An Experimental Comparison of Partitioning Strategies for Distributed Graph Neural Network Training

Nikolai Merkel
Technical University of Munich
nikolai.merkel@tum.de

Daniel Stoll
Technical University of Munich
daniel.stoll@tum.de

Ruben Mayer
University of Bayreuth
ruben.mayer@uni-bayreuth.de

Hans-Arno Jacobsen
University of Toronto
jacobsen@eecg.toronto.edu

ABSTRACT

Recently, graph neural networks (GNNs) have gained much attention as a growing area of deep learning capable of learning on graph-structured data. However, the computational and memory requirements for training GNNs on large-scale graphs make it necessary to distribute the training. A prerequisite for distributed GNN training is to partition the input graph into smaller parts that are distributed among multiple machines of a compute cluster. Although graph partitioning has been studied with regard to graph analytics and graph databases, its effect on GNN training performance is largely unexplored. As a consequence, it is unclear whether investing computational efforts into high-quality graph partitioning would pay off in GNN training scenarios.

In this paper, we study the effectiveness of graph partitioning for distributed GNN training. Our study aims to understand how different factors such as GNN parameters, mini-batch size, graph type, features size, and scale-out factor influence the effectiveness of graph partitioning. We conduct experiments with two different GNN systems using vertex and edge partitioning. We found that high-quality graph partitioning is a very effective optimization to speed up GNN training and to reduce memory consumption. Furthermore, our results show that invested partitioning time can quickly be amortized by reduced GNN training time, making it a relevant optimization for most GNN scenarios. Compared to research on distributed graph processing, our study reveals that graph partitioning plays an even more significant role in distributed GNN training, which motivates further research on the graph partitioning problem.

1 INTRODUCTION

The management and processing of graph-structured data has been in the focus of research in academia and industry for many decades [9, 13, 14, 17, 27]. Graphs are an excellent way of modeling real-world phenomena that focus on the interactions and relations between entities [35]. Recently, graph neural networks (GNNs) have emerged as a new category of machine learning models which are specialized for learning on graph-structured data. GNNs have successfully been applied to domains such as recommendation systems [10, 50], natural language processing [23], drug discovery [25] and fraud detection [26]. Despite their success, GNN training is challenging due to computationally intensive deep neural network operations and high memory requirements. Therefore, GNNs are commonly trained in a distributed fashion. In order to enable distributed training, the input graph needs to be *partitioned* into a predefined number of equally-sized parts which are distributed among different machines such that the cut size is minimized.

Many graph partitioning approaches exist. In edge partitioning [19, 30–32, 40, 49, 51], edges are assigned to partitions while in vertex partitioning [21, 29, 41, 44, 52], vertices are assigned to partitions. In the past decade, different works [2, 12, 37, 48] have studied the effect of graph partitioning on the performance of distributed graph processing systems such as Pregel [27], PowerGraph [13], PowerLyra [9] or GraphX [14] for classical graph analytics workloads such as PageRank, Connected Components and Shortest Paths. In these systems, a vertex function is iteratively executed for each vertex to compute state updates which are propagated along the edges to the vertices' neighbors [27]. Often the messages and vertex states are small, e.g., consisting of a single numerical value and light-weight vertex functions such as computing the maximum value or sum of values. Compared to GNN training, these workloads are light-weight in terms of computation and memory overhead, and are executed on the graph structure only. GNN training has unique features such as heavy-weight neural network operations, intensive communication of high dimensional feature vectors and large intermediate embeddings, and a large memory footprint to store intermediate results for each layer during forward and backward propagation. Further, GNNs have additional parameters such as number of layers, number of hidden dimensions or mini-batch size which influence memory, computation and communication overhead.

The effectiveness of graph partitioning has not been investigated for distributed GNN training. As a consequence, it is unclear under which conditions it is beneficial to invest computational resources to yield a high-quality graph partitioning. In this paper, we perform an extensive experimental analysis of graph partitioning for distributed GNN training to close this gap and make the following contributions:

(1) Based on extensive evaluations with two prominent state-of-the-art distributed GNN systems (*DistGNN* [33] and *DistDGL* [53]), 12 graph partitioning algorithms (edge partitioning and vertex partitioning), different GNN model architectures, different GNN parameters, and graphs of various categories with different feature sizes, we find that high-quality graph partitioning is effective for GNN training. Compared to random partitioning, e.g., the *HEP100* partitioner [31] leads to speedups of up to 10.41x (on average 4.36x) on *DistGNN* and the *KaHIP* partitioner [41] leads to speedups of up to 3.47x (on average 1.37x) on *DistDGL*. Further, the memory footprint can be decreased by up to 85.1% and on average by 51.4% if *HEP100* is used for partitioning which is crucial for memory-bound GNN training. This shows the enormous potential of graph partitioning for GNN workloads.

© 2025 Copyright held by the owner/author(s). Published in Proceedings of the 28th International Conference on Extending Database Technology (EDBT), 25th March-28th March, 2025, ISBN 978-3-98318-097-4 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

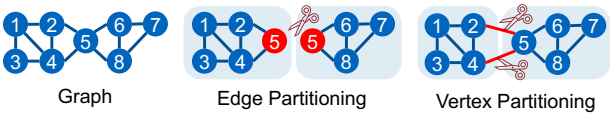


Figure 1: Edge partitioning vs. vertex partitioning.

(2) We show that partitioning quality properties such as the replication factor or vertex balance influence the GNN training a lot. We find a strong correlation between replication factor, network communication and memory footprint. Therefore, minimizing the replication factor is crucial for efficient distributed GNN training. We show that vertex imbalance can decrease the speedup and can also lead to severe imbalances regarding memory utilization. This motivates the need for further research on balanced graph partitioning algorithms.

(3) We find that GNN parameters such as the hidden dimension, the number of layers, the mini-batch size and the feature size influence the effectiveness of graph partitioning, both in terms of training time and memory overheads. Our experiments further show that a higher scale-out factor can decrease the effectiveness of vertex partitioning, while the effectiveness increases for edge partitioning. These insights help to understand the interplay between GNN parameters and partitioner effectiveness.

(4) We find that invested partitioning time can be amortized by faster GNN training in typical scenarios, making graph partitioning relevant for production systems. This is different from classical graph processing, where graph partitioning can often not be amortized and fast streaming-based graph partitioning performs best [34].

Our paper is organized as follows. In Section 2, we introduce graph partitioning and graph neural networks. In Section 3, we describe our methodology. Then, we analyze the results for DistGNN in Section 4 and for DistDGL in Section 5. In Section 6, we summarize our main findings and in Section 7 we discuss related work. Finally, we conclude our paper in Section 8.

2 BACKGROUND

Let $G = (V, E)$ be a graph consisting of a set of vertices V and a set of edges $E \subseteq V \times V$. $N(v)$ represents the set of vertices that are connected to v. In the following, we discuss graph partitioning in Section 2.1 and distributed GNN training in Section 2.2.

2.1 Graph Partitioning

The main approaches for graph partitioning are *edge partitioning* and *vertex partitioning* (see Figure 1). In the following, we present both approaches and commonly used partitioning quality metrics.

Edge Partitioning. In edge partitioning (vertex-cut), the set of edges E is divided into k partitions by assigning each edge to exactly one partition $p \in P = \{p_1, \ldots, p_k\}$ with $\cup_{i=1}^{k} p_i = E$. Through this process, vertices can be cut. A cut vertex is replicated to all partitions that have adjacent edges. Each partition p_i covers a set of vertices $V(p_i) = \{v \in V \mid (\exists e = (u, v) \in p_i) \vee (\exists e = (v, u) \in p_i)\}$. The goal of edge partitioning [51] is to minimize the number of cut vertices while keeping the partitions' edges α-balanced, meaning $\forall p_i \in P : |p_i| \leq \alpha \cdot \frac{|E|}{k}$.

Commonly used quality metrics to evaluate edge partitioners are the mean *replication factor* and *edge balance*. The replication factor $RF(P)$ is defined as $\frac{1}{|V|} \sum_{i=1}^{k} |V(p_i)|$ and represents the average number of partitions to which vertices are replicated. This metric is closely related to communication cost because replicated vertices need to synchronize their state via the network. Additionally, it indicates memory overhead, as for each replicated vertex, the state is replicated. This becomes particularly significant in the context of GNN training where vertex states are substantial in size, a concern less emphasized in classical graph processing, where memory constraints are not as pivotal. The edge balance is defined as $EB(P) = \frac{max(\{|p_1|,\ldots,|p_k|\})}{mean(\{|p_1|,\ldots,|p_k|\})}$, and the vertex balance as $VB(P) = \frac{max(\{|V(p_i)|,\ldots,|V(p_i)|\})}{mean(\{|V(p_i)|,\ldots,|V(p_i)|\})}$. Most edge partitioners do not explicitly balance vertices because the computational load of many graph algorithms is believed to be proportional to the number of edges as messages are aggregated along the edges (e.g., PageRank).

Vertex Partitioning. In vertex partitioning (edge-cut), the set of vertices V is divided into k partitions by assigning each vertex v to exactly one partition $p \in P = \{p_1, \ldots, p_k\}$ with $\cup_{i=1}^{k} p_i = V$. An edge $e = (u, v)$ is cut if both u and v are assigned to different partitions. Vertex partitioning aims to minimize the number of cut edges while balancing the partition sizes in terms of number of vertices. We define E_{cut} as the set of cut edges.

Commonly used vertex partitioning quality metrics to evaluate vertex partitioners are the *edge-cut ratio* and *vertex balance*. The edge-cut ratio is defined as $\lambda = \frac{|E_{cut}|}{|E|}$ and indicates communication costs, as messages are sent via edges, and cut edges lead to network communication between machines. The vertex balance is defined as $VB(P) = \frac{max(\{|p_1|,\ldots,|p_k|\})}{mean(\{|p_1|,\ldots,|p_k|\})}$ and indicates computation balance.

Partitioner Types. Both edge and vertex partitioning algorithms can be categorized into (1) *streaming partitioners*, which stream the graph and directly assign vertices or edges to partitions [30, 32, 40, 44, 49]. Streaming partitioners can be further divided into *stateless* partitioners, which do not keep any state, and *stateful* streaming partitioners, which use some state, e.g., the current load per partition or to which partition vertices or edges were assigned. The state is considered for the assignments. (2) *In-memory partitioners* which load the complete graph into the memory [16, 21, 29, 41, 51]. For edge partitioning, there exists also a *hybrid partitioner* that partitions one part with an in-memory partitioner and the remaining part with a streaming partitioner [31].

2.2 Graph Neural Network Training

Graph neural networks (GNNs) are a class of neural networks which operate on graph-structured data. GNNs iteratively learn on graphs by aggregating the local neighborhoods of vertices. At start, each vertex v is represented by its feature vector $h_v^{(0)}$. In each layer, a vertex v aggregates learned representations of its neighbors $N(v)$ of the previous layers, resulting in $a_v^{(k)} = AGGREGATE^{(k)}(\{h_u^{(k-1)} | u \in N(v)\})$. Based on $a_v^{(k)}$ and its previous intermediate representation $h_v^{(k-1)}$ in layer $k-1$, vertex v updates its representation to $h_v^{(k)} = UPDATE^{(k)}(a_v^{(k)}, h_v^{(k-1)})$.

The main approaches to train GNNs are *full-batch* and *mini-batch* training [3]. In full-batch training, the entire graph is used to update the model once per epoch. In mini-batch training, each epoch contains multiple iterations, where a mini-batch is sampled from the graph and used for training and model update.

Table 1: Graphs of different types. *Dir.* indicates if the graph is directed or undirected.

Graph	Type	Dir.	|E|	|V|
Hollywood-2011 (*HO*) [5, 6]	Colla.	no	229M	2M
Dimacs9-USA (*DI*) [22]	Road	yes	58M	24M
Enwiki-2021 (*EN*) [5, 6]	Wiki	yes	150M	6M
Eu-2015-tpd (*EU*) [4–6]	Web	yes	166M	7M
Orkut (*OR*) [24]	Social	no	234M	3M

3 EXPERIMENTAL METHODOLOGY

Our research aims to investigate the effect of graph partitioning on the performance of distributed GNN training. We want to answer the following five research questions:

- **RQ-1** How effective is graph partitioning for distributed GNN training in reducing training time and memory footprint?
- **RQ-2** Do classical partitioning quality metrics accurately describe the effectiveness of partitioning algorithms for distributed GNN workloads? Which partitioning quality metric is most crucial?
- **RQ-3** How much is the partitioning effectiveness influenced by GNN parameters such as the *number of layers*, *hidden dimension*, *feature size*, *mini-batch size*, and *type of graph*?
- **RQ-4** What is the impact of the scale-out factor (number of machines used for training) on the partitioning effectiveness?
- **RQ-5** Can the invested partitioning time be amortized by a reduced GNN training time?

To address these research questions, we conduct various experiments with two distributed GNN systems *DistGNN* [33] and *DistDGL* [53]. Both systems are extensions of the *Deep Graph Library (DGL)*, which is a leading framework for training GNNs on single machines. Consequently, *DistDGL* and *DistGNN* are based on a shared codebase. Both systems add functionality for distributed training with *DistDGL* focusing on mini-batch training and *DistGNN* on full-batch training. A key distinction between the two systems is their support for different graph partitioning methods: *DistGNN* utilizes edge partitioning (vertex-cut), whereas *DistDGL* employs vertex partitioning (edge-cut). In contrast to *DistGNN*, *DistDGL* is much more mature, well-documented, highly optimized, still under active development, and also used in industry.

In the following, we introduce the datasets and metrics we use to evaluate both systems.

Datasets. We selected five graphs (see Table 1) from the following different categories (1) web, (2) social, (3) collaboration, (4) road, and (5) wiki to investigate the influence of the graph type on the partitioning effectiveness. Compared to large-scale distributed graph processing, all datasets only have a medium-sized graph structure. However, GNN workloads are much more communication, memory, and compute-intensive because the vertices have large feature vectors, and the computation includes heavy-weight neural network computations with large intermediate states. Hence, graphs of this size already require heavy-weight processing for GNN training. In order to systematically investigate the influence of the feature size on the partitioning effectiveness, we set the feature size to commonly used values between 16 and 512.

Table 2: Different GNN hyper-parameters for both systems.

Hyper-parameter	Values
Hidden Dimension	16, 64, 512
Feature size	16, 64, 512
Number of layers	2, 3, 4

Infrastructure and infrastructure metrics. We use a cluster composed of 32 machines. Each machine is equipped with 64GB memory and 8 CPU cores (Intel Core Haswell, 2.4 GHz). We measure the CPU and memory utilization and network traffic of each worker.

Performance metrics. We compare the effectiveness of graph partitioners in reducing GNN training time based on the achieved speedup over random partitioning. The speedup of a graph partitioner p is defined as $speedup(p) = \frac{T_{random}}{T_p}$ with T_p and T_{random} being the GNN training time of partitioner p and random partitioning, respectively.

4 DISTGNN - EDGE PARTITIONING

4.1 Experiments

Graph partitioning algorithms. We use six state-of-the-art edge partitioners from different categories: (1) stateless streaming partitioning with *random partitioning* and *DBH* [49], (2) stateful streaming with *HDRF* [40] and *2PS-L* [32], and (3) hybrid-partitioning with *HEP* [31]. *HEP* uses a parameter τ which influences how much of the graph is partitioned in a streaming fashion and how much in memory. A larger value for τ leads to a better partitioning quality as a larger part of the graph is partitioned in memory. As suggested by the authors, we set τ to 10 and to 100, and we treat these two configurations as two different partitioners *HEP10* and *HEP100* in our evaluations. *HEP100* corresponds to in-memory partitioning, as the part not loaded into the memory is negligible if τ is set to 100.

Workloads. DistGNN currently only supports GraphSage [15], one of the most common model architectures [18, 33, 45, 53]. We reviewed the GNN literature to identify commonly used hyper-parameters and chose the following to cover the ranges accordingly. We vary the *hidden dimension* from 16 to 512, the *number of layers* from 2 to 4, and the *feature size* from 16 to 512 (see Table 2).

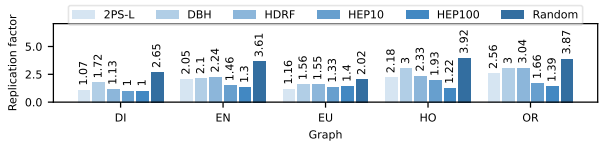
Partitioning metrics. We measure the well-known edge partitioning quality metrics *replication factor*, *edge balance* and *vertex balance* (see Section 2.1).

Training & partitioning time. We measure the time per epoch to compute the speedup over random partitioning (see. Section 3). Furthermore, we measure the partitioning time.

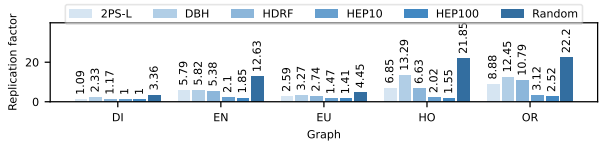
4.2 Partitioning Performance

We compare the partitioning algorithms regarding their communication costs and computational balance in the following.

(1) Communication costs. We observe significant differences in terms of replication factors for the different partitioners. In all cases, *HEP100* leads to the lowest (best) replication factor and *Random* to the largest (worst) one. Figure 2b shows, for example, that *HEP100* at 32 partitions leads to a replication factor of 2.52 on *OR*, which is much smaller compared to *Random* which leads to a replication factor of 22.2. In general, more partitions lead to larger replication factors. For some partitioners, the replication factors increase more sharply than for others if the number of partitions

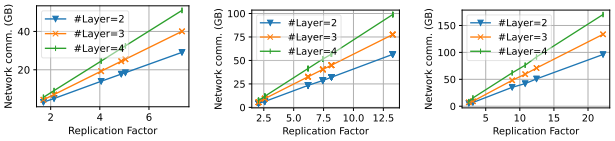


(a) 4 partitions.



(b) 32 partitions.

Figure 2: Replication factors.



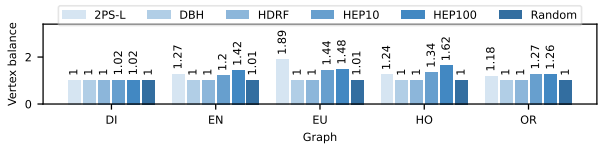
(a) 8 machines. (b) 16 machines. (c) 32 machines.

Figure 3: Replication factor vs. network communication for different number of machines and number of layers on *OR*.

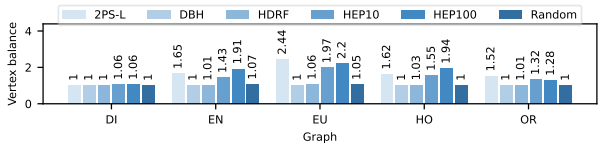
increases. We observe a strong correlation ($R^2 \geq 0.98$) between the replication factor and network traffic. This correlation is shown in Figure 3 for *OR* for different numbers of machines and number of layers and is also observed for the remaining graphs. The observation is plausible. The higher the replication factor, the more data is communicated via the network because more vertices are replicated and must synchronize their states. We conclude that **minimizing the replication factor is crucial for reducing network overhead**.

(2) Computational balance. It is crucial to balance the number of edges and vertices per partition. Each edge leads to an aggregation in the GNN, and neural network operations are performed for vertices. We observe a good edge balance of at most $\alpha \leq 1.11$ for all partitioners. However, we observe significant vertex imbalances (see Figure 4). Especially, the partitioners *2PS-L*, *HEP10* and *HEP100* lead to large vertex imbalances between 1.18 and 1.89 on 4 machines (see. Figure 4a) and can even increase up to 2.44 on 32 machines (see. Figure 4b). The vertex imbalance has a significant influence on the balance of memory utilization. Figure 5 reports the imbalance regarding memory utilization for all combinations of *feature size*, *hidden dimension*, and *number of layers* per graph and graph partitioner. We observe that vertex imbalance correlates with memory utilization imbalance. We conclude that **minimizing vertex imbalance is crucial for balancing memory utilization**.

(3) Partitioning time. Figure 6 reports the partitioning time for partitioning all graphs into 4 and 32 partitions. We observe that the partitioning times of some algorithms, e.g., *Random*, *2PS-L* and *DBH*, are less dependent on the number of partitions, compared to the remaining partitioners, where more partitions lead to higher partitioning times. For example, *HDRF* takes much more time to partition the graphs into 32 partitions compared



(a) 4 partitions.



(b) 32 partitions.

Figure 4: Vertex balance.

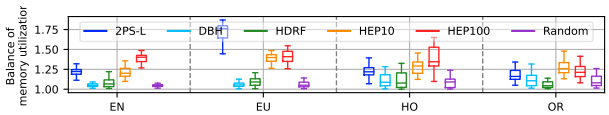
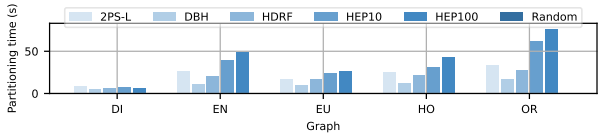
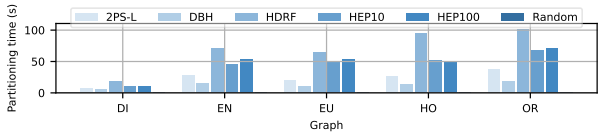


Figure 5: Memory utilization balance (4 machines).

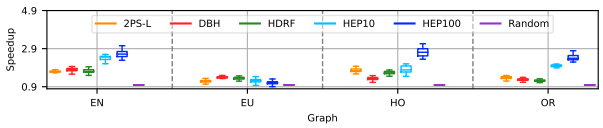


(a) 4 partitions.

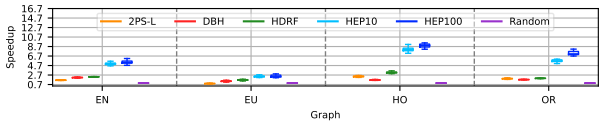


(b) 32 partitions.

Figure 6: Partitioning time.



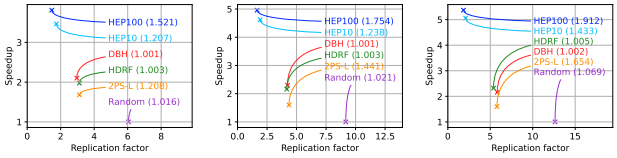
(a) 4 machines.



(b) 32 machines.

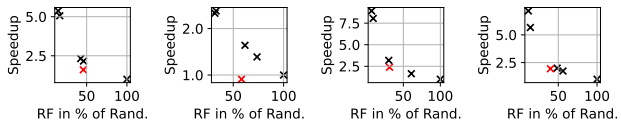
Figure 7: Speedup distribution of graph partitioners on 4 and 32 machines for all experiments.

to 4 partitions, which is expected because the complexity of the scoring function depends on the number of partitions.



(a) 8 machines. (b) 16 machines. (c) 32 machines.

Figure 8: Replication factor vs. speedup on *EN* along with the vertex balance in the brackets.



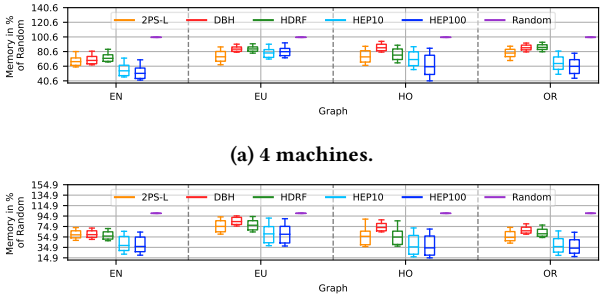
(a) EN. (b) EU. (c) HO. (d) OR.

Figure 9: Replication factor (RF) in % of random partitioning (Rand.) vs. speedup on 32 machines. *2PS-L* is highlighted red.

4.3 GNN Training Performance

In Figure 7, we report the speedup distribution for all partitioners compared to random partitioning for all combinations of *feature size*, *hidden dimension*, and *number of layers* (see Table 2).

We observe that graph partitioning is effective in reducing the training time. *HEP100* leads to the largest speedups of up to 3.53x, 6.18x, 8.15x and 10.41x on the graphs *EU*, *EN*, *OR* and *HO*, respectively. However, the effectiveness of the partitioners differs. In most cases, the more lightweight streaming-based partitioners *DBH*, *2PS-L* and *HDRF* yield smaller speedups compared to more heavy-weight partitioning with *HEP10* and *HEP100*. For example, on *OR*, compared to *Random*, on average *DBH*, *2PS-L*, *HDRF*, *HEP10* and *HEP100* lead to speedups of 1.40x, 1.46x, 1.44x, 2.96x and 3.68x on 8 machines, 1.62x, 1.61x, 1.75x, 4.37x and 7.16x on 16 machines and 1.74x, 1.95x, 2.00x, 5.67x and 7.16x on 32 machines. Further, we observe that the effectiveness of the partitioners is influenced by the graph type, e.g., the speedups for the web graph *EU* which has a lower density than the remaining graphs, are smaller for all partitioners, compared to the remaining graph types. We also observe for *EU*, that the partitioners are less effective in reducing the replication factor over random partitioning and that the replication factors are lower - even under random partitioning - compared to the other graphs. In the Figures 9a-9d, we plot the replication factor in percent of random partitioning against the speedup for *EN*, *EU*, *HO* and *OR*, respectively. Each marker represents a graph partitioner, and *2PS-L* is highlighted in red. A trend emerges from the data: a lower replication factor relative to the replication factor of random partitioning correlates with a higher speedup. Consequently, it's evident that to achieve large speedups, it's vital to minimize the replication factor. Nonetheless, exceptions such as *2PS-L* on *EU* (see Figure 9b) do exist. Such deviations are often due to a significant vertex imbalance. The vertex imbalance leads to slowdowns for *2PS-L* of 0.92x, 0.92x, and 0.91x on 8, 16 and 32 machines on *EU*. Another example can be seen in Figure 8. *2PS-L* leads to a similar replication factor as *HDRF* and *DBH* on *EN*. However, in contrast to *HDRF* and *DBH* which are perfectly balanced, *2PS-L* leads to a large vertex imbalance resulting in a smaller speedup.



(a) 4 machines.

(b) 32 machines.

Figure 10: Distribution of memory footprint of graph partitioners in % of random partitioning on 4 and 32 machines.

Figure 10 gives an overview of how much memory is needed for training with different partitioners and graphs in percent of random partitioning for all GNN parameter combinations *feature size*, *hidden dimension*, and *number of layers*. We make two main observations: (1) The high-quality partitioners (*HEP10* and *HEP100*) are more effective in reducing memory footprint than the other partitioners. (2) For each combination of graph partitioner and graph in Figure 10, we observe large deviations, indicating that the partitioners' effectiveness depends on the GNN parameters. In the following, we first discuss why the partitioners differ in their effectiveness in terms of memory footprint (observation 1) and then analyze how the partitioners' effectiveness depends on the GNN parameters (observation 2).

We observe a strong correlation ($R^2 \geq 0.99$) between the replication factor and memory footprint. Further, we observe that the memory footprint can heavily decrease, e.g., *HEP100* reduces the memory for the graphs *EU*, *OR*, *HO*, *EN* by 37% 53%, 56% and 60% on 8 machines, by 44%, 60%, 65% and 63% on 16 machines and by 40%, 67%, 66% and 63% on 32 machines, respectively compared to *Random*. There are also cases where random partitioning leads to out-of-memory errors. For example, in many cases, *DI* can not be processed if random partitioning is applied, but in contrast, the more advanced partitioners enable the processing in such cases. In classical distributed graph processing, the replication factor is often minimized with the primary goal of reducing network communication. The memory load is less critical, especially if the vertex state is small which is the case for many graph processing algorithms such as Connected Components, PageRank, and K-cores. In contrast, in GNNs, the vertex state consists of large feature vectors meaning that the vertex state, not the graph structure, dominates the required memory. We conclude that **minimizing the replication factor is crucial for minimizing the memory overhead and can be decisive for GNN training.**

In the following, we analyze how the different GNN parameters influence the effectiveness of the partitioners **in terms of memory**.

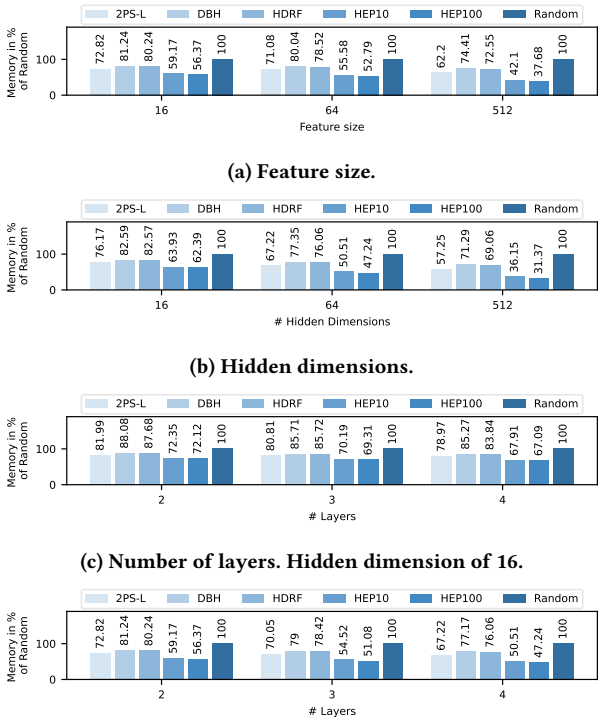
(1) Feature size. In Figure 11a, we report the memory footprint for all partitioners in percent of random partitioning dependent on the feature size. We observe that if we keep all other parameters constant, graph partitioning is more effective in the face of larger feature sizes. For example, *HEP*10 has a memory footprint of 59.17% of *Random* for a feature size of 16, and a memory footprint of only 42.1% of *Random* for a higher feature size of 512. This result seems plausible. A fixed amount of memory is needed,

e.g., for storing the graph structure which is not dependant on the feature size for all partitioners. In addition, the features need to be stored, and with an increasing feature size, the memory footprint for storing the features increases. Therefore, the better the partitioning (in terms of less replications of feature vectors), the higher the memory savings. We conclude that **graph partitioning is more effective in reducing the memory footprint in the face of larger feature sizes.**

(2) Hidden dimension. If the feature size and number of layers are kept constant, there will be some fixed amount of memory for the graph structure, the corresponding features, and the replication of the features. We observe that the higher the hidden dimension, the more effective the partitioners become (see Figure 11b). For example, *2PS-L* has memory footprint of 76.17% and 57.25% of *Random* for a hidden dimension of 16 and 512, respectively. This observation seems plausible. Larger hidden sizes lead to larger intermediate representations which need to be replicated to other machines as they are needed as the input of the next layer. We conclude that **graph partitioning is more effective in reducing the memory footprint in the face of larger hidden dimensions.**

(3) Number of layers. Our experiments show that the number of layers can influence the effectiveness of graph partitioning. The higher the number of layers, the more intermediate representations need to be stored (one per vertex and layer) which are needed in the backward pass. The hidden dimension influences the size of the intermediate representations. The higher the replication factor, the higher the number of machines to which the intermediate representations are replicated. We observe that graph partitioning becomes more effective as the number of layers increases. However, there is also an interesting reinforcing effect between the number of layers and the size of the hidden dimension: With a large hidden dimension, the memory saving of graph partitioning under a growing number of layers becomes even more significant. For example, the memory footprint of *HEP100* in percent of random partitioning is reduced from 72.12% for 2 layers to 67.09% for 4 layers if the hidden dimension is 16 (See. Figure 11c) and is reduced from 56.37% for 2 layers to 47.24% for 4 layers if the hidden dimension is 64 (See. Figure 11d). We conclude, **the more layers, the more important the partitioning.**

(4) Scale-out factor. In the following, we analyze the effectiveness of graph partitioning in terms of speedup and memory footprint across various scale-out factors. Figure 12a shows the average speedup for all graph partitioners at different scale-out factors compared to random partitioning. We observe that the effectiveness of all graph partitioners increases if more machines are used for training. However, there are differences in how sharply the effectiveness increases. For the more light-weight partitioners *2PS-L*, *DBH* and *HDRF*, the speedups increase moderately from 1.57x, 1.37x, and 1.49x on 4 machines to 1.79x, 1.7x, and 2.06x on 32 machines, respectively. The partitioners *HEP10* and *HEP100* lead to a speedup of 1.95x and 2.47x on 4 machines which increase sharply to 5.41x and 6.77x on 32 machines, respectively. We make similar observations for the memory overhead (see Figure 12b). All partitioners lead to substantial savings which increase if the number of machines increases. Both observations are plausible: In Figure 12c, we report the achieved replication factors for all partitioners for all scale-out factors in percentages of *Random*, meaning lower numbers are better. We find that the replication factor of all partitioners increases less sharply than *Random* if the scale-out factor increases. We observe that the



(a) Feature size.

(b) Hidden dimensions.

(c) Number of layers. Hidden dimension of 16.

(d) Number of layers. Hidden dimension of 64.

Figure 11: Memory footprint in % of random partitioning dependant on different GNN parameters for *OR* on 8 machines.

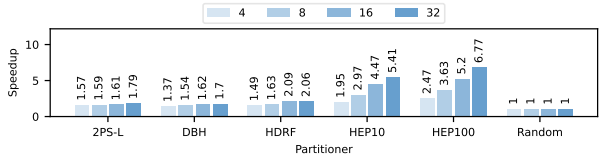
replication factors of *2PS-L*, *DBH* and *HDRF* are 56.74%, 76.49% and 62.16% of *Random* on 4 machines and 39.99%, 60.81% and 48.58% of *Random* on 32 machines. *HEP10* and *HEP100* achieve a replication factor of 49.27% and 36.05% of *Random* on 4 machines and significantly lower replication factors of 14.05% and 11.37% of *Random* on 32 machines, respectively. We conclude that **graph partitioning is more effective in reducing training time and memory footprint in the face of larger scale-out factors.**

(5) Partitioning time amortization. In Table 3, we report the average number of epochs until the partitioning time is amortized by faster training time for each combination of graph and partitioner. We assume that random partitioning does not take any time. *DBH* is the partitioner that amortizes the fastest: on average, it takes 1.39, 3.79, 3.05, and 3.83 epochs on the graphs *EN*, *EU*, *HO* and *OR* to amortize the partitioning time. *HEP100* which leads to the largest speedups amortizes after 4.29, 12.0, 4.7, and 7.03 epochs on the graphs *EN*, *EU*, *HO* and *OR*. Training is often performed for hundreds of epochs [33], **therefore, the partitioning time can be amortized**. In addition, a hyper-parameter search is often performed which requires even more training epochs. Therefore, it is even more beneficial to invest time into partitioning.

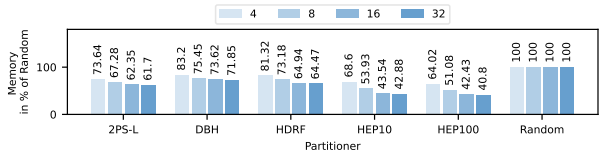
5 DISTDGL - VERTEX PARTITIONING

5.1 Experiments

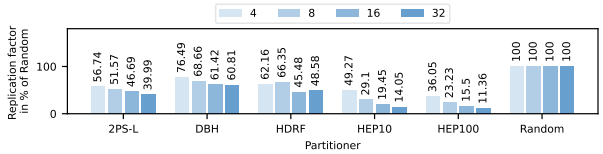
Graph partitioning algorithms. We use six state-of-the-art vertex partitioners from different categories: (1) stateless streaming partitioning with random partitioning, (2) stateful streaming with



(a) Speedup (larger is better).



(b) Memory overhead (lower is better).



(c) Replication factor (lower is better).

Figure 12: The effectiveness of graph partitioning increases in terms of speedup and memory overhead if the scale-out factor is increased from 4 to 8, 16 and 32 machines.

Table 3: Number of epochs until amortization.

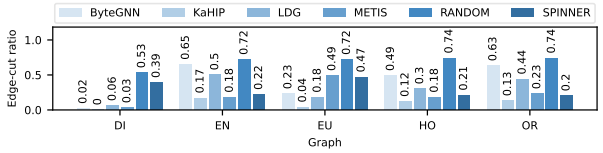
Graph	DBH	2PS-L	HDRF	HEP10	HEP100
EN	1.39	4.57	4.64	3.35	4.29
EU	3.79	never	8.8	10.15	12.0
HO	3.05	4.22	7.26	4.48	4.7
OR	3.83	7.39	11.69	6.64	7.03

LDG [44], and (3) in-memory partitioning with ByteGNN [52], Spinner [29], Metis [21], and KaHIP [41].

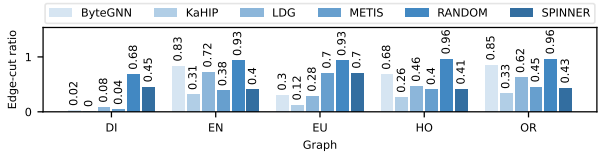
Workloads. We selected a representative set of graph neural network architectures commonly used in distributed GNN training, namely, GAT, GraphSage, and GCN. We use the same hyperparameters as for *DistGNN* (see Table 2). *DistDGL* uses mini-batch training with neighborhood sampling. If not mentioned otherwise, we perform neighborhood sampling for all GNN models with the following configuration. Let S_i be the number of neighbors to sample for layer i. For two layer GNNs, we use $l_1 = 25$ and $l_2 = 20$, for three layer GNNs $l_1 = 15$, $l_2 = 10$ and $l_3 = 5$ and for four layer GNNs, $l_1 = 10$, $l_2 = 10$, $l_3 = 5$ and $l_4 = 5$. We use a global batch size *GBS* of 1024 if not stated otherwise. Therefore, each worker $w_i \in W$ trains with $\frac{GBS}{|W|}$ samples.

Partitioning metrics. We compare the partitioners with the commonly used partitioning quality metrics *edge-cut* and *vertex balance* introduced in Section 2.1. In addition, we measure the training vertex balance. Further, we measure metrics based on the sampled mini-batches: the number of edges of the computation graphs, the number of local input vertices, and the number of vertices that need to be fetched via the network.

Training & Partitioning Time. We measure the epoch and step time and all phases (mini-batch sampling, feature loading, forward pass, backward pass, and model update) for each step. In addition, we measure the partitioning time.



(a) 4 partitions.



(b) 32 partitions.

Figure 13: Edge-cut ratio for all combinations of graph, number of partitions and graph partitioner.

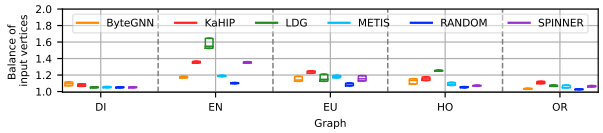
5.2 Partitioning Performance

In the following, we compare the graph partitioners regarding communication costs and computational balance in GNN training.

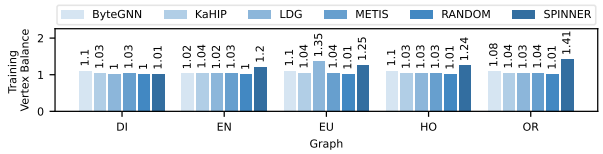
(1) Communication Costs. In Figure 13, we report the edge-cut ratio obtained for each combination of graph, graph partitioning algorithm, and number of partitions. In most cases, *KaHIP* achieves the lowest edge-cut, and random partitioning leads to the largest edge-cut. We observe significant differences between the partitioning algorithms in terms of edge-cut, e.g., *KaHIP* achieves an edge-cut ratio smaller than 0.001 and 0.12 on the graph *DI* and *EU* for 32 partitions, respectively, which is much lower (better) compared to random partitioning which leads to edge-cuts of 0.68 and 0.93 for the same graphs. We also observed for all partitioning algorithms that a higher number of partitions leads to a larger edge-cut.

In the following, we investigate the influence of the edge-cut on network communication. There are cases where a lower edge-cut results in less network communication. However, there are also cases where even if the edge-cut of different partitioners is similar, the network communication can differ a lot. For example, we observed that Spinner has an edge-cut lower than Metis on *OR*, but the network communication is higher. This observation can be explained as follows. There can be edges that are more frequently involved in the sampling process. If these edges are cut, they lead to more network traffic than edges hardly visited in the sampling process. To ensure that the observed anomaly is related to graph partitioning, we measure for each mini-batch the number of vertices needed for processing the mini-batch that are not local to the respective worker. We define these vertices as *remote vertices*. We observe a strong correlation between the number of remote vertices and network traffic. We conclude that edge-cut is not always a perfect predictor for network traffic and that there are cases where a lower edge-cut still leads to more remote vertices and higher network traffic.

(2) Computation Balance. For efficient distributed graph processing, it is crucial that the computation is balanced among machines which is commonly associated with the vertex balance. However, unlike distributed graph processing algorithms such as PageRank, the computational load of mini-batch GNN training depends on the size of the sampled mini-batches. Each worker samples a mini-batch based on the k-hop neighborhood of the training vertices. To ensure load balance, it is essential that the

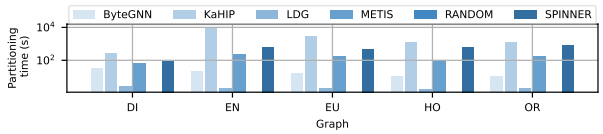


(a) Input vertex balance.

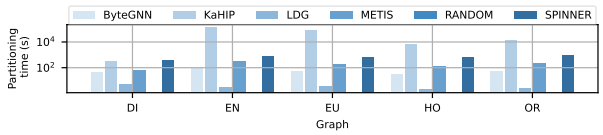


(b) Training vertex balance.

Figure 14: Different balancing metrics for 8 partitions.

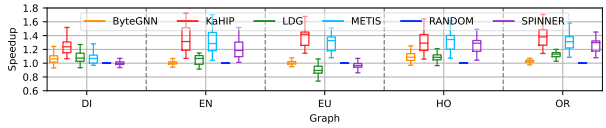


(a) 4 partitions.

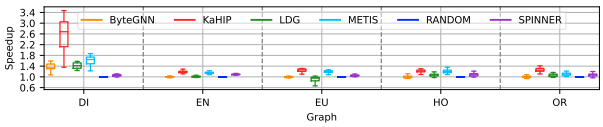


(b) 32 partitions.

Figure 15: Partitioning time on a logarithmic scale.

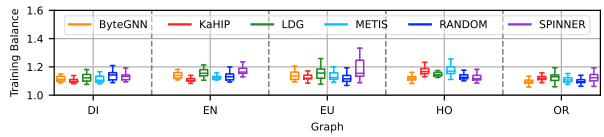


(a) 4 machines.

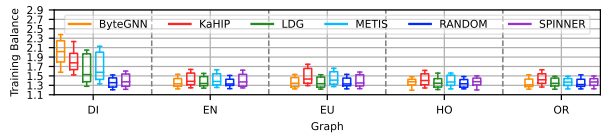


(b) 32 machines.

Figure 16: Speedup distribution of graph partitioners on 4 and 32 machines for all GraphSage experiments.



(a) 4 machines.



(b) 32 machines.

Figure 17: Balance in terms of training time.

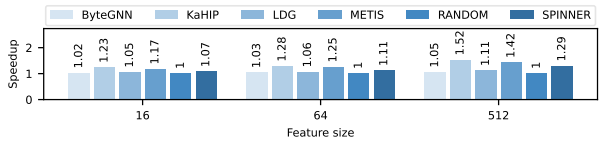
computation graphs of mini-batches are of similar size. We define the number of vertices that are needed to compute a mini-batch as the *input vertices* and *input vertex balance* per step as the number of input vertices of the largest mini-batch divided by the average number of input vertices per mini-batch in the respective step. In Figure 14a, we report the imbalance of the mini-batches in terms of input vertices. We observe a large imbalance, which is increasing as the number of partitions is increased. This imbalance occurs, although the number of training vertices is balanced (see. Figure 14b). This observation indicates that tailoring the partitioning to the sampling and vice versa could be an interesting research direction to ensure computational balancing.

(3) Partitioning Time. In Figure 15, we report the partitioning time for all graphs and partitioning algorithms for 4 and 32 partitions. We observe that the best (in terms of lowest edge-cut) performing partitioner *KaHIP*, leads to the highest partitioning time.

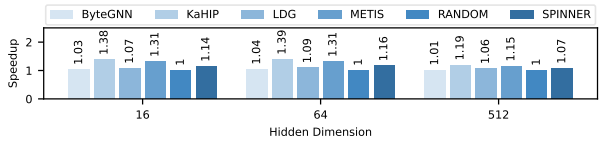
5.3 GNN Training Performance

In Figure 16, we report the average speedups for all combinations of *feature size*, *hidden dimension* and *number of layers* (see Table 2) for all graph partitioners with random partitioning as a baseline on 4 and 32 machines for the GraphSage [15] architecture. In our experiments, *KaHIP* and *Metis* lead to the largest speedups of up to 1.84x, 1.84x, 3.09x and 3.47x on a cluster with 4, 8, 16 and 32 machines, respectively. We see significant variances in terms of speed up that indicate that the effectiveness of the partitioning algorithms depends on the GNN parameters. Therefore, we conduct a detailed analysis of how different GNN parameters influence the different training phases and how the partitioners differ from each other. On each worker in each step, we measure the phase times (1) mini-batch sampling, (2) feature loading, (3) forward pass, (4) backward pass, and (5) model update. In each step, we identify the worker that leads to the longest mini-batch sampling, feature loading and forward pass time as the straggler. We exclude the time for the backward pass because it also contains the time for the all-reduce operation in which the gradients are synchronized between the workers. The model update time is negligible. Then, we get the phase times of the slowest worker per step and sum up the phase times of all steps of the respective slowest worker. In other words, we are interested in how much time the straggler spends on average in each phase. In Figure 17, we report the imbalances in terms of training time for all combinations of GNN parameters per graph and graph partitioner. We observe imbalances (values greater 1) for all partitioners and graphs, showing that even if the number of training vertices is balanced, training time can be imbalanced. As discussed in Section 5.2, this can be attributed to an imbalance in the computation graphs which is caused by the sampling step. In the following, we investigate how the different GNN model parameters influence the effectiveness of graph partitioning in terms of speedup of the distributed training compared to random partitioning.

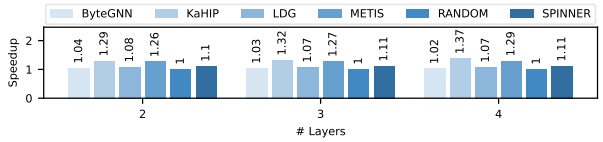
(1) Feature size. We observe that **graph partitioning is more effective in the face of larger feature sizes.** Figure 18a shows the speedups for the partitioners compared to random partitioning dependent on the feature size. For example, the training for GraphSage with *KaHIP* leads to a speedup of 1.23x and 1.52x



(a) Feature size.



(b) Hidden dimension.



(c) Number of Layers.

Figure 18: Speedup of graph partitioners for the GraphSage model on 4 machines for different GNN parameters.

for a feature size of 16 and 512, respectively. This observation is plausible. As feature sizes increase, network communication increases because larger feature vectors are sent over the network, making graph partitioning even more valuable in reducing communication costs.

For each combination of number of layers and hidden dimension, we vary the feature size. We make the following key observations: (1) The larger the feature size, the longer the feature fetching phase (see Figure 19) while the sampling time stays constant. We also observe that for small feature sizes (up to 64), the sampling takes more time than fetching features, but for large feature sizes of 512, the time for feature fetching dominates the sampling time by a lot (see Figure 19). Only, for the road network *DI*, we observe that sampling always takes more time than fetching features which seems plausible because the mean degree in the road network is small and the skew of the degree distribution is low. Therefore, the sampled mini-batches are small, and only a few input vertices must be fetched. We also observe that the edge-cut for the road network is much lower than for the remaining graphs (see Figure 13). (2) The forward and backward pass time increases with larger feature sizes, which is plausible because more computations will be performed in the first layer.

We observe that the partitioners differ a lot from each other when varying the feature size and that the feature size influences the different phases. The feature fetching phase is influenced most, which can for example be seen in Figure 19 for training a three layer GraphSage with a hidden dimension of 64 on the graph *EU*. In most cases, the better the partitioner in terms of edge-cut, the lower the communication costs, which can speed up both the mini-batch sampling and the feature fetching phase.

(2) Number of hidden dimensions. We found that **partitioning becomes less crucial as the hidden dimension increases**. In Figure 18b we report the speedups for the partitioners compared to random partitioning dependent on the number of hidden dimensions. For example, compared with random partitioning, *KaHIP* leads to a speedup of 1.38x and 1.19x, and *Metis* to 1.31x

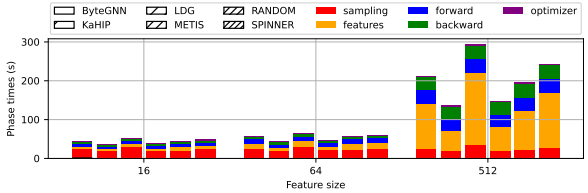


Figure 19: Phase times for a 3 layer GraphSAGE with a hidden dimension of 64 on 4 machines on *EU* graph.

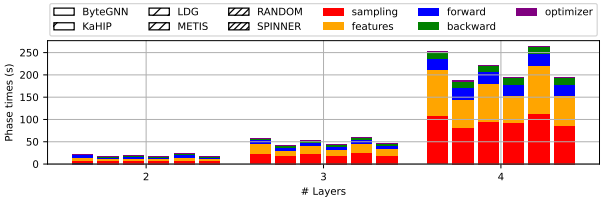


Figure 20: Phase times for a GraphSAGE with a hidden dimension and feature size of 64 on 4 machines on *OR* graph.

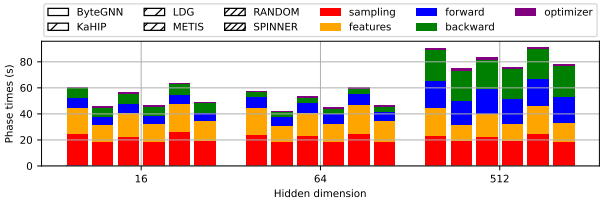


Figure 21: Phase times for a 3 layer GraphSAGE with a feature size of 64 on 4 machines on *OR* graph.

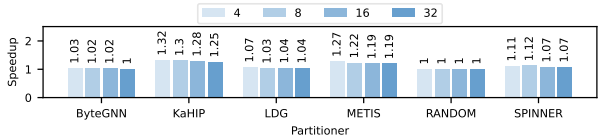
and 1.15x for a hidden dimension of 16 and 512, respectively. This result is reasonable since an increased hidden dimension leads to greater computational costs, potentially dominating the communication costs. We vary the hidden dimension for each combination of feature size and number of layers. Our main observations are: (1) sampling and feature loading time stay constant, which is expected as only the neural network operations are influenced by the hidden dimension. The larger the hidden dimension, the more time is used for computation. (2) We also observe that the effectiveness of partitioners decreases for the larger hidden dimension because most of the differences are in feature loading and sampling. However, if the hidden dimension increases, the computation takes a larger share of the overall training time. Therefore, the difference between the partitioners is lower.

(3) Number of Layers. We observe that **the effectiveness of graph partitioning remains relatively unaffected by an increasing number of layers**. Figure 18c plots the speedups for the partitioners compared to random partitioning dependent on the number of layers. In some cases, the effectiveness slightly increases or decreases, but the influence is much smaller than the influence of the feature size and hidden dimension, and there is also no clear trend. This is an unexpected observation. One could think that the effectiveness of the partitioning algorithms would heavily decrease if the number of layers increases because large parts of the graph will be contained in the mini-batches,

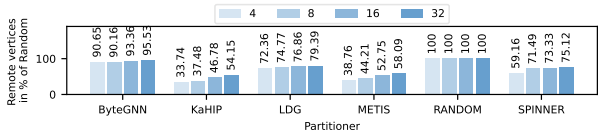
but still, the partitioning algorithms lead to different training times, and many partitioners outperform random partitioning. When varying the number of layers for each combination of feature size and hidden dimension, we make the following key observations: (1) All phases increase in run-time if the number of layers increases. This is expected because an increase in the number of layers leads to larger computation graphs within the mini-batches, which increase the communication costs (more remote accesses in the sampling phase and more remote vertices to fetch via the network) and the computation costs (more neural network operations). (2) We observe, especially for 3 and 4 layer GraphSage, that most of the speedup gained by different partitioning algorithms comes from faster sampling and feature fetching (see Figure 20).

(4) Scale-out factor. In the following, we investigate the effectiveness of graph partitioning in the face of larger scale-out factors. We scale out from 4 to 8, 16 and 32 machines. We make the following observations: (1) For *DI*, scaling out increases the effectiveness of the partitioners. Especially for the partitioners *KaHIP*, *Metis*, *LDG* and *ByteGNN*, the effectiveness increases a lot. However, for *Spinner*, the effectiveness stays relatively constant. This seems plausible as the edge-cut for *Random* and *Spinner* is far higher on *DI* than for the remaining partitioners (see Figure 13). (2) For the remaining graphs, we observe that the effectiveness on GraphSage decreases on average (see Figure 22a). For example, *KaHIP* and *Metis* lead to a speedup of 1.32x and 1.27x on 4 machines and to a smaller speedup of 1.25x and 1.19x on 32 machines, respectively. We found that the number of remote vertices (see Figure 22b) and the edge-cut (see Figure 22c) of the partitioners in percentages of *Random* is increasing when scaling out to more machines. We also observe that the network communication of the partitioners in percentages of *Random* is also increasing. In other words, the effectiveness of the partitioners is also decreasing in terms of partitioning metrics and network communication compared to *Random*, when the number of machines increases. It is worth to note that the feature loading phase scales well. We found that graph partitioning is more effective in the face of large feature vectors. For large feature vectors and few machines, the feature fetching phase can take a large share of the training time and also leads to large differences between the partitioners (see Figure 23). The feature fetching phase can decrease sharply when scaling out to more machines. Therefore, the difference between the partitioners is decreasing, resulting in lower effectiveness. We make a similar observation with the sampling phase. Further, we observe that the effectiveness in terms of memory is decreasing (see Figure 22e) with more machines. This is plausible. For each mini-batch, feature vectors of remote vertices need to be fetched and stored. Therefore, each remote vertex leads to some additional memory. As mentioned above, the number of remote vertices in percent of random partitioning increases with more machines (see Figure 22b) meaning less memory can be saved. We conclude that **in most cases graph partitioning is slightly less effective in the face of larger scale-out factors.**

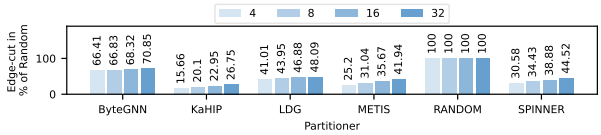
(5) Partitioning time amortization. In Table 4, we report the average number of epochs until the partitioning time is amortized by faster training time for each combination of graph and partitioner. We observe that **the partitioning time can be amortized by faster GNN training**. However, *KaHIP*, the partitioner which leads to the largest speedups only amortizes for *DI*, but barely for the remaining graphs. *Metis*, which also leads to significant speedups, does amortize for all graphs within less than 20 epochs.



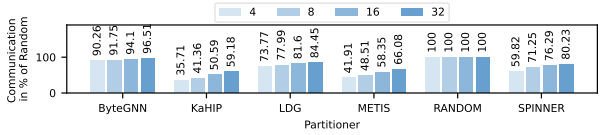
(a) Speedup.



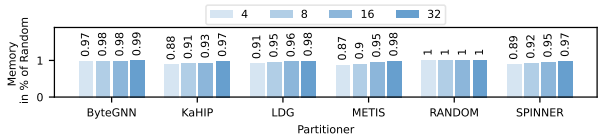
(b) Remote vertices.



(c) Edge-cut.



(d) Communication.



(e) Memory.

Figure 22: The effectiveness of partitioning decreases when scaling GraphSage from 4 to 32 machines.

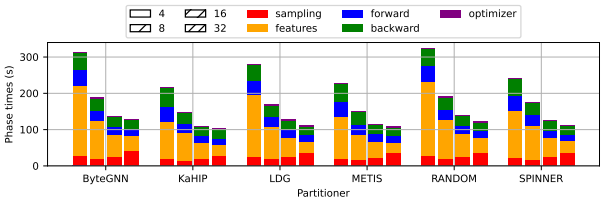


Figure 23: Phase times for a 3 layer GraphSage with a feature size of 512 and a hidden dimension of 64 on *OR* graph trained with 4, 8, 16 and 32 machines.

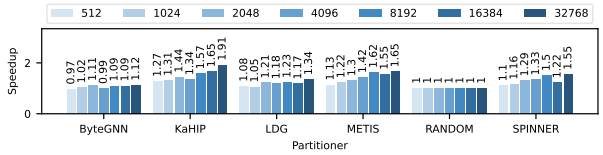
5.4 Influence of mini-batch size

The following experiments aim to investigate the influence of the *mini-batch size* on the effectiveness of partitioning. In other words, we want to evaluate if the partitioning is more crucial (in terms of reduced training time) if the mini-batch size increases.

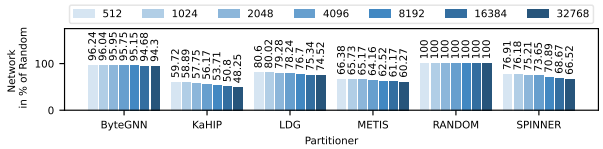
We fix the number of workers to 16 and set the mini-batch size to 512, 1024, 2048, 4096, 8192, 16384, and 32768 for a three layer GAT and a three layer GraphSage. For both GNN architectures, we use two configurations: (1) hidden dimension and feature size of 64 (low communication) and (2) hidden dimension of 64 and feature size of 512 (high communication).

Table 4: Number of epochs until amortization.

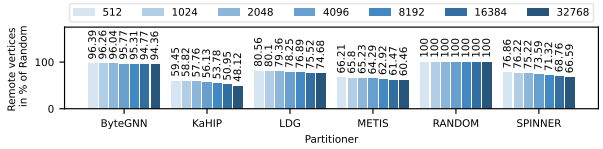
Graph	ByteGNN	KaHIP	LDG	SPINNER	METIS
DI	0.93	2.61	0.1	14.37	1.13
EN	2.16	2501.93	0.39	54.07	16.79
EU	never	1197.25	never	53.8	8.14
HO	0.68	347.51	0.47	77.78	10.7
OR	3.14	223.19	0.27	70.19	14.59



(a) Speedup compared to *Random*.



(b) Network communication in % of *Random*.



(c) Number of remote vertices in % of *Random*.

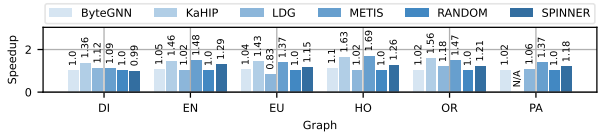
Figure 24: Different metrics for a 3 layer GraphSage with a hidden dimension of 64 and feature size of 512 on the *OR* graph for varying batch sizes between 512 and 32768.

We observe for all partitioners that the network traffic decreases compared to *Random* when the batch size increases (see Figure 24b). For example, *KaHIP* and *Spinner* lead to a network communication of 66% and 77% of *Random* with a batch size of 512 and of 48% and 67% with a batch size of 32768, respectively. We observe a similar trend for the number of remote vertices (see Figure 24c). This seems reasonable. Many vertices can end up in *different* mini-batches. However, if the mini-batch increases, the overlap in the larger mini-batches increases, leading to fewer remote vertices.

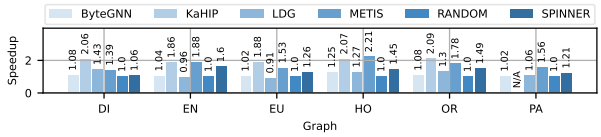
The effectiveness of the partitioners can decrease or increase with larger batch sizes if the feature size is low (64): However, there is no clear trend. In contrast, if the feature size is high (512), in most cases, the effectiveness of the partitioners increases. For example, Figure 24a shows that training with *KaHIP* and *Metis* leads to a speedup of 1.27x and 1.13x for a small batch size of 512 and to a larger speed up of 1.91x and 1.65x if the batch size is set to 32768. We conclude that **graph partitioning is more effective in the face of larger batch and feature sizes.**

5.5 Distributed GPU Training

We conducted distributed training on 8 NVIDIA Jetson AGX Orin GPUs, each equipped with 64 GB GPU memory, resulting in a total of 512 GB of GPU memory. We trained the GraphSage model



(a) Low communication.



(b) High communication.

Figure 25: Speedups for distributed GPU training.

with *DistDGL* on all graphs of Table 1. In addition, we integrated the *ogbn-papers100M (PA)* citation graph from the Open Graph Benchmark (OGB) [20] which has 111 million vertices and 1.6 billion edges. All partitioners were applied to all graphs, except for *KaHIP* which could not partition the *PA* graph[1]. The GNN parameters are set to medium values: the *hidden dimension* is set to 64, the *number of layers* to 3, and the *batch size* to 1024 per GPU. We use two different configurations for the feature size representing low and high communication, respectively. Low communication: The *feature size* is set to 64 for all graphs. High communication: The *feature size* is set to 512 for all graphs except *PA* which runs out of GPU memory in the training process for a feature size of 512. Therefore, the *feature size* was set to 256 instead of 512 for *PA*.

Figure 25a and Figure 25b report the achieved speedups for all combinations of graph and graph partitioner under both configurations, respectively. We observe that graph partitioning significantly speeds up GNN training. Specifically, a high quality partitioner such as *Metis* leads to speedups of 1.09x, 1.48x, 1.37x, 1.69x, 1.47x, and 1.37x for the graphs *DI*, *EN*, *EU*, *HO*, *OR*, and *PA*, respectively in the low communication setting (see Figure 25a). For the high communication setting, *Metis* leads to higher speedups of 1.39x, 1.88x, 1.53x, 2.21x, 1.78x, and 1.56x for the same graphs, respectively (see Figure 25b). In some cases, even higher speedups were observed when using *KaHIP*.

We conclude that **graph partitioning is an effective optimization in reducing GNN training time on GPUs.**

6 LESSONS LEARNED

In the following, we summarize our main findings and relate them to the research questions introduced in Section 3.

(1) Graph partitioning is effective in speeding up GNN training (RQ-1). High-quality graph partitioning significantly enhances GNN training speeds. For *DistGNN*, we observe maximum speedups of 10.41x and average speedups of 4.36x when applying *HEP100*. For *DistDGL*, we observe maximum speedups of 3.47x and average speedups of 1.37x when applying *KaHIP*. The speedups achieved for *DistDGL* are comparable to the speedups seen in distributed graph processing [31, 32, 34], whereas the speedups for *DistGNN* are higher. However, compared to *DistGNN*, *DistDGL* is a more mature and highly optimized system that is still under active development, indicating that there is less room

[1] *KaHIP* returns "The graph is too large."

for improvement with partitioning as for a less optimized system such as *DistGNN*. Still, the results show the importance of partitioning for efficient distributed GNN training.

(2) Graph partitioning is effective in reducing the memory footprint (RQ-1). Graph partitioning effectively reduces memory demands. We found a direct correlation between the replication factor and the memory requirement. Unlike in classical distributed graph processing where vertex states are small, GNNs exhibit large vertex states, often comprising large feature vectors and intermediate representations. Reducing the replication factor notably decreases memory consumption. In many cases, advanced partitioning algorithms leading to smaller replication factors reduced the memory requirements by up to 85.1% and on average by 51.4%. Hence, the replication factor is pivotal for training a GNN within a limited memory budget.

(3) Although classical partitioning metrics are relevant for predicting the performance of GNN training, further aspects should be considered (RQ-2). The replication factor in DistGNN strongly correlates with network communication and memory overhead. In scenarios where different partitioners yield similar replication factors, vertex balancing emerges as crucial. Vertex balance aligns well with memory utilization balance, vital for memory-intensive GNN training, providing a key observation since most edge partitioners prioritize balancing edges and do not focus on vertex balancing. This motivates including vertex balance into the edge partitioning problem definition.

(4) GNN parameters influence the effectiveness of graph partitioning (RQ-3). Unlike in traditional distributed graph processing, GNN training incorporates parameters such as number of layers, number of hidden dimensions, and batch size, alongside attached graph features. For DistDGL, we observed that GNN parameters influence the effectiveness of graph partitioning. Graph partitioning is effective, especially if the feature vectors are large and the hidden dimensions are low. We also found that if the feature size is large and the mini-batch size increases, the effectiveness markedly increases. For DistGNN, while training runtime is less influenced by these parameters, memory overhead efficiency improves with increases in feature size, hidden dimension, or number of layers.

(5) The scale-out factor influences the effectiveness of graph partitioning (RQ-4). For DistDGL, we observed that in most cases, the effectiveness of graph partitioning slightly decreases when scaling out to more machines, while for DistGNN the effectiveness increases. We found that for vertex partitioning (DistDGL) the cut-size of the partitioners increases more sharply than random partitioning. This results in a smaller effectiveness in terms of memory, communication and speedup. This is different from edge partitioning (DistGNN) where we observed an opposite trend: With more machines, the cut-size of the partitioners increases less sharply than for random partitioning and the effectiveness increases.

(6) Graph partitioning can usually be amortized by faster GNN training (RQ-5). Our findings for both DistDGL and DistGNN suggest that the invested graph partitioning time can in many cases be amortized already after a few epochs, making graph partitioning an important optimization for distributed GNN training.

(7) Graph partitioning and sampling are interdependent. Our observations indicate that edge-cut and partitioning balance metrics do not always adequately represent computational balance and communication costs, since both can be influenced by sampling. These findings suggest a potential avenue for research on customizing sampling based on partitioning and vice versa.

7 RELATED WORK

Different studies [2, 12, 37, 48] investigated how graph partitioning influences the performance of distributed graph processing. Verma et al. [48] study graph partitioners available in GraphX [14], PowerGraph [13], and PowerLyra [9] for graph analytics. Abbas et al. [2] study streaming graph partitioners and compare them in a graph processing framework based on Apache Flink [7] for graph analytics. Gill et al. [12] investigate the influence of different partitioning strategies in D-Galois for graph analytics workloads. Pacaci and Özsu [37] study streaming graph partitioners for graph analytics with PowerLyra and graph query workloads with JanusGraph [1]. The studies focus only classical graph workloads. However, distributed GNN training is different. First, GNN training leads to large memory and communication overheads. Huge feature vectors and large intermediate states are computed, stored, and sent over the network. Second, the computations consist of computationally expensive neural network operations. Third, GNN workloads are characterized by GNN parameters such as the number of layers and hidden dimensions. Forth, mini-batch-based training has a complex data loading phase consisting of distributed multi-hop sampling followed by a communication intensive feature loading phase.

Graph partitioning is a vibrant research area and many different approaches exist [16, 19, 21, 28–32, 34, 36, 38–44, 49, 51, 54]. See [8] for a recent survey about graph partitioning. We selected representative state-of-the-art streaming and in-memory graph partitioners for edge and vertex partitioning.

Many distributed graph neural network systems exist [11, 18, 33, 45, 46, 52, 53]. A recent survey [47] gives an overview of different systems. We extend this research by experimentally investigating the effectiveness of graph partitioning for distributed GNN training.

8 CONCLUSIONS

We performed an experimental evaluation to investigate the effectiveness of graph partitioning for distributed GNN training. We showed that graph partitioning is an essential optimization for distributed GNN training and that different factors such as GNN parameters, the scale-out factor, and the feature size can influence the effectiveness of graph partitioning for GNN training, both in terms of memory footprint and training time. Further, we found that invested partitioning time can be amortized by reduced GNN training time. Based on our findings, we conclude that graph partitioning has great potential to make GNN training more effective. We hope our research can spawn the development of even more effective graph partitioning algorithms in the future.

ACKNOWLEDGMENTS

This work is funded in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 438107855 and by the Bavarian Ministry of Economic Affairs, Regional Development and Energy (Grant: DIK0446/01).

REFERENCES

[1] 2023. *JanusGraph*. Accessed: 19 July 2023, https://janusgraph.org/.
[2] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming Graph Partitioning: An Experimental Study. *Proc. VLDB Endow.* 11, 11 (July 2018), 14. https://doi.org/10.14778/3236187.3236208

[3] Maciej Besta and Torsten Hoefler. 2022. Parallel and distributed graph neural networks: An in-depth concurrency analysis. *arXiv preprint arXiv:2205.09702* (2022).

[4] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. 2014. BUbiNG: Massive Crawling for the Masses. In *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 227–228.

[5] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[6] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[8] Ümit Çatalyürek, Karen Devine, Marcelo Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. 2023. More Recent Advances in (Hyper)Graph Partitioning. *ACM Comput. Surv.* 55, 12, Article 253 (mar 2023), 38 pages. https://doi.org/10.1145/3571808

[9] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 1, 15 pages. https://doi.org/10.1145/2741948.2741970

[10] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference (WWW '19)*. Association for Computing Machinery, New York, NY, USA, 417–426. https://doi.org/10.1145/3308558.3313488

[11] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi

[12] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. 2018. A Study of Partitioning Policies for Graph Analytics on Large-Scale Distributed Platforms. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 14. https://doi.org/10.14778/3297753.3297754

[13] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, USA, 14.

[14] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, USA, 15.

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[16] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis Liu, Georgios Theodoropoulos, and Wentong Cai. 2019. Distributed Edge Partitioning for Trillion-Edge Graphs. *Proc. VLDB Endow.* 12, 13 (Sept. 2019), 14. https://doi.org/10.14778/3358701.3358706

[17] Safiollah Heidari, Yogesh Simmhan, Rodrigo N. Calheiros, and Rajkumar Buyya. 2018. Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges. *ACM Comput. Surv.* 51, 3, Article 60 (jun 2018), 53 pages. https://doi.org/10.1145/3199523

[18] Loc Hoang, Xuhao Chen, Hochan Lee, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2021. Efficient distribution for deep learning on large graphs. *update* 1050 (2021), 1.

[19] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2021. CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics. *SIGOPS Oper. Syst. Rev.* 55, 1 (jun 2021), 47–60. https://doi.org/10.1145/3469379.3469385

[20] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[21] George Karypis and Vipin Kumar. 1996. Parallel Multilevel K-Way Partitioning Scheme for Irregular Graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (Supercomputing '96)*. IEEE Computer Society, USA. https://doi.org/10.1145/369028.369103

[22] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web (WWW '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1343–1350. https://doi.org/10.1145/2487788.2488173

[23] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 184–195. https://doi.org/10.1145/3387904.3389268

[24] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[25] Shuangli Li, Jingbo Zhou, Tong Xu, Liang Huang, Fan Wang, Haoyi Xiong, Weili Huang, Dejing Dou, and Hui Xiong. 2021. Structure-Aware Interactive Graph Neural Networks for the Prediction of Protein-Ligand Binding Affinity. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD '21)*. Association for Computing Machinery, New York, NY, USA, 975–985. https://doi.org/10.1145/3447548.3467311

[26] Yang Liu, Xiang Ao, Zidi Qin, Jianfeng Chi, Jinghua Feng, Hao Yang, and Qing He. 2021. Pick and Choose: A GNN-Based Imbalanced Learning Approach for Fraud Detection. In *Proceedings of the Web Conference 2021 (WWW '21)*. Association for Computing Machinery, New York, NY, USA, 3168–3177. https://doi.org/10.1145/3442381.3449989

[27] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 12. https://doi.org/10.1145/1807167.1807184

[28] Daniel Margo and Margo Seltzer. 2015. A Scalable Distributed Graph Partitioner. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 12. https://doi.org/10.14778/2824032.2824046

[29] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. 2017. Spinner: Scalable graph partitioning in the cloud. In *2017 IEEE 33rd international conference on data engineering (ICDE)*. Ieee, 1083–1094.

[30] Christian Mayer, Ruben Mayer, Muhammad Adnan Tariq, Heiko Geppert, Larissa Laich, Lukas Rieger, and Kurt Rothermel. 2018. ADWISE: Adaptive Window-Based Streaming Edge Partitioning for High-Speed Graph Processing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 685–695. https://doi.org/10.1109/ICDCS.2018.00072

[31] Ruben Mayer and Hans-Arno Jacobsen. 2021. Hybrid Edge Partitioner: Partitioning Large Power-Law Graphs under Memory Constraints. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 14. https://doi.org/10.1145/3448016.3457300

[32] Ruben Mayer, Kamil Orujzade, and Hans-Arno Jacobsen. 2022. Out-of-Core Edge Partitioning at Linear Run-Time. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2629–2642. https://doi.org/10.1109/ICDE53745.2022.00242

[33] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K. Ahmed, and Sasikanth Avancha. 2021. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 76, 14 pages. https://doi.org/10.1145/3458817.3480856

[34] Nikolai Merkel, Ruben Mayer, Tawkir Ahmed Fakir, and Hans-Arno Jacobsen. 2023. Partitioner Selection with EASE to Optimize Distributed Graph Processing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2400–2414. https://doi.org/10.1109/ICDE55515.2023.00185

[35] M. E. J. Newman. 2003. The Structure and Function of Complex Networks. *SIAM Rev.* 45, 2 (2003), 167–256. https://doi.org/10.1137/S003614450342480 arXiv:https://doi.org/10.1137/S003614450342480

[36] Joel Nishimura and Johan Ugander. 2013. Restreaming Graph Partitioning: Simple Versatile Algorithms for Advanced Balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '13)*. Association for Computing Machinery, New York, NY, USA, 9. https://doi.org/10.1145/2487575.2487696

[37] Anil Pacaci and M. Tamer Özsu. 2019. Experimental Analysis of Streaming Algorithms for Graph Partitioning. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 18. https://doi.org/10.1145/3299869.3300076

[38] Md Anwarul Kaium Patwary, Saurabh Garg, and Byeong Kang. 2019. Window-Based Streaming Graph Partitioning Algorithm. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW 2019)*. Association for Computing Machinery, New York, NY, USA, Article 51, 10 pages. https://doi.org/10.1145/3290688.3290711

[39] François Pellegrini and Jean Roman. 1996. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, Heather Liddell, Adrian Colbrook, Bob Hertzberger, and Peter Sloot (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 493–498.

[40] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. HDRF: Stream-Based Partitioning for Power-Law Graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM '15)*. Association for Computing Machinery, New York, NY, USA, 10. https://doi.org/10.1145/2806416.2806424

[41] Peter Sanders and Christian Schulz. 2013. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13) (LNCS)*, Vol. 7933. Springer, 164–175.

[42] Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Darren Strash. 2019. *Scalable Edge Partitioning*. 211–225. https://doi.org/10.1137/1.9781611975499.

17
[43] George M. Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. 2017. Partitioning Trillion-Edge Graphs in Minutes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 646–655. https://doi.org/10.1109/IPDPS.2017.95
[44] Isabelle Stanton and Gabriel Kliot. 2012. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)*. Association for Computing Machinery, New York, NY, USA, 1222–1230. https://doi.org/10.1145/2339530.2339722
[45] Qidong Su, Minjie Wang, Da Zheng, and Zheng Zhang. 2021. Adaptive load balancing for parallel gnn training. (2021).
[46] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 495–514. https://www.usenix.org/conference/osdi21/presentation/thorpe
[47] Jana Vatter, Ruben Mayer, and Hans-Arno Jacobsen. 2023. The Evolution of Distributed Systems for Graph Neural Networks and Their Origin in Graph Processing and Deep Learning: A Survey. *ACM Comput. Surv.* (may 2023). https://doi.org/10.1145/3597428 Just Accepted.
[48] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. 2017. An Experimental Comparison of Partitioning Strategies in Distributed Graph Processing. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 12. https://doi.org/10.14778/3055540.3055543
[49] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed Power-law Graph Computing: Theoretical and Empirical Analysis.. In *Nips*, Vol. 27. 1673–1681.
[50] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '18)*. Association for Computing Machinery, New York, NY, USA, 974–983. https://doi.org/10.1145/3219819.3219890
[51] Chenzi Zhang, Fan Wei, Qin Liu, Zhihao Gavin Tang, and Zhenguo Li. 2017. Graph Edge Partitioning via Neighborhood Heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. Association for Computing Machinery, New York, NY, USA, 10. https://doi.org/10.1145/3097983.3098033
[52] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1228–1242.
[53] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.
[54] Michał Zwolak, Zainab Abbas, Sonia Horchidan, Paris Carbone, and Vasiliki Kalavri. 2022. GCNSplit: Bounding the State of Streaming Graph Partitioning. In *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '22)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. https://doi.org/10.1145/3533702.3534920