf{W}_j^{-1}\left(\mathbf{h}_j - \mathbf{w}_j\right)\right)$	$\exp\left([\cdot\cdot] \times [▮▮] \times [▮]\right)$	✖	$O\left(k^2\right)$	$O(\log k)$
GAT [8]	A-GNN	$\frac{\exp\left(\sigma\left(\mathbf{a}^T \cdot \left[\mathbf{W}\mathbf{h}_i \middle\| \mathbf{W}\mathbf{h}_j\right]\right)\right)}{\sum_{y \in \widehat{N}(i)} \exp\left(\sigma\left(\mathbf{a}^T \cdot \left[\mathbf{W}\mathbf{h}_i \middle\| \mathbf{W}\mathbf{h}_y\right]\right)\right)}\mathbf{h}_j$	$\frac{\exp\left([\cdot\cdot] \cdot \left[[▮▮] \times [▮] \middle\| [▮▮] \times [▮]\right]\right)}{\sum \exp\left([\cdot\cdot] \cdot \left[[▮▮] \times [▮] \middle\| [▮▮] \times [▮]\right]\right)} \cdot [▮]$	✖	$O\left(dk^2\right)$	$O\left(\log k + \log d\right)$
Attention-based GNNs [71]	A-GNN	$w\frac{\mathbf{h}_i^T \cdot \mathbf{h}_j}{\|\mathbf{h}_i\|\|\mathbf{h}_j\|}\mathbf{h}_j$	$\left([\cdot\cdot] \cdot [▮]\right) \cdot [▮]$	✖	$O(k)$	$O(\log k)$
G-GCN [9]	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 [52] (pooling)	MP-GNN	$\sigma\left(\mathbf{W}\mathbf{h}_j + \mathbf{w}\right)$	[▮▮] × [▮]	✖	$O(k^2)$	$O(\log k)$
EdgeConv [72] "choice 1"	MP-GNN	$\mathbf{W}\mathbf{h}_j$	[▮▮] × [▮]	✖	$O(k^2)$	$O(\log k)$
EdgeConv [72] "choice 5"	MP-GNN	$\sigma\left(\mathbf{W}_1\left(\mathbf{h}_j - \mathbf{h}_i\right) + \mathbf{W}_2\mathbf{h}_i\right)$	[▮▮] × [▮]	✖	$O(k^2)$	$O(\log k)$

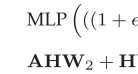
For clarity and brevity of equations, when it is obvious, we omit the matrix multiplication symbol $\times$ and the indices of a given iteration (GNN layer) number (l). "**Class**": class of a GNN model with respect to the complexity of ψ, details are in Section 2. All models considered in this table 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.

TABLE VI
COMPARISON OF LOCAL (LC) FORMULATION OF GNN MODELS WITH RESPECT TO THE OUTER FUNCTION ϕ

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 [7]	C-GNN	$\mathbf{W} \times \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_j\right)\right)$	[▮▮] × ∑ [▮]	$O(Lmk + Lnk^2)$	$O(L\log d + L\log k)$
GraphSAGE [52] (mean)	C-GNN	$\mathbf{W} \times \left(\frac{1}{d_i} \cdot \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_j\right)\right)\right)$	[▮▮] × ∑ [▮]	$O(Lmk + Lnk^2)$	$O(L\log d + L\log k)$
GIN [67]	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 [68]	C-GNN	$\mathbf{W}_1\mathbf{h}_i + \mathbf{W}_2 \times \left(\sum_{j \in N^+(i)} \psi\left(\mathbf{h}_j\right)\right)$	[▮▮] × ∑ [▮]	$O(Lmk + Lnk^2)$	$O(L\log d + L\log k)$
Vanilla attention [69]	A-GNN	$\mathbf{W} \times \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)\right)$	[▮▮] × ∑ [▮]	$O(Lmk + Lnk^2)$	$O(L\log d + L\log k)$
GAT [8]	A-GNN	$\mathbf{W} \times \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)\right)$	[▮▮] × ∑ [▮]	$O(Lmdk^2 + Lnk^2)$	$O(L\log d + L\log k)$
Attention-based GNNs [71]	A-GNN	$\mathbf{W} \times \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)\right)$	[▮▮] × ∑ [▮]	$O(Lmk + Lnk^2)$	$O(L\log d + L\log k)$
MoNet [70]	A-GNN	$\mathbf{W} \times \left(\sum_{j \in \widehat{N}(i)} \psi\left(\mathbf{h}_j\right)\right)$	[▮▮] × ∑ [▮]	$O(Lmk^2 + Lnk^2)$	$O(L\log d + L\log k)$
G-GCN [9]	MP-GNN	$\mathbf{W} \times \left(\sum_{j \in N^+(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)\right)$	[▮▮] × ∑ [▮]	$O(Lmk^2 + Lnk^2)$	$O(L\log d + L\log k)$
GraphSAGE [52] (pooling)	MP-GNN	$\left(\mathbf{W} \times \left(\mathbf{h}_i \middle\| \left(\max_{j \in N(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)\right)\right)\right)$	[▮▮] × $\left([▮] \middle\| \left([▮▮] \times \sum [▮]\right)\right)$	$O(Lmk^2 + Lnk^2)$	$O(L\log d + L\log k)$
EdgeConv [72] "choice 1"	MP-GNN	$\sum_{j \in N^+(i)} \psi\left(\mathbf{h}_j\right)$	∑ [▮]	$O(Lmk^2 + Lnk^2)$	$O(L\log d + L\log k)$
EdgeConv [72] "choice 5"	MP-GNN	$\max_{j \in N^+(i)} \psi\left(\mathbf{h}_i, \mathbf{h}_j\right)$	∑ [▮]	$O(Lmk^2 + Lnk^2)$	$O(L\log d + L\log k)$

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.

TABLE VII
COMPARISON OF GLOBAL (GL) LINEAR ALGEBRA FORMULATIONS OF GNN MODELS

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 [7]	L	$\widehat{\mathbf{A}}\mathbf{H}\mathbf{W}$		L	$O(mkL + Lnk^2)$	$O(L\log k + L\log d)$
GraphSAGE [52] (mean)	L	$\widehat{\mathbf{A}}\mathbf{H}\mathbf{W}$		L	$O(mkL + Lnk^2)$	$O(L\log k + L\log d)$
GIN [67]	L	$\text{MLP}\left(((1+\epsilon)\mathbf{I} + \widehat{\mathbf{A}})\mathbf{H}\right)$		L	$O(mkL + KLnk^2)$	$O(LK\log k + LK\log d)$
CommNet [68]	L	$\mathbf{A}\mathbf{H}\mathbf{W}_2 + \mathbf{H}\mathbf{W}_1$		L	$O(mkL + Lnk^2)$	$O(L\log k + L\log d)$
Dot Product [69]	L	$(\mathbf{A} \odot (\mathbf{H}\mathbf{H}^T))\,\mathbf{H}\mathbf{W}$		L	$O(Lmk + Lnk^2)$	$O(L\log k + L\log d)$
EdgeConv [72] "choice 1"	L	$\mathbf{A}\mathbf{H}\mathbf{W}$		L	$O(mkL + Lnk^2)$	$O(L\log k + L\log d)$
SGC [11]	P	$\widehat{\mathbf{A}}^s\mathbf{H}\mathbf{W}$		1	$O(mn\log s + nk^2)$	$O(\log k + \log s\log d)$
DeepWalk [73]	P	$\left(\sum_{s=0}^{T}\overline{\mathbf{A}}^s\right)\mathbf{H}\mathbf{W}$		1	$O(mn\log T + nk^2)$	$O(\log k + \log T\log d)$
ChebNet [74]	P	$\left(\sum_{s=0}^{T}\theta_s\overline{\mathbf{A}}^s\right)\mathbf{H}\mathbf{W}$		1	$O(mn\log T + nk^2)$	$O(\log k + \log T\log d)$
DCNN [75], GDC [76]	P	$\left(\sum_{s=1}^{T}w_s\overline{\mathbf{A}}^s\right)\mathbf{H}\mathbf{W}$		1	$O(mn\log T + nk^2)$	$O(\log k + \log T\log d)$
Node2Vec [77]	P	$\left(\frac{1}{p}\mathbf{I} + \left(1-\frac{1}{q}\right)\overline{\mathbf{A}} + \frac{1}{q}\overline{\mathbf{A}}^2\right)\mathbf{H}\mathbf{W}$		1	$O(mn + nk^2)$	$O(\log k + \log d)$
LINE [78], SDNE [79]	P	$\left(\overline{\mathbf{A}} + \theta\overline{\mathbf{A}}^2\right)\mathbf{H}\mathbf{W}$		1	$O(mn + nk^2)$	$O(\log k + \log d)$
Auto-Regress [80], [81]	R	$\left((1+\alpha)\mathbf{I} - \alpha\widehat{\mathbf{A}}\right)^{-1}\mathbf{H}\mathbf{W}$		1	$O(n^3 + nk^2)$	$O(\log^2 n + \log k)$
PPNP [54], [82], [83]	R	$\alpha\left(\mathbf{I} - (1-\alpha)\widehat{\mathbf{A}}\right)^{-1}\mathbf{H}\mathbf{W}$		1	$O(n^3 + nk^2)$	$O(\log^2 n + \log k)$
ARMA [84], ParWalks [85]	R	$b\left(\mathbf{I} - a\widehat{\mathbf{A}}\right)^{-1}\mathbf{H}\mathbf{W}$		1	$O(n^3 + nk^2)$	$O(\log^2 n + \log k)$

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.

Note that each iteration may have different projection matrices $\mathbf{W}^{(l)}$.

Depending on the details of ψ, one can further distinguish three GNN classes [22]: *Convolutional GNNs* (C-GNNs), *Attentional GNNs* (A-GNNs), and *Message-Passing GNNs* (MP-GNNs). Example models from each class can be found in Table V. In short, in these three classes of models, ψ respectively applies – as a weight on the features – a fixed scalar coefficient (C-GNNs), a learnable scalar coefficient (A-GNNs), or a learnable vector coefficient (MP-GNNs).

Importantly, these approaches form a hierarchy, i.e., C-GNNs $\subseteq$ A-GNNs $\subseteq$ MP-GNNs [22]. Specifically, A-GNNs can represent C-GNNs by implementing attention as a look-up table $a(x_u, x_v) = c_{uv}$. Then, both C-GNNs and A-GNNs are special cases of MP-GNNs: $\psi(x_u, x_v) = c_{uv}\psi(x_v)$ (for GNNs) and $\psi(x_u, x_v) = a(x_u, x_v)\psi(x_v)$ for A-GNNs.

Note that we follow the taxonomy established by Bronstein et al. [22], [24], where MP-GNNs is a parent class of C-GNNs, A-GNNs, but also more specialized message-passing model classes such as MPNN by Gilmer et al. [25] or Graph Networks by Battaglia et al. [26].

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.

3) Global GNN Formulations: Details: 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 I): $\widehat{\mathbf{A}} = \widetilde{\mathbf{D}}^{-\frac{1}{2}}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-\frac{1}{2}}$. 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) [23]. This aspect impacts how to best parallelize a given model, as we illustrate in Section IV. For example, the GCN model [7] is linear.

GNN computations involve both sparse and dense matrices, which enbtail different performance patterns [27]. Hence, this comes with potential for different parallelization routines. We analyze this in more detail in Section IV.

D. GNN Inference versus GNN Training

A series of GNN layers stacked one after another, as detailed in Fig. 2 and in Section II-C, 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: $\mathcal{L} = \frac{1}{|\mathcal{Y}|} \sum_{i \in \mathcal{Y}} \text{loss}(\mathbf{y}_i, \mathbf{t}_i)$, 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 [28]; other functions may also be used [29].

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 [30]:

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

where $\mathbf{T}$ is a matrix grouping all the ground-truth vertex labels, cf. Table I for other symbols. This equation reflects the forward propagation formula (cf. Section II-C3); 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, see Section III.

E. 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). Fig. 5 shows both LC and GL formulations, and how they translate to the programming kernels.

The most widespread programming/execution model is SAGA [12] ("Scatter-ApplyEdge-Gather-ApplyVertex"), used in many GNN libraries [32]. 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.

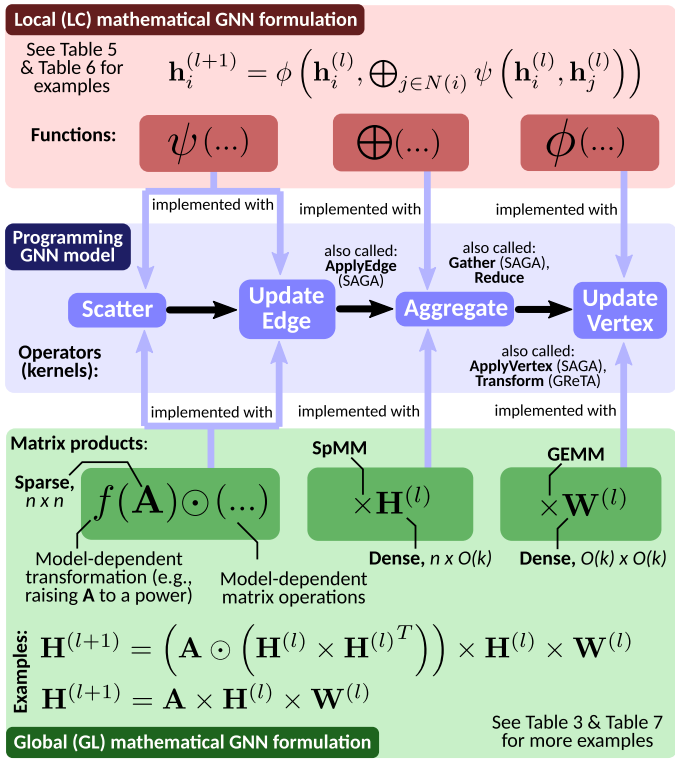


Fig. 5. *(Sections II-C, II-D, and II-E) GNN model formulations* (top part: the local (LC) approach, bottom part: the global (GL) approach), and *how they translate into GNN operators* (central part). SAGA [12], NAU [31], and GReTA [13] are GNN programming models. Red/green indicate formulations from Fig. 4.

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 [13], FlexGraph [31], 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 Fig. 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 [33].[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 IV. We also analyze parallel and distributed frameworks and accelerators based on these models in Section V.

F. Taxonomy of Parallelism in GNNs

In traditional DL, there are two fundamental ways to parallelize the processing of a neural network [34]: *data parallelism*

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

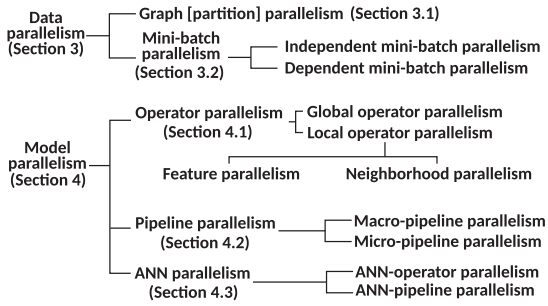


Fig. 6. (Section II-F) Parallelism taxonomy in GNNs.

and *model parallelism* where one partitions, respectively, data samples and neural weights among different workers. Parallelism in GNNs also has data parallelism (detailed in Section III) and model parallelism (detailed in Section IV). Yet, there are certain differences that we identify and analyze. We overview the GNN parallelism taxonomy and the classes of GNN parallelism in Figs. 6 and 7, respectively.

The first form of data parallelism in GNNs is *independent mini-batch parallelism*. Here, parallel workers process a single mini-batch; the samples in this mini-batch have no inter-sample dependencies (i.e., the samples are independent graphs, see Fig. 3). This form of parallelism is analogous to the one in deep learning with images, where one parallelizes a mini-batch of pictures. Second, GNNs also exhibit *dependent mini-batch parallelism*. Here, a mini-batch is also processed in parallel by multiple workers, but the samples do have inter-sample dependencies (e.g., a mini-batch could be a set of vertices and edges, sampled from a large input graph). These dependencies make parallelization much more complex, as we detail in Section III. Note that in GNNs, as in traditional DL, one can also use full-batch training. Finally, one can combine both mini-batch parallelism and full-batch processing with *graph [partition] parallelism*. Here, one distributes a given mini-batch or a whole batch across different workers, usually to fit it in memory.

Model parallelism in GNNs can be divided into *operator parallelism*, *artificial neural network (ANN) parallelism*, and *pipeline parallelism*. First, in operator parallelism, one parallelizes the Scatter and Reduce kernels. Here, we further distinguish between *[graph] local operator parallelism* (parallel processing of individual vertices and edges) and *[graph] global operator parallelism* (parallel processing of collections of vertices/edges). Examples of local operator parallelism are *feature parallelism* (processing a feature vector of a given vertex in parallel) and *graph [neighborhood] parallelism* (processing in parallel the edges to the neighbors of a given vertex). Second, in *ANN parallelism*, one parallelizes the UpdateEdge and UpdateVertex kernels. These kernels can harness any form of parallelism that has been developed for traditional deep neural networks such as MLPs [34]. Examples of ANN parallelism are *ANN-pipeline parallelism* (pipelining MLP layers) and *ANN-operator parallelism* (parallel processing of single NN operations). Finally, in *[GNN] pipeline parallelism*, one assigns different workers to different stages of the GNN processing pipeline. Here, we distinguish *macro-pipeline parallelism* (pipelining the whole GNN layers) and *micro-pipeline parallelism* (pipelining the stages within a single GNN layer).

G. 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 run-times 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 [10], [35]. 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 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)* [36].

III. 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 [34]. 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. Fig. 3 and Section II-A. 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 Fig. 8.

A. Graph Partition Parallelism

Some graphs may have more than 250 billion vertices and beyond 10 trillion edges [37], and each vertex and/or edge may have a large associated feature vector [38]. 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 [39], [40] and the associated mincut problem [40], [41], [42]. 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 Fig. 9. When distributing a graph over different workers and servers, one can specifically distribute vertices (*edge [structure] partitioning*,

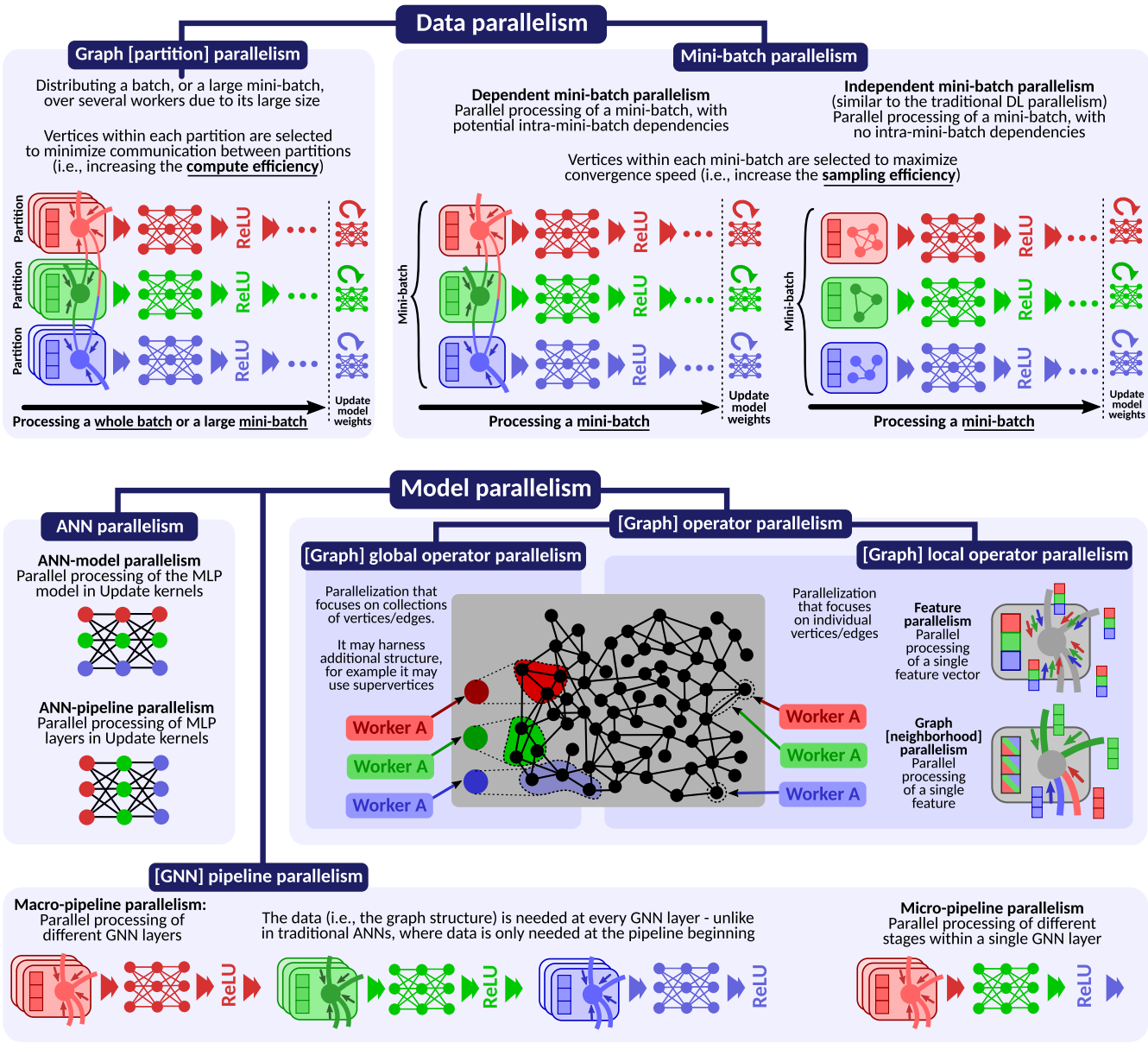


Fig. 7. (Section II-F) Overview of parallelism in GNNs. Different colors (red, green, blue) correspond to different workers.

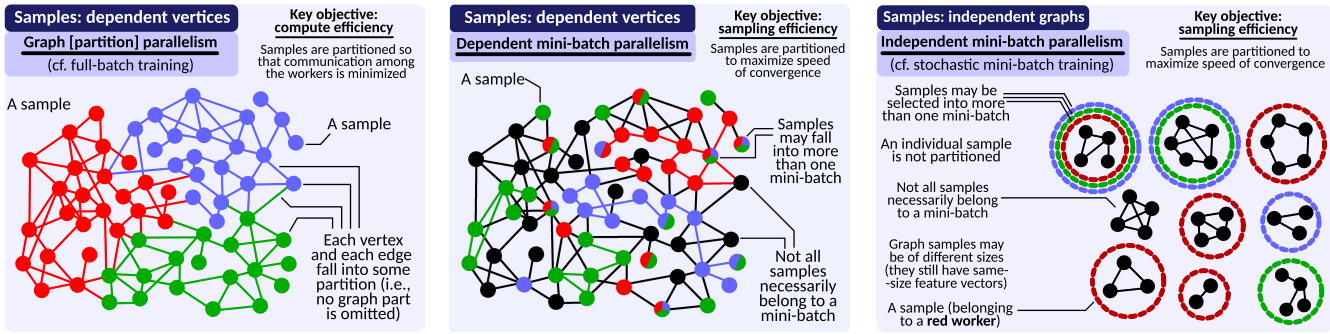


Fig. 8. *(Sections III-A and III-B) Graph partition parallelism versus dependent and independent mini-batch parallelism in GNNs.* Different colors (**red**, **green**, **blue**) indicate different graph partitions or mini-batches, and the associated different workers. Note that applying different colors to a vertex or to an independent graph does not mean physical partitioning but it indicates that a given vertex or a given indepedent graph – as a whole – is used in more than a single mini-batch, indicated by the respective color. Black vertices do not belong to any mini-batch.

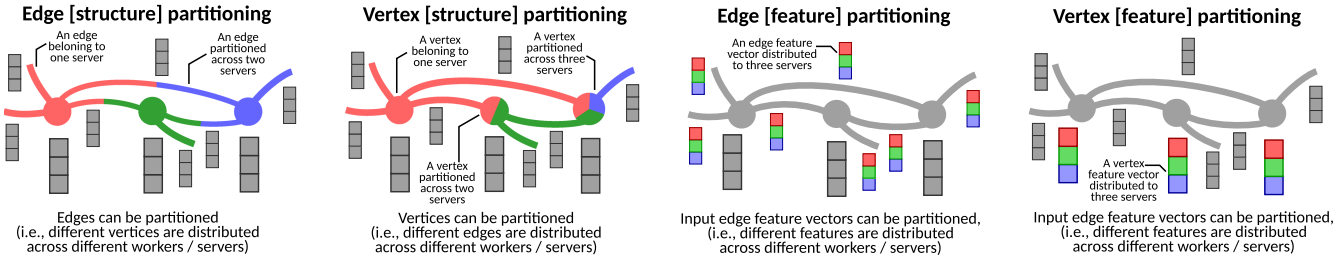


Fig. 9. *(Section III-A) Different forms of graph partition parallelism.* Different colors (**red**, **green**, **blue**) indicate different graph partitions, and the associated different workers. The **gray** graph element is oblivious to a given form of partitioning. *Note that different partitioning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

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 also incurs overheads in maintaining a consistent distributed vertex state. Differences between edge and vertex partitioning are covered in rich literature [39], [40], [43], [44], [45]. Feature vertex partitioning was not addressed in the graph processing area because in traditional distributed graph algorithms, vertices and/or edges are usually 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.

Partition parallelism usually does not allow a single vertex to belong to multiple partitions (unlike mini-batch parallelism, where a single sample may belong to more than one mini-batch). However, there are strategies for reducing communication, in which vertices are cached on remote partitions. Such schemes would involve maintaining multiple copies of a given vertex on several partitions.

Note that, while graph partition is usually conducted once, before training starts, it could also be in principle reapplied during training, to alleviate potential load imbalance (e.g., due to inserting new vertices or edges). Such schemes are an interesting direction for future work.

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.

B. 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. This may cause *load imbalance*), similarly to, e.g., videos [16]. For example, a single dataset (e.g., ChemInformatics) [46] may contain graphs both 5 vertices and 18 edges as well as with 121 vertices and 298 edges. This setting is common in graph classification or graph regression. We illustrate this in Fig. 8 (right), and we refer to it as *independent mini-batch parallelism*. Note that – while graph samples may have different sizes (e.g., molecules can have different counts of atoms and bonds) – their feature counts are the same.

Still, 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. Several works alleviate this [47], [48].

While the graph prediction setting has been explored in data science works [1], [38], [49], it has been largely unaddressed in system design studies [50]. As such, to the best of our knowledge, there are no detailed existing load balancing studies or schemes for independent mini-batch parallelism (unlike for node or edge predictions [50]). Dependent mini-batch parallelism *with graphs as samples* is even more scarcely researched; for example, it does not even have a representing dataset in the Large-Scale OGB challenge [38]. Hence, we list these as research opportunities in Section VII.

Another 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

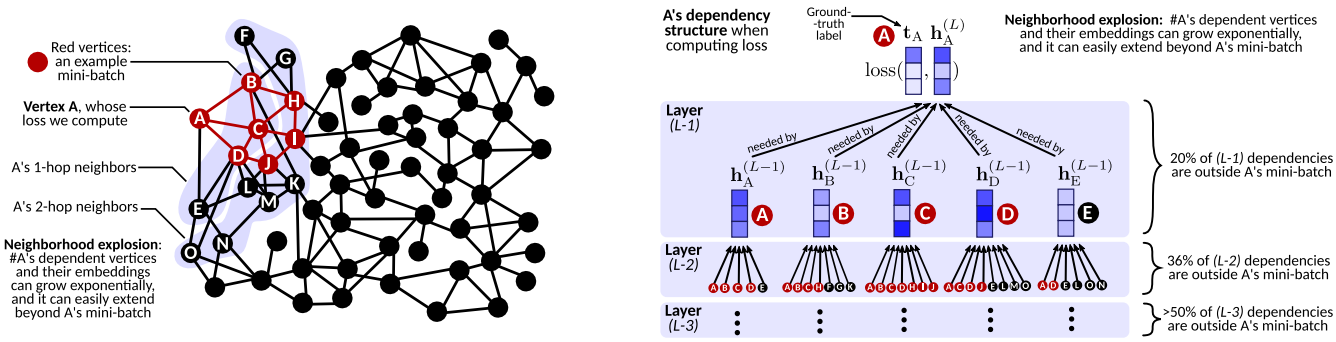


Fig. 10. *(Section III-B) Neighborhood explosion* in mini-batching in GNNs. This phenomenon is characteristic to schemes based on node-wise sampling [51].

structure is lost during the Aggregate or Scatter operator. This information loss challenge was circumvented in the early GNN works with full-batch training [1], [7] (cf. Section III-A1). 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 recent approaches that address specifically mini-batching proposed *sampling neighborhoods*, and *appropriately selecting target vertices*.

1) Neighborhood Sampling: In a line of works initiated by GraphSAGE [52], one adds sampled neighbors of each selected target vertex v to the mini-batch. Sampled neighbors of v usually come from not only 1-hop, but also from H-hop neighborhoods of v, where H may be as large as graph's diameter. The exact selection of sampled neighbors depends on the details of each scheme. In GraphSAGE, they are sampled (for each target vertex) for each GNN layer before the actual training.

One challenge related to neighborhood sampling 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 conduct sampling for each layer and for each target 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.[2] Even if some of these vertices are target vertices in that mini-batch and thus are already maintained, when increasing 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 [53]. We show an example neighborhood explosion in Fig. 10.

[2]The above bound is not tight because not all overlaps (e.g., between sampled neighbors of different target vertices) are considered. However, we reflect the approach taken by all the considered GNN schemes analyzed in Table II. To enhance the bound, one needs additional assumptions, e.g., on the graph structure or its generating model.

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 assigns clusters to mini-batches [28], [54]. This way, one reduces the loss of information because a mini-batch usually contains a tightly knit community of vertices. 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 [40].

C. Graph Partitions versus Mini-Batch/Full-Batch Training

Graph partitions are used primarily to contain the graph fully in memory (avoiding expensive disk accesses), while mini-batches are used to speed up convergence. The first fundamental difference between these two is the key objective when splitting the graph dataset across workers. For graph partition parallelism, one aims to maximize *compute efficiency*, i.e., minimize runtimes by minimizing the amount of communication between workers. For the former, one focuses on increasing *sampling efficiency*, i.e., creating mini-batches in such a way that the convergence speed is maximized. With certain schemes, these objectives could result in selecting similar vertices. For example, Cluster-GCN selects dense clusters as mini-batches and such clusters could also be effective graph partitions [28]. However, this is not always the case as other mini-batching schemes do not necessarily focus on dense clusters [55]. Another difference is that, while a single graph partition is processed by a single worker (i.e., multiple workers process multiple partitions), one mini-batch is processed by multiple workers. We also note that one could consider the parallel processing 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.

Commonly, one uses graph partition parallelism with full-batch training [30], [56], [57], [58]. However, in principle, graph partition and mini-batch parallelism are orthogonal to each other, and could thus be used together. For example, a large mini-batch running on workers with not much memory could utilize graph

partition parallelism to avoid I/O. Such an approach has also been proposed in traditional DL [59], [60].

D. Work-Depth Analysis

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

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 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 GraphSAINT, 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.

E. 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 [34]. 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 [28].

$$\begin{aligned}
\text{Work (generic)} &= W_{pre} + LW_l + W_{post} \\
\text{Depth (generic)} &= D_{pre} + LD_l + D_{post} \\
\text{Work of Eq. (1)} &= W_{pre} + L \cdot \left(mW_\psi + nW_\oplus + nW_\phi\right) + W_{post} \\
\text{Depth of Eq. (1)} &= D_{pre} + L \cdot \left(D_\psi + D_\oplus + D_\phi\right) + D_{post}
\end{aligned}$$

Fig. 11. Generic equations for work and depth in GNN LC formulations.

IV. 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 Section II-F and in Fig. 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 III 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.

A. Local Operator Parallelism

In local operator parallelism, one focuses on parallelizing executions of Scatter and Gather on individual vertices or edges (i.e., local graph elements). We further structure our investigation by considering separately local operator parallelism over LC and GL GNN model formulations.

1) Parallelism in LC Formulations of GNN Models: We illustrate generic work and depth equations of LC GNN formulations in Fig. 11. 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 (1), 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).

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 V and VI. We select the representative models based on a recent survey [23]. We also indicate whether a model belongs to the class of convolutional (C-GNN), attentional (A-GNN), or message-passing (MP-GNN) models [24] (cf. Section II-C2).

Analysis of ψ We show the analysis results in Table V. 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 ($c_{ij} = 1/\sqrt{d_i d_j}$) [1]. 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.

Note that, by default, ψ corresponds to edge feature vectors that are "transient", i.e., they are computed on the fly and are not stored explicitly (unlike 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. [26], also an LC formulation. The insights about parallelism in such a formulation are not different than the ones provided in this section; the main different is the additional memory overhead of $O(mk)$ needed for storing all edge feature vectors.

Analysis of $\bigoplus$ The aggregate operator $\bigoplus_{j \in N(i)}$ is almost always commutative and associative (e.g., min, max, or plain sum [64], [65]). 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 [66]. 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 VI (for the same models as in Table V). 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 Fig. 12. We provide the corresponding generic work-depth analysis in Table IV. The four programming kernels follow the work and depth of the corresponding LC functions ($\psi, \oplus, \phi$).

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 V and VI. 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 IV-C3, we discuss how this requirement can be relaxed by allowing asynchronous execution.

2) Parallelism in GL Formulations of GNN Models: Parallelism in GL formulations is analyzed in Table VII. 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 $\mathbf{H}$.

There are also models designed in the GL formulations with no known LC formulations, cf. Tables V and VI. These are models that use polynomial and rational powers of the adjacency matrix, cf. Section II-C3 and Fig. 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 [86]). 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 Fig. 12 (bottom). In this figure, we also indicate how the LC GNN kernels are reflected in the flow of matrix operations in the GL formulation.

Communication & 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

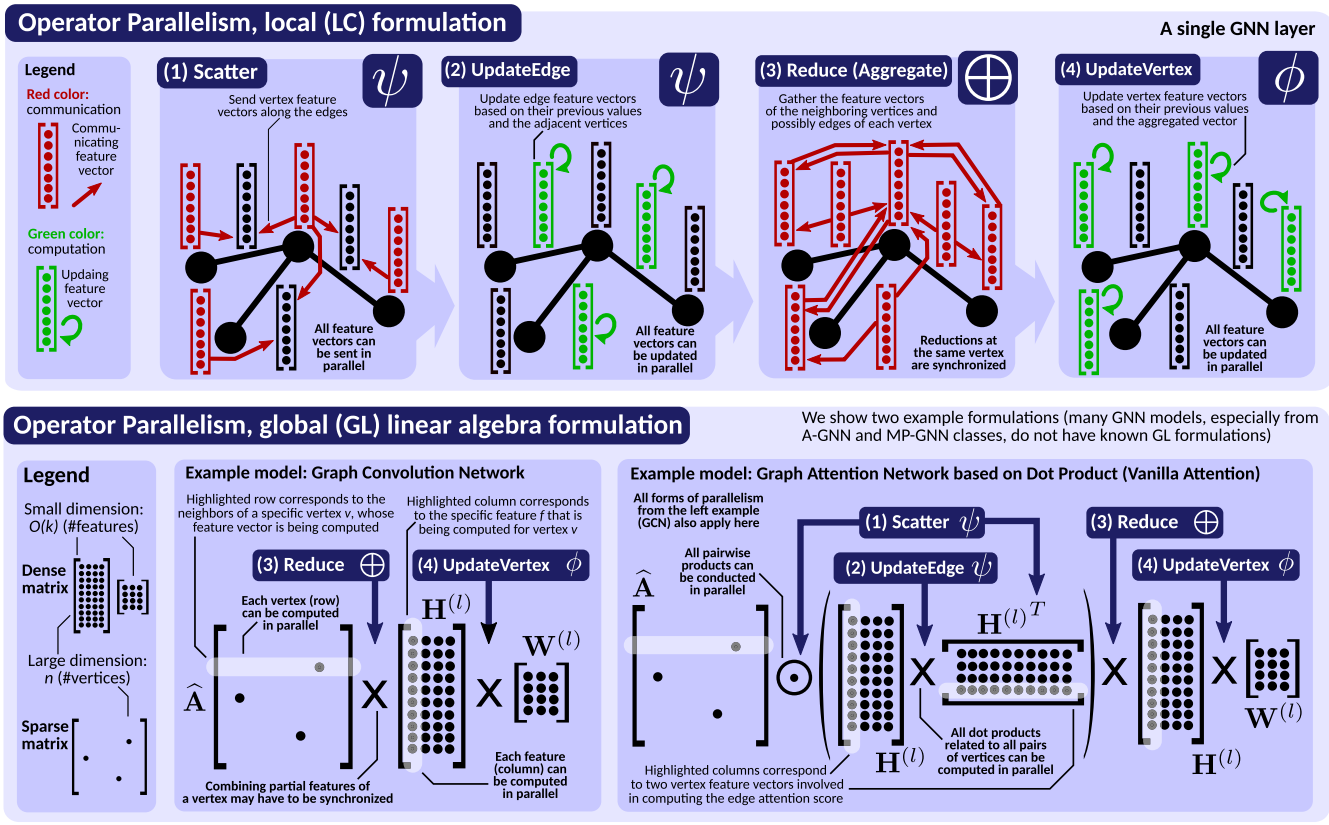


Fig. 12. *Operator parallelism in GNNs* for LC formulations (top) and GL formulations (bottom).

targeting dense matrix multiplications [19], [87], [88], [89] or sparse matrix multiplications [20], [90], [91], [92]. They could be used with different GNN operations (cf. Table III) and different models (cf. Table VII). 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 [20]. This could be used in GNNs for more performance.

3) Discussion: Feature versus Structure versus Model Weight Parallelism: Feature parallelism is straightforward in both LC and GL formulations (cf. Fig. 12). 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 neighborhood parallelism is available 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 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 [38], [49], 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 [34].

Graph [Neighborhood] versus Graph [Partition] Parallelism: Graph [partition] parallelism is used to partition the graph as a whole among workers in order to contain the graph fully in memory. Graph [neighborhood] parallelism is used solely among the neighbors of a single vertex during aggregation, and it can be applied orthogonally to graph [partition] parallelism. For example, a single graph partition may contain a high-degree vertex, and the aggregation applied to this vertex can be parallelized within this partition.

The relation between graph partition parallelism and graph neighborhood parallelism is similar to that of macro-pipeline and micro-pipeline parallelism. Specifically, while in macro-pipeline parallelism, one partitions whole GNN layers among the workers, parts of this single layer can be further parallelized using micro-pipeline parallelism, orthogonally to the macro-pipeline structure.

B. Global Operator Parallelism

In global operator parallelism, one executes Scatter & Gather in parallel over *collections* of vertices or edges. This approach could utilize additional structure on top of the processed graph for more performance. For example, one could harness primitives similar to graph contraction and supervertices (used in

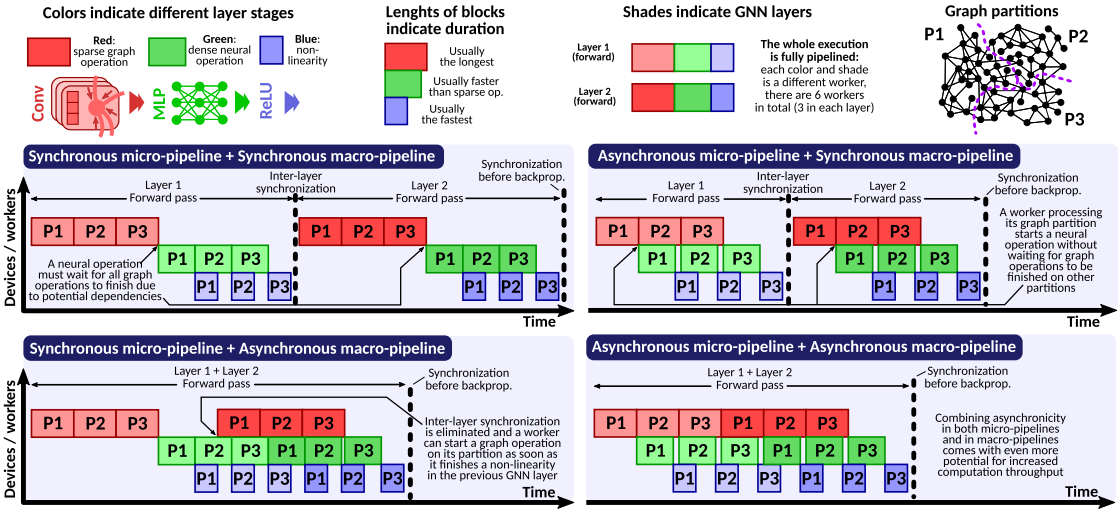


Fig. 13. *(Section IV-C) Overview of pipelining combined with graph partition parallelism* (top-left panel), and of *(Section IV-C3) asynchronous pipelined execution (other panels)*. Each of four example GNN executions processes three graph partitions (P1, P2, P3) on three stages: **red** (a sparse graph 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).

Karger's algorithm for min-cuts [93]) or hooking trees (used in Shiloach and Vishkin's algorithm for connected components [94]). Such supervertices or trees could potentially be used to speed up Gather.

Global Operator versus Graph Partition Parallelism: Graph partition parallelism is a straightforward approach that only distributes vertices and edges across different workers so that they can all fit into memory. Global operator parallelism, on the contrary, is more complex and it can harness additional structure, such as supervertices, on top of the processed collections of vertices/edges.

C. 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 Fig. 7 and Section II-F.

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. Fig. 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 $3\times$, 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 V.

We show an example micro-pipeline (synchronous) in the top panel of Fig. 13. 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. Fig. 3), where chunks have no inter-chunk dependencies, and thus neural processing of P1 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.

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 [95], PipeDream [96], or Chimera [97]. However, pipelining GNN layers is more difficult because of dependencies between data samples, and it is only in its early development stage [30]. In Fig. 13, the execution is fully pipelined, i.e., all layers are processed by different workers.

3) Asynchronous Pipelining: In asynchronous pipelining, pipeline stages proceed without waiting for the previous stages to finish [96]. 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 (Fig. 13, 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 (Fig. 13,

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. 14. *(Section IV-C3) Message Passing GNN formulation that includes graph partition parallelism combined with fully synchronous (top) and potentially stale asynchronous computation (bottom).* The equation generalizes the Message Passing formulation [25] and past synchronous GCN models [56].

bottom-left panel). Finally, *both forms can be combined*, see Fig. 13 (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) Theoretical Formulation of Arbitrarily Deep Pipelines: To understand GNN pipelining better, we first provide a variant of (1), namely (2), which defines a synchronous Message Passing GNN execution with graph partition parallelism (Fig. 14). 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 (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, (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 (2) into (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 (3) such that these six parameters *upper bound the maximum allowed staleness*, i.e., (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. Fig. 13, top-left panel). Setting any of these parameters to be larger than this introduces staleness. For example, PipeGCN [56] proposes to pipeline communication and computation *between training iterations* in the GCN model [7]

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):

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

Fig. 15. *(Section IV-C3) Generalization of computing gradients in GNNs* to include graph partition parallelism combined with fully synchronous (top) and potentially stale asynchronous computation (bottom).

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 (3) for future work.

Finally, we also obtain the equivalent formulations for the asynchronous computation of *stale gradients*, see Fig. 15. This establishes a similar approach for optimizing backward propagation passes.

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 [98] that could be used to further accelerate the Aggregate operator ($\bigoplus$).

D. Artificial Neural Network (ANN) Parallelism

In some GNN models such as GIN [67], the dense UpdateVertex or UpdateEdge kernels are MLPs. They can be parallelized with traditional DL approaches, which are not the focus of this work; they have been extensively described elsewhere [34]. Overall, one can use *ANN-operator parallelism* (parallel processing of single NN operators within one layer, e.g., computing the value of a single neuron) and *ANN-pipeline parallelism* (parallel pipelined processing of consecutive MLP layers), cf. Fig. 7.

We identify 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*.

E. 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* [99]. More elaborate forms of model parallelism are also possible. An example is Mixture of Experts (MoE) [100], in which different models could be evaluated in parallel. Currently, MoE usage in GNNs is in its infancy [101], [102].

V. Frameworks, Accelerators, Techniques

We finally analyze existing GNN SW frameworks and HW accelerators.[3] For this, we first describe parallel and distributed architectures used by these systems.

A. Parallel and Distributed Computing Architectures

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) [103] 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 [50]. 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 VIII, 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 [104], and they thus have high potential for also accelerating sparse GNN operations.

There is also interest in HW accelerators for GNNs (Table VIII, 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.

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 [38], which often requires multi-node settings to avoid expensive I/Os. We observe (Table VIII, 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.

B. 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 [12] or Gunrock [148]). On the contrary, systems in the second category (e.g., the initial PyG design [65]) 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.

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 [116], [149], [150], [151]. For example, Zhang et al. [152] analyze the computational graph of GNNs, and propose optimizations tailored specifically for GNNs.

C. 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 [39]. 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 [129] combines sharding with replication, in order to accelerate GNN training by communication avoidance [153].

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 [134] combine graph partition parallelism with pipelining, offering further speedups and reductions in used storage resources.

[3] 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.

TABLE VIII
COMPARISON OF GNN PROCESSING FRAMEWORKS AND ANALYSES OF GNN IMPLEMENTATIONS

Reference	Arch.	Ds?	T?	I?	Op?	mp?	Mp?	Dp?	Dpp	PM	Remarks
[SW] PipeGCN [56]	CPU+GPU	■	■ (fb)	✖	■	■	✖	■	sh	LC	
[SW] BNS-GCN [57]	GPU	■	■ (fb)	✖	?	?	✖	■	sh		
[SW] PaSca [105]	GPU	■	■	■	?	?	?	■			
[SW] Marius++ [106]	CPU	✖	■ (mb)	✖	■ (v)	?	✖	■		LC (SU)	Focus on using disk
[SW] BGL [107]	GPU	■	■ (mb)	✖	■	■	✖	■	sh	—	
[SW] DistDGLv2 [108]	CPU+GPU	■	■ (mb)	✖	■	■	✖	■	sh	—	
[SW] SAR [109]	CPU	■	■ (fb)	✖	✖	✖	✖	■			
[SW] DeepGalois [58]	CPU	■	■ (fb)	✖	?	✖	✖	■ (v)	sh	LC (AU)	
[SW] DistGNN [110]	CPU	■	■ (fb)	✖	?	✖	✖	■ (v)	sh	LC(AU) ?	
[SW] DGCL [111]	GPU	◧*	■	✖	?	✖	✖	■ (v)		LC (AU)	*Only two servers used.
[SW] Seastar [112]	GPU	✖	■	✖	■ (f)	✖	✖	■ (v, t)		LC (VC)	
[SW] Chakaravarthy [113]	GPU	■	■ (fb)	✖	✖	✖	✖	■ (v, sn)		✖	
[SW] Zhou et al. [114]	CPU	✖	✖	■	■ (f)	✖	✖	?		✖	
[SW] MC-GCN [115]	GPU	◧*	■ (fb)	✖	■ (f)	✖	✖	■ (v)		GL	*Multi-GPU within one node.
[SW] Dorylus [30]	CPU	■	■ (fb)	✖	✖	✖	■	■ (v)		LC (SAGA)	
[SW] GNNAdvisor [116]	GPU	✖	■	■	■ (f, n)	✖	✖	?		GL, LC	
[SW] AliGraph [117]	CPU	■	■	■	?	✖	?	■		LC (NAU)	
[SW] FlexGraph [31]	CPU	■	■ (fb)	✖	■ (n)	?	✖	■		LC (NAU)	
[SW] Kim et al. [118]	CPU+GPU	✖	■ (mb)	✖	■ (n)	?	✖	■		LC (AU)	
[SW] AGL [119]	CPU	■	■ (mb)	■	✖	■	■	■		MapReduce	
[SW] ROC [120]	CPU+GPU	■	■ (fb)	■	✖	✖	✖	■		✖	
[SW] DistDGL [121]	CPU	■	■ (mb)	✖	?	✖	✖	■		✖	
[SW] PaGraph [122], [123]	GPU	◧*	■ (mb)	✖	?	✖	✖	■		✖	*Multi-GPU within one node.
[SW] 2PGraph [124]	GPU	■	■ (mb)	✖	?	■	?	■		—	
[SW] GMLP [125]	GPU	■	■ (mb)	✖	?	✖	?	■		LC	
[SW] fuseGNN [126]	GPU	✖	■	✖	■	✖	✖	■		LC (AU)*	*Two aggregation schemes are used.
[SW] P^3 [127]	CPU+GPU	■	■ (mb)	✖	■	■	✖	■		LC (SAGA)*	*A variant called P-TAGS
[SW] QGTC [128]	GPU	✖	✖	■	■	◧	◧	■	sh	GL	
[SW] CAGNET [129]	CPU+GPU	■	■ (fb)	✖	■ (f, n)	✖	✖	■ (v, e)	sh+rep	GL	
[SW] PCGCN [130]	CPU+GPU	✖	■	✖	✖	✖	✖	■ (e)	sh	—	
[SW] FeatGraph [131]	CPU, GPU	✖	■ (fb)	■	■ (f, n)	✖	✖	■ (v)	sh	GL	
[SW] G^3 [132]	GPU	✖	■	■	?	?	?	?		GL	
[SW] NeuGraph [12]	GPU	■	■	✖	◧	■	✖	■ (v, e)	sh	LC (SAGA)	
[SW] PyTorch-Direct [133]	GPU	■	■	■	■	?	?	■ (v, e)		GL, LC	
[SW] PyG [65]	CPU, GPU	■	■*	■	■	?	?	■ (v, e)		GL, LC	*Mini-batching for graph components
[SW] DGL [64]	CPU, GPU	■	■*	■	■	?	◧	■		GL, LC	*Mini-batching for graph components
[HW] ZIPPER [134]	new	✖	✖	■	◧	■	✖	■ (v, e)	sh	GL, LC	
[HW] GCNear [135]	new (PIM)	✖	■ (fb)	✖	■	■	✖	■ (v, e)	sh	LC (AU)	
[HW] BlockGNN [136]	new	✖	✖	■	◧	■	✖	■		LC	
[HW] TARe [137]	new (ReRAM)	✖	✖	■	◧	■	✖	?		GL	
[HW] Rubik [138]	new	✖	■ (mb)	✖	◧	?	✖	■ (v, e)	sh	LC (AU)	
[HW] GCNAX [139]	new	✖	✖	■	◧	?	✖	?		GL	
[HW] Li et al. [140]	new	✖	✖	■	■	■	■	■		LC (AU)	
[HW] GReTA [13]	new	✖	✖	■	?	?	?	■	sh	LC (GReTA)	
[HW] GNN-PIM [141]	new (PIM)	✖	✖	■	?	■	✖	■ (v)	sh	LC (SAGA)	
[HW] EnGN [142]	new	✖	✖	■	◧	?	✖	■ (v, e)	sh	LC (AU + "feature extraction" stage)	
[HW] HyGCN [14]	new	✖	✖	■	■	■	✖	■ (v, e)	sh	LC (AU)	
[HW] AWB-GCN [143]	new	✖	✖	■	◧	■	■	■ (v, e)	sh	GL	
[HW] GRIP [144]	new	✖	✖	■	?	■	◧	■ (v, e)	sh	GL, LC (GReTA)	
[HW] Zhang et al. [145]	new	✖	✖	■	■	■	◧	■ (v, e)	sh	GL	
[HW] GraphACT [146]	new	✖	■ (mb)	✖	◧	■	✖	?		GL	
[HW] Auten et al. [147]	new	✖	✖	■	?	?	?	?		LC (AU)	

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, "**n**": neighborhood 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.

Operator parallelism is supported by most systems. Both *feature parallelism* and *neighborhood parallelism* have been investigated, and there are systems supporting both (FeatGraph [131], GNNAdvisor [116], CAGNET [129]). Most of these systems target these forms of parallelism explicitly (e.g., by programming binary reduction trees processing Reduce). For example, Seastar [112] focuses on combining feature-level, vertex-level, and edge-wise parallelism. 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" [154]).

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 [144] and work by Zhang et al. [145], where pipelining is enabled by simply loading the weights of the next GNN layer, while the current layer is being processed.

D. 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 [58], DistGNN [110]) use vertex cuts. Others (e.g., DGCL [111], QGTC [128]) incorporate METIS partitioning [40]. ROC [120] 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 [58] and DGCL [111] use message combination and aggregation, DGCL also balances loads on different interconnect links, DistGNN [110] delays updates to optimize communication, Min et al. [133] use zero copy in the CPU–GPU setting (GPU threads access host memory without requiring much CPU interaction) and computation & communication overlap, and GNNAdvisor [116] and 2PGraph [124] renumber nodes to achieve more locality [155]. 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 [28].

Moreover, there are many interesting *operator related optimizations*. A common optimization is operator fusion (sometimes referred to as "kernel fusion" [126]), in which one merges the execution of different operators, for example Aggregate and UpdateVertex [64], [111], [112], [126]. 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 [128]. Other interesting schemes include operator reorganization [152], 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.

Some systems *hybridize* various aspects of GNN computing. For example, Dorylus [30] executes graph-related sparse operators (e.g., Aggregate) on CPUs, while dense tensor related operators (e.g., UpdateVertex) run on Lambdas. fuseGNN [126] 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.

VI. Selected Insights

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

- *Two biggest bottlenecks in LC GNNS are reduction $\bigoplus$ and the number of layers K in MLPs (in UpdateVertex/UpdateEdge kernels)* First, our formal analysis illustrates that the depth of a single GNN layer in all the considered LC models (except for GIN) is $O(\log d + \log k)$. In today's models, #features k is many orders of magnitude smaller than the maximum degree d. Thus, the reduction $\bigoplus$ – which is responsible for the $\log d$ term – is the bottleneck in today's LC GNNs, and it is important to optimize it (e.g., using pipelined reduction trees [98]. Second, while most GNN models use linear projections to implement UpdateVertex/UpdateEdge, some models (e.g., GIN) use MLPs with at least one hidden layer K. The depth of such GNN models depends linearly on K, and it then becomes another bottleneck for a GNN layer. In such models, it is important to also optimize the depth of MLP (e.g., using the ANN-pipeline parallelism).
- *Polynomial GL models are the most parallelizable* Models such as LINE, DCNN, or GDC, come with lowest overall depth of one training iteration, being $O(\log k + \log d \log T)$ (where T is the power of the adjacency matrix used in a given model). Simultaneously, all the LC GNN models have the depth of a single iteration being linear with respect to L (#GNN layers). However, polynomial models do not use non-linearities, making their predictive power potentially lower than that of LC GNNs. This indicates that when scalability is of top importance, it may be desirable to consider a polynomial GL model. Moreover, our blueprint for asynchronous LC GNNs may be the key to LC GNNs that have both high predictive power *and* high parallelism, as asynchronicity directly reduces the overheads due to L; this is one of the research opportunities that we detail in the next section.
- *GAT has the highest amount of work, followed by other A-GNNs and by MP-GNNs* The established Graph Attention Network model comes with the highest amount of work. Simultaneously, it still is *as parallelizable as other models* (i.e., its depth is equal to the depth of other models). Hence, when one has enough parallel workers, the high amount of work needed for GAT is alleviated. However, with limited parallel resources, it may be better to consider other GNN models that need less work.
- *GNNs come with new forms of parallelism* Graph partition parallelism (Section III), 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 neighborhood parallelism (Section V-C).
- *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 III and assigned to models in Tables V, VI, and VII.

- *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 VII). However, many local GNN formulations have complex tensor operations associated with UpdateEdge or UpdateVertex operators, see Tables V and VI.
- *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 the same work and depth, LC and GL 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 [19], [20], [153]. 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 [21].

VII. 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 [95] or PipeDream [96], that investigate asynchronous or bi-directional mini-batching, such works are yet to be developed for GNNs. Here, our blueprint for *asynchronous GNNs* can spearhead novel works.
- *More performance in GNNs via replication* Many GNN works have explored graph partitioning. However, very few (e.g., CAGNET [129]) uses replication for more performance (e.g., through 2.5D & -3D matrix multiplications).
- *Parallelization of GNN models beyond simple C-GNNs* There is not much work on parallel and distributed GNN models beyond the simple seminal ones from the C-GNN or A-GNN classes, such as GCN [7] or GraphSAGE [52]. One would welcome works on more complex models from the MP-GNN class, cf. Tables V and VI.
- *Parallelization of GNN models beyond linear ones* Virtually no research exists on parallel and distributed GNN models of polynomial and rational types, cf. Table VII.
- *Parallelization of other GNN settings* Besides very few attempts [117], there is no work on parallelizing heterogeneous GNNs [156], dynamic and temporal GNNs [1], or hierarchical GNNs [17]. 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.
- *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.
- *Incorporating high-performance distributed-memory capabilities* CAGNET [129] 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.
- *System designs for graph predictions* There are few studies and systems focusing on graph-related predictions. For example, developing load balancing schemes for mini-batch parallelism where samples are graphs would be a promising area of study.

VIII. 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.

Acknowledgment

The authors would like to thank Petar Veličković for useful comments. The authors thank Hussein Harake, Colin McMurtrie, Mark Klein, Angelo Mangili, and the whole CSCS team granting access to the Ault and Daint machines, and for their excellent technical support. We thank Timo Schneider for help with computing infrastructure at SPCL. *An extended version: https://arxiv.org/abs/2205.09702*

References

[1] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Jan. 2021.

[2] A. Mirhoseini et al., "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, 2021.

[3] A. Davies et al., "Advancing mathematics by guiding human intuition with AI," *Nature*, vol. 600, no. 7887, pp. 70–74, 2021.

[4] J. Jumper et al., "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021.

[5] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Process. Lett.*, vol. 17, no. 01, pp. 5–20, 2007.

[6] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, 1999.

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

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

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

[10] G. E. Blelloch and B. M. Maggs, *Parallel Algorithms*, London, U.K./Boca Raton, FL, USA: Chapman & Hall/CRC, 2010.

[11] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger, "Simplifying graph convolutional networks," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 6861–6871.

[12] L. Ma et al., "Neugraph: Parallel deep neural network computation on large graphs," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2019, pp. 443–457.

[13] K. Kiningham, P. Levis, and C. Ré, "GReTA: Hardware optimized graph processing for GNNs," in *Proc. Workshop Resour.-Constrained Mach. Learn.*, 2020.

[14] M. Yan et al., "HyGCN: A GCN accelerator with hybrid architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 15–29.

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

[16] 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 *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2020, pp. 45–61.

[17] J. Li, Y. Rong, H. Cheng, H. Meng, W. Huang, and J. Huang, "Semi-supervised graph classification: A hierarchical graph perspective," in *Proc. World Wide Web Conf.*, 2019, pp. 972–982.

[18] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond euclidean data," *IEEE Signal Process. Mag.*, vol. 34, no. 4, pp. 18–42, Jul. 2017.

[19] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, Art. no. 24.

[20] E. Solomonik, E. Carson, N. Knight, and J. Demmel, "Tradeoffs between synchronization, communication, and work in parallel linear algebra computations," Jan. 25, 2014.

[21] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 211–222, 2007.

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

[23] Z. Chen et al., "Bridging the gap between spatial and spectral domains: A unified framework for graph neural networks," 2021, *arXiv:2107.10234*.

[24] P. Veličković, "Theoretical foundations of graph neural networks," 2021. [Online]. Available: https://petar-v.com/talks/GNN-Wednesday.pdf

[25] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1263–1272.

[26] P. W. Battaglia et al., "Relational inductive biases, deep learning, and graph networks," 2018, *arXiv: 1806.01261*.

[27] J. Kepner et al., "Mathematical foundations of the graphblas," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2016, pp. 1–9.

[28] 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 *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 257–266.

[29] W. L. Hamilton, *Graph Representation Learning*, vol. 14. San Rafael, CA, USA: Morgan & Claypool, 2020, pp. 1–159.

[30] J. Thorpe et al., "Dorylus: Affordable, scalable, and accurate GNN training over billion-edge graphs," 2021, *arXiv:2105.11118*.

[31] L. Wang et al., "FlexGraph: A flexible and efficient distributed framework for GNN training," in *Proc. Eur. Conf. Comput. Syst.*, 2021, pp. 67–82.

[32] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, and M. Guo, "Architectural implications of graph neural networks," *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 59–62, Jan.–Jun. 2020.

[33] MPI Forum, "MPI: A message-passing interface standard. Version 3," Sep. 2012. [Online]. Available: http://www.mpi-forum.org

[34] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Comput. Surv.*, vol. 52, 2019, Art. no. 65.

[35] G. Bilardi and A. Pietracaprina, *Models of Computation, Theoretical*, Boston, MA, USA: Springer, 2011, pp. 1150–1158.

[36] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[37] H. Lin et al., "ShenTu: Processing multi-trillion edge graphs on millions of cores in seconds," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2018, Art. no. 56.

[38] W. Hu et al., "Open graph benchmark: Datasets for machine learning on graphs," 2020, *arXiv: 2005.00687*.

[39] A. Buluç et al., "Recent advances in graph partitioning," in, *Algorithm Engineering*, Berlin, Germany: Springer, 2016.

[40] G. Karypis and V. Kumar, "METIS–Unstructured graph partitioning and sparse matrix ordering system, version 2.0 (2nd ed.)," Univ. Minnesota, Dept. Comput. Sci., 1995.

[41] L. Gianinazzi, P. Kalvoda, A. De Palma, M. Besta, and T. Hoefler, "Communication-avoiding parallel minimum cuts and connected components," *ACM SIGPLAN Notices*, vol. 53, pp. 219–232, 2018.

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

[43] Ü. V. Çatalyürek and C. Aykanat, "A fine-grain hypergraph model for 2D decomposition of sparse matrices," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2001, pp. 1199–1204.

[44] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek, "Parallel hypergraph partitioning for scientific computing," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp.*, 2006, pp. 10.

[45] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2012, Art. no. 2.

[46] R. A. Rossi and N. K. Ahmed, "An interactive data repository with visual analytics," *ACM SIGKDD Explorations Newslett.*, vol. 17, no. 2, pp. 37–41, 2016. [Online]. Available: http://networkrepository.com

[47] Q. Su, M. Wang, D. Zheng, and Z. Zhang, "Adaptive load balancing for parallel GNN training," 2021.

[48] S. Mondal et al., "GNNIE: GNN inference engine with load-balancing and graph-specific caching," in *Proc. 59th ACM/IEEE Des. Automat. Conf.*, 2022, pp. 565–570.

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

[50] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Comput. Surv.*, vol. 54, 2021, Art. no. 191.

[51] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu, "Layer-dependent importance sampling for training deep and large graph convolutional networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, Art. no. 1009.

[52] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1024–1034.

[53] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graph-SAINT: Graph sampling based inductive learning method," 2019, *arXiv: 1907.04931*.

[54] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 974–983.

[55] X. Liu, M. Yan, L. Deng, G. Li, X. Ye, and D. Fan, "Sampling methods for efficient training of graph convolutional networks: A survey," *IEEE/CAA J. Automatica Sinica*, vol. 9, no. 2, pp. 205–234, Feb. 2022.

[56] 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," 2022, *arXiv:2203.10428*.

[57] 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," in *Proc. Mach. Learn. Syst.*, 2022, pp. 673–693.

[58] L. Hoang, X. Chen, H. Lee, R. Dathathri, G. Gill, and K. Pingali, "Efficient distribution for deep learning on large graphs," in *Proc. Workshop Graph Neural Netw. Syst.*, 2021.

[59] S. U. Stich, "Local SGD converges fast and communicates little," 2018, *arXiv: 1805.09767*.

[60] Y. Oyama, T. Ben-Nun, T. Hoefler, and S. Matsuoka, "Accelerating deep learning frameworks with micro-batches," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2018, pp. 402–412.

[61] G. Li, M. Müller, B. Ghanem, and V. Koltun, "Training graph neural networks with 1000 layers," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 6437–6449.

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

[63] J. Chen et al., "FastGCN: Fast learning with graph convolutional networks via importance sampling," 2018, *arXiv: 1801.10247*.

[64] M. Wang et al., "Deep graph library: A graph-centric, highly-performant package for graph neural networks," 2019, *arXiv: 1909.01315*.

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

[66] G. E. Blelloch, "Programming parallel algorithms," *Commun. ACM*, vol. 39, no. 3, pp. 85–97, 1996.

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

[68] S. Sukhbaatar et al., "Learning multiagent communication with backpropagation," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 2244–2252.

[69] A. Vaswani et al., "Attention is all you need," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[70] 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 *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 5425–5434.

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

[72] 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 Trans. Graph.*, vol. 38, no. 5, pp. 1–12, 2019.

[73] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2014, pp. 701–710.

[74] M. Defferrard et al., "Convolutional neural networks on graphs with fast localized spectral filtering," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 3837–3845.

[75] J. Atwood and D. Towsley, "Diffusion-convolutional neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 1993–2001.

[76] J. Klicpera, S. Weißenberger, and S. Günnemann, "Diffusion improves graph learning," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 13333–13345.

[77] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 855–864.

[78] Y. Lin, Z. Liu, M. Sun, Y. Liu, and X. Zhu, "Learning entity and relation embeddings for knowledge graph completion," in *Proc. AAAI Conf. Artif. Intell.*, 2015, pp. 2181–2187.

[79] D. Wang, P. Cui, and W. Zhu, "Structural deep network embedding," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 1225–1234.

[80] D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf, "Learning with local and global consistency," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2004, pp. 321–328.

[81] X. Zhu, Z. Ghahramani, and J. D. Lafferty, "Semi-supervised learning using Gaussian fields and harmonic functions," in *Proc. Int. Conf. Mach. Learn.*, 2003, pp. 912–919.

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

[83] A. Bojchevski et al., "Scaling graph neural networks with approximate pagerank," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2020, pp. 2464–2473.

[84] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi, "Graph neural networks with convolutional ARMA filters," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 7, pp. 3496–3507, Jul. 2022.

[85] X.-M. Wu, Z. Li, A. M.-C. So, J. Wright, and S.-F. Chang, "Learning with partially absorbing random walks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 3077–3085.

[86] K. Mulmuley, "A fast parallel algorithm to compute the rank of a matrix over an arbitrary field," in *Proc. 18th Annu. ACM Symp. Theory Comput.*, 1986, pp. 338–339.

[87] E. Georganas, J. Gonzalez-Dominguez, E. Solomonik, Y. Zheng, J. Tourino, and K. Yelick, "Communication avoiding and overlapping for numerical linear algebra," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–11.

[88] 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 *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 463–464.

[89] G. Kwasniewski et al., "Pebbles, graphs, and a pinch of combinatorics: Towards tight I/O lower bounds for statically analyzable programs," in *Proc. 33rd ACM Symp. Parallelism Algorithms Architectures*, 2021, pp. 328–339.

[90] E. Solomonik and T. Hoefler, "Sparse tensor algebra as a parallel programming model," 2015, *arXiv:1512.00066*. [Online]. Available: http://arxiv.org/abs/1512.00066

[91] E. Solomonik, M. Besta, F. Vella, and T. Hoefler, "Scaling betweenness centrality using communication-efficient sparse matrix multiplication," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, Art. no. 47.

[92] N. Gleinig, M. Besta, and T. Hoefler, "I/O-optimal cache-oblivious sparse matrix-sparse matrix multiplication," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2022, pp. 36–46.

[93] D. R. Karger, "Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm," in *Proc. 4th Annu. ACM-SIAM Symp. Discrete Algorithms*, 1993, pp. 21–30.

[94] Y. Shiloach and U. Vishkin, "An O(logn) parallel connectivity algorithm," *J. Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[95] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, Art. no. 10.

[96] D. Narayanan et al., "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 1–15.

[97] S. Li and T. Hoefler, "Chimera: Efficiently training large-scale neural networks with bidirectional pipelines," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, Art. no. 27.

[98] T. Hoefler and D. Moor, "Energy, memory, and runtime tradeoffs for implementing collective communication operations," *Supercomputing Front. Innov.*, vol. 1, pp. 58–75, 2014.

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

[100] S. Masoudnia and R. Ebrahimpour, "Mixture of experts: A literature survey," *Artif. Intell. Rev.*, vol. 42, no. 2, pp. 275–293, 2014.

[101] X. Zhou and Y. Luo, "Explore mixture of experts in graph neural networks," 2019. [Online]. Available: https://snap.stanford.edu/class/cs224w-2019/project/26424363.pdf

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

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

[104] X. Shi et al., "Graph processing on GPUs: A survey," *ACM Comput. Surv.*, vol. 50, 2018, Art. no. 81.

[105] W. Zhang et al., "PaSca: A graph neural architecture search system under the scalable paradigm," in *Proc. ACM Web Conf.*, 2022, pp. 1817–1828.

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

[107] T. Liu et al., "BGL: GPU-efficient GNN training by optimizing graph data I/O and preprocessing," 2021, *arXiv:2112.08541*.

[108] D. Zheng et al., "Distributed hybrid CPU and GPU training for graph neural networks on billion-scale graphs," 2021, *arXiv:2112.15345*.

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

[110] V. Md et al., "DistGNN: Scalable distributed training for large-scale graph neural networks," 2021, *arXiv:2104.06700*.

[111] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "DGCL: An efficient communication library for distributed GNN training," in *Proc. Eur. Conf. Comput. Syst.*, 2021, pp. 130–144.

[112] Y. Wu et al., "Seastar: Vertex-centric programming for graph neural networks," in *Proc. Eur. Conf. Comput. Syst.*, 2021, pp. 359–375.

[113] V. T. Chakaravarthy, S. S. Pandian, S. Raje, Y. Sabharwal, T. Suzumura, and S. Ubaru, "Efficient scaling of dynamic graph neural networks," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, pp. 1–13.
[114] A. Zhou et al., "Optimizing memory efficiency of graph neural networks on edge computing platforms," 2021, *arXiv:2104.03058*.
[115] M. F. Balın et al., "MG-GCN: Scalable multi-GPU GCN training framework," 2021, *arXiv:2110.08688*.
[116] Y. Wang et al., "GNNAdvisor: An efficient runtime system for GNN acceleration on GPUs," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2021, pp. 515–531.
[117] R. Zhu et al., "AliGraph: A comprehensive graph neural network platform," 2019, *arXiv: 1902.08730*.
[118] T. Kim et al., "Accelerating GNN training with locality-aware partial execution," in *Proc. 12th ACM SIGOPS Asia-Pacific Workshop Syst.*, 2021, pp. 34–41.
[119] D. Zhang et al., "AGL: A scalable system for industrial-purpose graph machine learning," 2020, *arXiv: 2003.02454*.
[120] Z. Jia et al., "Improving the accuracy, scalability, and performance of graph neural networks with ROC," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2020, pp. 187–198.
[121] D. Zheng et al., "DistDGL: Distributed graph neural network training for billion-scale graphs," in *Proc. IEEE/ACM 10th Workshop Irregular Appl. Architectures Algorithms*, 2020, pp. 36–44.
[122] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "PaGraph: Scaling GNN training on large graphs via computation-aware caching," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 401–415.
[123] Y. Bai et al., "Efficient data loader for fast sampling-based GNN training on large graphs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 10, pp. 2541–2556, Oct. 2021.
[124] L. Zhang, Z. Lai, S. Li, Y. Tang, F. Liu, and D. Li, "2PGraph: Accelerating GNN training over large graphs on GPU clusters," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2021, pp. 103–113.
[125] W. Zhang et al., "GMLP: Building scalable and flexible graph neural networks with feature-message passing," 2021, *arXiv:2104.09880*.
[126] Z. Chen et al., "fuseGNN: Accelerating graph convolutional neural network training on GPGPU," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des.*, 2020, pp. 1–9.
[127] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2021, pp. 551–568.
[128] Y. Wang, B. Feng, and Y. Ding, "QGTC: Accelerating quantized GNN via GPU tensor core," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2022, pp. 107–119.
[129] A. Tripathy, K. Yelick, and A. Buluç, "Reducing communication in graph neural network training," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, Art. no. 70.
[130] C. Tian, L. Ma, Z. Yang, and Y. Dai, "PCGCN: Partition-centric processing for accelerating graph convolutional network," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2020, pp. 936–945.
[131] Y. Hu et al., "FeatGraph: A flexible and efficient backend for graph neural network systems," 2020, *arXiv: 2008.11359*.
[132] H. Liu, S. Lu, X. Chen, and B. He, "G3: When graph neural networks meet parallel graph processing systems on GPUs," in *Proc. VLDB Endowment*, vol. 13, pp. 2813–2816, 2020.
[133] S. W. Min et al., "Large graph convolutional network training with GPU-oriented data communication architecture," 2021, *arXiv:2103.03330*.
[134] Z. Zhang et al., "ZIPPER: Exploiting tile-and operator-level parallelism for general and scalable graph neural network acceleration," 2021, *arXiv:2107.08709*.
[135] Z. Zhou, C. Li, X. Wei, and G. Sun, "GCNear: A hybrid architecture for efficient GCN training with near-memory processing," 2021, *arXiv:2111.00680*.
[136] Z. Zhou, B. Shi, Z. Zhang, Y. Guan, G. Sun, and G. Luo, "Blockgnn: Towards efficient GNN acceleration using block-circulant weight matrices," in *Proc. 58th ACM/IEEE Des. Automat. Conf.*, 2021, pp. 1009–1014.
[137] Y. He, Y. Wang, C. Liu, H. Li, and X. Li, "TARe: Task-adaptive in-situ ReRAM computing for graph learning," in *Proc. 58th IEEE/ACM Des. Automat. Conf.*, 2021, pp. 577–582.
[138] X. Chen et al., "Rubik: A hierarchical architecture for efficient graph neural network training," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 4, pp. 936–949, Apr. 2022.
[139] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2021, pp. 775–788.
[140] H. Li et al., "Hardware acceleration for GCNs via bidirectional fusion," *IEEE Comput. Archit. Lett.*, vol. 20, no. 1, pp. 66–4, Jan.–Jun. 2021.
[141] Z. Wang et al., "GNN-PIM: A processing-in-memory architecture for graph neural networks," in *Proc. Conf. Adv. Comput. Architecture*, 2020, pp. 73–86.
[142] S. Liang et al., "EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Trans. Comput.*, vol. 70, no. 9, pp. 1511–1525, Sep. 2021.
[143] T. Geng et al., "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *Proc. IEEE/ACM 53rd Annu. Int. Symp. Microarchitecture*, 2020, pp. 922–936.
[144] K. Kiningham, C. Re, and P. Levis, "Grip: A graph neural network accelerator architecture," 2020, *arXiv: 2007.13828*.
[145] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale GCN inference," in *Proc. IEEE 31st Int. Conf. Appl.-Specific Syst. Architectures Processors*, 2020, pp. 61–68.
[146] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2020, pp. 255–265.
[147] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *Proc. 57th ACM/IEEE Des. Automat. Conf.*, 2020, pp. 1–6.
[148] Y. Wang et al., "Gunrock: A high-performance graph processing library on the GPU," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2016, pp. 11:1–11:12.
[149] M. K. Rahman, M. H. Sujon, and A. Azad, "FusedMM: A unified SDDMM-SpMM kernel for graph embedding and graph neural networks," in *Proc. Int. Parallel Distrib. Process. Symp.*, 2021, pp. 256–266.
[150] S. Qiu, Y. Liang, and Z. Wang, "Optimizing sparse matrix multiplications for graph neural networks," 2021, *arXiv:2111.00352*.
[151] G. Huang, G. Dai, Y. Wang, and H. Yang, "GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–12.
[152] H. Zhang et al., "Understanding GNN computational graph: A coordinated computation, IO, and memory perspective," in *Proc. Int. Conf. Mach. Learn. Syst.*, 2022, pp. 467–484.
[153] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms," in *Proc. Eur. Conf. Parallel Process.*, 2011, pp. 90–109.
[154] R. Garg et al., "Understanding the design space of sparse/dense multiphase dataflows for mapping graph neural networks on spatial accelerators," 2021, *arXiv:2103.07977*.
[155] A. Tate et al., "Programming abstractions for data locality," in *Proc. Program. Abstractions Data Locality Workshop*, 2014, pp. 1–50.
[156] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 793–803.

Maciej Besta is a researcher with ETH Zurich. He works on understanding and accelerating large-scale irregular computations, such as graph streaming, graph neural networks, or graph databases, at all levels of the computing stack.

Torsten Hoefler is a professor with ETH Zurich, where he leads the Scalable Parallel Computing Lab. His research aims at understanding performance of parallel computing systems ranging from parallel computer architecture through parallel programming to parallel algorithms.