DGLight: Efficient DGL with Light Communication

Noah Cohen Kalafut, Winfred Li, Hao-Yu Shih

December 20th, 2021

1 Introduction

Graph classification is a crucial problem with practical applications in many different domains [12]. The data from bioinformatics [4], chemoinformatics [7], social network analysis [1], urban computing [2], and cyber-security [6] can all be naturally represented as graphs. In urban computing, constructing effective bike lanes has become a crucial task for governments promoting the cycling lifestyle. Given a directed graph of roadways, the task is to predict and build well-planned bike paths that reduce traffic congestion and decrease safety risks for cyclists and motor vehicle drivers. With the importance of graph classification in mind, we are going to improve the efficiency of graph classification models' training. In this paper, we present *DGLight*, a new model that utilizes partial synchronizations to cut down communication costs during training.

2 Background

Graph classification is a subset of the broader field of graph analysis focused on producing a *classification* based on a given input graph. Graph analysis concerns all matter of producing learned output from a given input graph; this can include generating node embeddings, as is common for shopping/recommender applications, PageRank, and optimal navigation tasks, such as those used in map applications, among others. A unique aspect of graph classification when compared to other graph analysis learning techniques is the focus on the graph rather than on the nodes within. In particular, nodes or edges are not being classified (at least externally) like in recommender systems. Rather, the graph itself is being evaluated. An example can be seen in Figure 1.

In the real world, graphs can have a variety of attributes. They can be directed or undirected, weighted or unweighted. Graphs also fluctuate in size making computations involving them rather difficult. Additionally, operations involving graphs are conceptually strange to parallelize. In the case of *DGL* [16], the graph analysis backend used within *DGLight*, parallelization and batching are performed by combining all n graphs being classified into one graph of n connected-components before performing graph convolution/generating node features. This can be seen in Figure 2.

We focus primarily on implementing a maximally-efficient distributed computing solution for graph classification models. This solution will ideally have compatibility with the major graph classification models and training techniques discussed in Section 5. The solution should improve by-epoch training time on methods such as those in [5, 8] with minimal loss in accuracy. These results will be achieved through optimization on top of DGL. Specifically, we will experiment with changes to gradient synchronization frequency and aggregation techniques to find the most efficient solution for various graph classification problems.

3 Design

3.1 Motivation

A primary bottleneck in distributed training is communication. There are various communication overheads during training. Primarily, these overheads revolve around weight synchronizations. Weight synchronization is the process by which models on each node in the cluster communicate to ensure proper convergence.

In the most strict distributed models, weight synchronization will occur during each backward step. This will ensure that the convergence properties and accuracy of the model are identical to that of the non-distributed version. In tightly-connected or smaller networks, this overhead is negligible. However, as models get larger, weight synchronizations become more frequent, or cluster size increases, the distributed overhead becomes the primary barrier to scalability of distributed model training. To illustrate the diminishing returns caused by network overhead, we show *GraphSAGE*'s average runtime by number of nodes over 100 epochs in Figure 3.

For comparison, we also run a typical graph-convolutional network in a batch-distributed environment and compare to the training time of individual models with equivalent batch sizes. This is shown in Figure 4. Notice that the individual models scale more linearly and

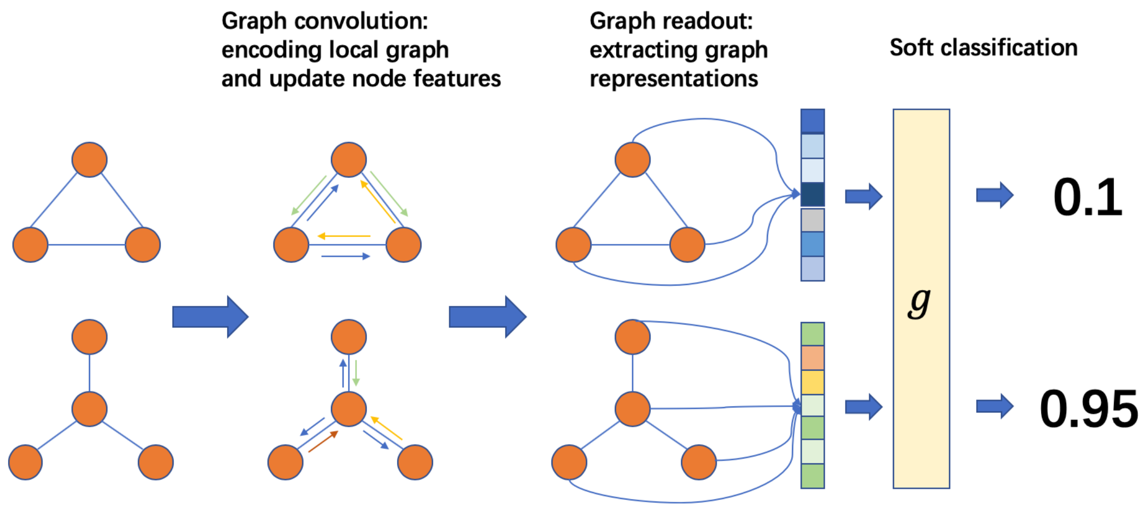


Figure 1: Simple example of a graph convolutional network from [16]. In this case, node features are generated and extracted to produce a vector representative of the input graph. A simple feed-forward network is then used to produce a classification based on the graph embedding.

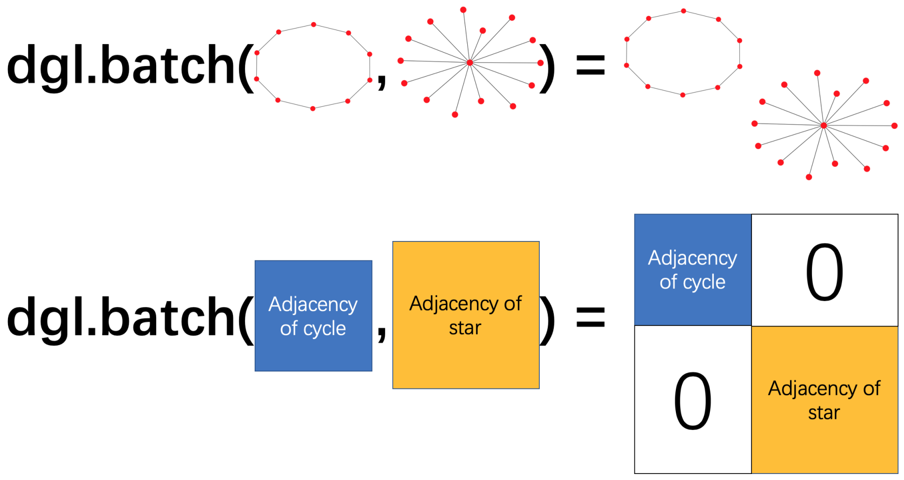


Figure 2: Batching technique in [16]. Essentially, the adjacency matrices of each graph are concatenated into one block diagonal matrix, which allows for parallelized graph convolution.

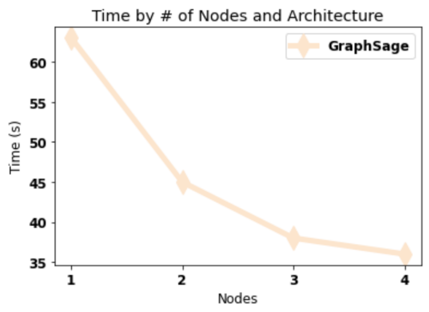


Figure 3: Average runtime of [10] on over 100 epochs.

do not increase their proportional backpropagation time when compared to a synchronized model. This is a direct result of the elimination of communication overhead.

3.2 Solution

To combat these issues, we can reduce the number/size of gradient computations taking place. Some DGL applications utilize *gather-scatter* all-reduce for synchronization and perform weight synchronization on every backward step. Gather-scatter all-reduce aggregates model weights on a single node before distributing the new weights (usually the average) back to each node This incurs a near-maximal amount of overhead for the purposes of maximal compatibility. Additionally, stragglers and long-running batches more frequently take the longest time possible, as all weight updates from each node need to be received before they can be aggregated and distributed. We can improve on this.

Firstly, we can skip synchronizations. This is not a new idea, and has been shown to be beneficial in previous implementations. Frequent skipping of synchronizations can lead to calculation on stale weights, causing weight updates to have less statistical efficacy. Further, models on different nodes can diverge. If this occurs, it is unlikely that aggregation of the multiple models present in the system will produce a favorable result. As such, skipping weight synchronization needs to be done carefully and tuned according to the dataset and model.

We also utilize a modified version of *ring-reduce*. Ring-reduce is a popular reduction technique in distributed computing. Within ring-reduce, weights are passed from node to node sequentially. Eventually, all weights are aggregated (once the final node in the sequence is reached) and the process is repeated. This technique staggers network communication and is particularly useful for applications with network constraints.

Both of these techniques are useful for optimizing algorithm training time. However, these systems have yet to be combined. We suggest a hybrid approach. Each ring-reduce step can occur on a separate backward run. Pairs of nodes are chosen sequentially and one node inherits the average model weights from combination with the other. This will minimize network congestion by further staggering synchronizations when compared to ring-reduce. Additionally, our hope is that the *staggered staleness* of the weights on each node will aid with convergence when compared to traditional synchronization skipping. A visual explanation of the hybrid approach is seen in Figure 5.

Although not implemented in this paper, in the future, this hybrid approach could be adapted to have multiple send/receive pairs in a single step. Visually, in Figure 5, as nodes 1-2 exchange weights in step 1, nodes 3-4 could as well, continuing with 2-3 and 4-1 in step 2, and so-on.

Finally, we also explore alternative training methods with graph partitioning. By default, DGL's distributed module communicates between partitions in every update, and similar to the idea of skipping synchronizations, we wanted to skip updates that involve edges between partitions. However, we found implementing this behavior on DGL to be rather challenging and so we ended up considering the extreme case of completely independent training between partitions. We leave the synchronization case for future work.

4 Evaluation

For all runs in this paper, Cloudlab [14] was used for the distributed environment. Each of the 4 nodes had 5 cores and 16 Gbs of RAM each with Ubuntu 18 installed on a virtual machine. Minimal packages (`DGL, Numpy, Torch`) were installed on the cluster before runtime. Separate runs were conducted throughout the paper. However, all plots in the same figure were run on the same nodes in as short a time-frame possible. This was to ensure maximal result consistency.

4.1 Skipping Synchronizations

Firstly, we test purely skipping weight synchronizations. This will certainly speed up iterations. However, what needs to be assessed is how much each iteration speeds up and how the classifier accuracy is affected. The loss for various synchronization frequencies on [13] using a typical GCN can be seen in Table 4.1. The runtime can be seen in Figure 6.

Sync Frequency	**Always**	**2**	**10**	**Never**
Loss	.721	.847	.846	.846

Table 1: Minimal loss for various distributed synchronization skipping frequencies on *OHSU* [13] over 100 epochs using 4 nodes.

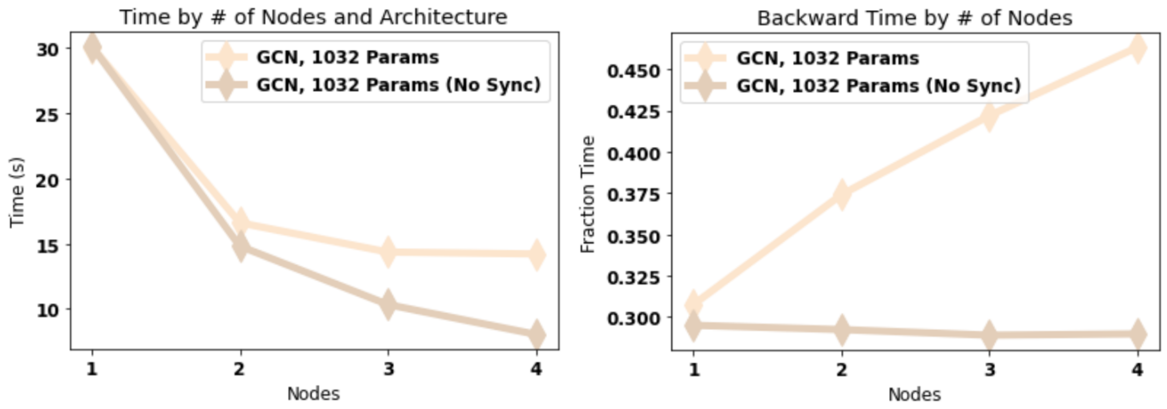
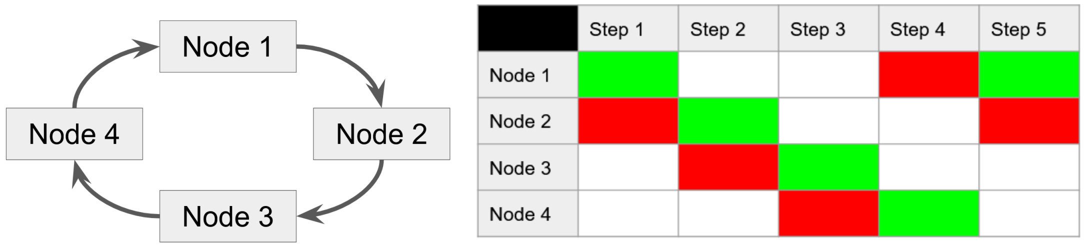


Figure 4: Comparison between GCNs of constant size with and without synchronization on a distributed setup.



	Step 1	Step 2	Step 3	Step 4	Step 5
Node 1	🟩			🟥	🟩
Node 2	🟥	🟩			🟥
Node 3		🟥	🟩		
Node 4			🟥	🟩	

Figure 5: Hybrid ring-reduce setup and runtime synchronization timings. Green and red correspond to sending and receiving weights, respectively.

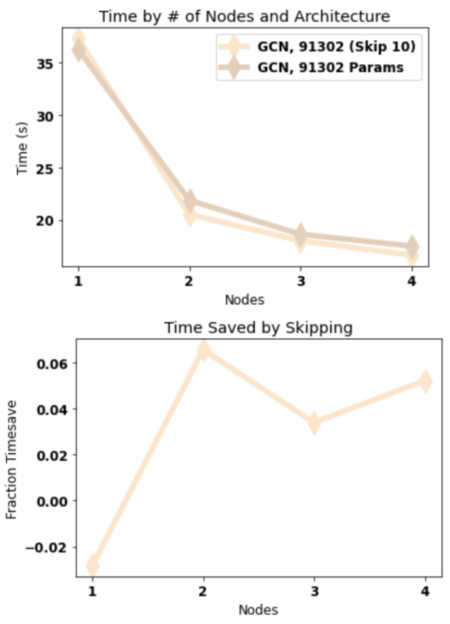


Figure 6: (Top) Total runtimes over 100 epochs using a GCN with $91,302$ parameters on [13] for different synchronization skip strategies. (Bottom) Time saved through the use of synchronizing $1/10$ epochs.

Skipping synchronizations was rather inconsistent in terms of convergence. This could be a potential result of the model architecture or the input graphs being volatile such that different batches have wildly different characteristics. It makes sense that this problem would then be exacerbated for batch-split distributed systems. It would be interesting to see work in the future that explores this type of reliability for weight synchronization skipping as a function of number of nodes.

The time saved through skipping can be seen in Figure 6. There was only around a 5 percent timesave from syncing once every 10 epochs, which does not speak well for the efficiency of skipping synchronizations as a solitary method for reducing iteration time when considered with the effect on model loss.

4.2 Hybrid Approach

We want to compare our syncing approach with gather-scatter and ring-reduce on a network of sufficient bandwidth. This is done in Figure 7. We find that our hybrid approach is roughly equivalent in timing to gather-scatter with a synchronization once every 3 epochs, but with closer-to-optimal accuracy.

A runtime spike with two nodes occurs in both ring-reduce and hybrid implementations. When only two

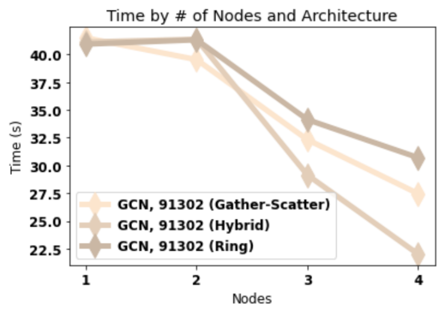


Figure 7: Comparison of gather-scatter, ring-reduce, and our hybrid approach on *Peking_1* data [13] over 100 epochs under uncongested network conditions. (Left) Accuracy was the same for gather-scatter and ring-reduce but fell from .688 to .647 when using the hybrid approach. (Right) Runtime segmented by number of nodes and time spent in each training phase

nodes are involved, both ring-reduce and our hybrid approach essentially act as a more complicated gather-scatter, which explains the performance difference. We also note that, in our environment with sufficient network bandwidth, gather-scatter appears to out-perform ring-reduce. In the future, we would like to more thoroughly test the approach on networks with insufficient bandwidth.

During a period within which CloudLab's network appeared to slow down, some runtime tests were performed. We see that, strangely, the hybrid approach appears to perform better under network congestion in Figure 8. Similarly, when looking at the runtime for each method segmented by training phase in Figure 9, we note that our hybrid approach is, on average, faster in the forward step as well as the backward step when compared to both alternative synchronization methods. The forward step's time reduction was likely the result of natural error/randomness, as no synchronization takes place during this step. Depending on the architecture, this could also be a result of CloudLab's virtualization and load balancing. However, we maintain that the relationship between existing methods and our hybrid approach in the results shown are nonetheless accurate, and the time reduction's usefulness is supplemented by minimally decreased classification accuracy.

4.3 Graph Partitioning

We compare the results of training the *GraphSAGE* model for 100 epochs on the CiteSeer [15] dataset using DGL's distributed module and training partitions independently with weight synchronization every 5 epochs. The results are shown in Figures 10 and 11. We can see that training partitions independently clearly reduces training time, however yields worse validation accuracy due to losing information from edges between partitions. We also note that the runtime does not seem to scale proportionally over the number of partitions, but we believe this is due to communication bottlenecks and we would expect to see better scaling with a more computationally intensive workload.

5 Related Work

5.1 Neural Network Architecture

In [17], the authors designed novel neural network structures that can accept graphs to train graph classification models. They proposed an end-to-end deep learning architecture for graph classification and a spatial graph convolution layer to extract vertex features. The key innovation is a new SortPooling layer which takes as input a graph's unordered vertex feature from spatial graph convolution. These three aspects make their Graph Convolution Neural Network highly competitive when compared to state-of-the-art graph kernels [3].

5.2 Training Methodology

There are many papers which focus on optimal convergence of graph classification models. Included among these are [8] and [5]. Between the two of these papers, several training techniques and architectures are explored for use in graph classification problems. In order to maximize compatibility, it would be ideal for our final efficient training solution to be able to overlap with these existing methods. Additionally, in [8], several best-practices are laid out for use in reporting graph classification model results. Adhering to these best-practices and encouraging replication studies will be an objective in this paper during reporting.

5.3 Existing Frameworks

Distributed computing and parallelization systems centered around graph classification models are somewhat rare. However, several papers discuss efficient graph computation. Among these papers is [9]. [9] creates several standalone applications for use in graph computation on distributed systems and is flexible enough to handle both synchronous and asynchronous workflows. However, these approaches are not specifically tailored to graph classification, and could be optimized using the above methods or using a similar methodology to *PipeDream* [11].

Additionally, we can look at existing frameworks for graph processing in order to get a sense of how these systems are designed. For instance, the DGL Python pack-

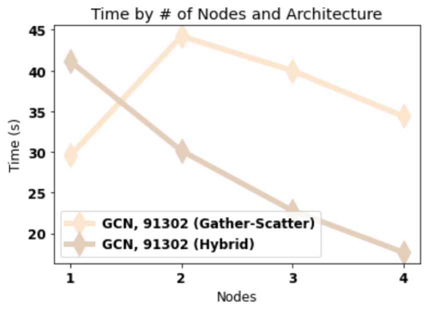


Figure 8: Total runtimes over 100 epochs using a GCN with $91,302$ parameters on [13] under (likely) congested network conditions. Unfortunately, Cloudlab's virtualization makes some processing times sporadic, so it is difficult to form a conclusion here.

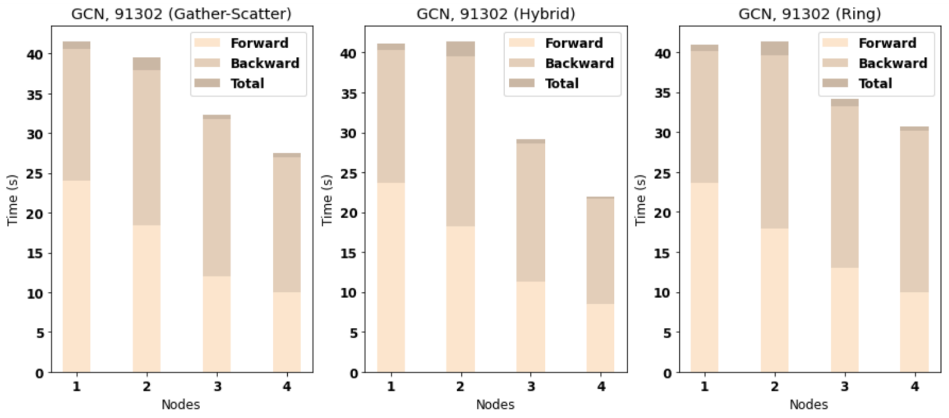


Figure 9: Comparison of gather-scatter, ring-reduce, and our hybrid approach over 100 epochs, segmented by number of nodes and time spent in each training phase.

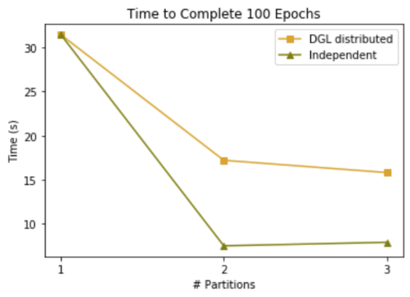


Figure 10: Comparison of runtimes between DGL distributed and independent partition training

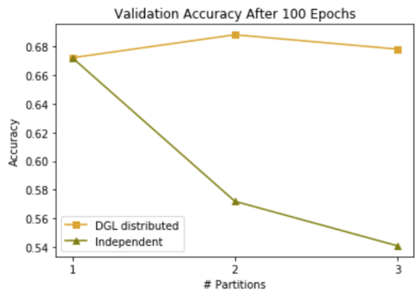


Figure 11: Comparison of validation accuracies between DGL distributed and independent partition training

age [16] provides easy implementations of graph neural networks. It also provides support for distributed training. Their framework involves three processes: server, sampler, and trainer. The server process runs on each machine and stores a graph partition. The sampler interacts with servers to generate minibatches for training, and the trainers interact with the servers and samplers in order to carry out training.

6 Future work

There is room for future research in several areas covered within this paper.

For our hybrid reduction method, multiple sends/receives could be overlaid. This was briefly discussed in Section 3.2. Doing so could allow users to bound staleness of weight values while maintaining low network utilization. We would also like to study the effects of the averaging technique (computing the average of (1,2), then ((1,2),3), and so on) on convergence.

More analysis could also be performed using an artificially limited or congested network. Doing so would reveal the strengths of ring-reduce and our hybrid approach and demonstrate their performances in an environment more suited to their construction. Additionally, it would be useful to collect more substantial network consumption information.

The integration of graph partitioning methods and our hybrid approach was left open in this paper. Their integration would be crucial to optimizing for graph classification problems. Additionally, graph partitioning overhead, given that multiple graphs are utilized in graph classification, would also be a useful metric to observe. Then, load balancing techniques and graph partitioning methodologies could be compared more thoroughly.

We also currently only use four nodes in our models. It would be useful to see how *DGLight*'s scaling holds as more nodes are introduced. The absolute performance cap of the algorithm would also be useful to survey on different hardware/networks.

As mentioned previously, we found it challenging to implement training on graph partitions where the updates would consider edges between partitions only periodically. We showed that training partitions independently decreases runtime but at the cost of accuracy. It would be interesting to further explore intermediate solutions between the two extremes of full synchronization between partitions and none.

7 Contributions

The responsibilities of each group member were distributed in the following manner: Noah was responsible for coding/running the hybrid algorithm, creating visualizations, and writing sections 2-4.2. Winfred ran *GraphSAGE* on a distributed cluster, fully managed the topic of graph partitioning, and configured the CloudLab instance. Hao-Yu ran a *DGL* graph classification algorithm, and wrote on neural network architectures and outlined topics for future work.

All code can be found at `github.com/Oafish1/CS-744-Project`

References

[1] L. Backstrom and J. Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 635–644, 2011.

[2] J. Bao, T. He, S. Ruan, Y. Li, and Y. Zheng. Planning bike lanes based on sharing-bikes' trajectories. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1377–1386, 2017.

[3] K. Borgwardt, E. Ghisu, F. Llinares-López, L. O'Bray, and B. Rieck. Graph kernels: State-

of-the-art and future challenges. *arXiv preprint arXiv:2011.03854*, 2020.

[4] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. Vishwanathan, A. J. Smola, and H.-P. Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(suppl_1):i47–i56, 2005.

[5] M. T. Do, N. Park, and K. Shin. Two-stage training of graph neural networks for graph classification, 2021.

[6] H. Duen, N. Carey, W. Jeffrey, W. Adam, and F. Christos. Polonium: tera-scale graph mining for malware detection. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*, 2011.

[7] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. *arXiv preprint arXiv:1509.09292*, 2015.

[8] F. Errica, M. Podda, D. Bacciu, and A. Micheli. A fair comparison of graph neural networks for graph classification. In *International Conference on Learning Representations*, 2020.

[9] A. Guerrieri. Distributed computing for large-scale graphs. 2015.

[10] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017.

[11] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, and P. B. Gibbons. Pipedream: Fast and efficient pipeline parallel DNN training. *CoRR*, abs/1806.03377, 2018.

[12] J. B. Lee, R. Rossi, and X. Kong. Graph classification using structural attention. pages 1666–1674, 2018.

[13] S. Pan, J. Wu, X. Zhu, G. Long, and C. Zhang. Task sensitive feature exploration and learning for multitask graph classification. *IEEE Transactions on Cybernetics*, 47(3):744–758, 2017.

[14] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), Dec. 2014.

[15] P. Sen, G. Namata, M. Bilgic, L. Getoor, and B. Galligher. Collective classification in network data. *AI Magazine*, 29(3):93, 2008.

[16] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2020.

[17] M. Zhang, Z. Cui, M. Neumann, and Y. Chen. An end-to-end deep learning architecture for graph classification. 2018.