2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-1016-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/HIPC53243.2021.00042

DistMILE: A Distributed Multi-Level Framework for Scalable Graph Embedding

Yuntian He, Saket Gurukar, Pouya Kousha, Hari Subramoni, Dhabaleswar K. Panda, Srinivasan Parthasarathy
Dept. of Computer Science and Engineering
The Ohio State University
{he.1773, gurukar.1, kousha.2}@osu.edu, {subramon, panda, srini}@cse.ohio-state.edu

***Abstract*—Scalable graph embedding on large networks is challenging because of the complexity of graph structures and limited computing resources. Recent research shows that the multi-level framework can enhance the scalability of graph embedding methods with little loss of quality. In general, methods using this framework first coarsen the original graph into a series of smaller graphs then learn the representations of the original graph from them in an efficient manner. However, to the best of our knowledge, most multi-level based methods do not have a parallel implementation. Meanwhile, the emergence of high-performance computing for machine learning provides an opportunity to boost graph embedding by distributed computing.**

In this paper, we propose a Distributed MultI-Level Embedding (DistMILE[1]) framework to further improve the scalability of graph embedding. DistMILE leverages a novel shared-memory parallel algorithm for graph coarsening and a distributed training paradigm for embedding refinement. With the advantage of high-performance computing techniques, DistMILE can smoothly scale different base embedding methods over large networks. Our experiments demonstrate that DistMILE learns representations of similar quality with respect to other baselines, while reduces the time of learning embeddings on large-scale networks to hours. Results show that DistMILE can achieve up to $28\times$ speedup compared with a popular multi-level embedding framework MILE and expedite existing embedding methods with $40\times$ speedup.

***Index Terms*—Graph Embedding, High-Performance Computing, Distributed Machine Learning, Multi-Level Framework**

I. Introduction

Graph embedding aims to learn low-dimensional representations capturing the structural properties in the network. The learned representations can be used as features in a variety of machine learning tasks such as node classification and link prediction. Graph embedding has been deployed in various real-world applications including anomaly detection [1] and e-commerce recommendation [2].

In recent years, many graph embedding methods have been proposed using different methodologies such as matrix factorization (NetMF [3]), random walk (DeepWalk [4] and node2vec [5]), and deep neural network (SDNE [6]). These algorithms usually either consume too much time or memory, which prevents them from scaling to large datasets. To improve the scalability of existing embedding methods, Liang et al. [7] proposed a multi-level embedding framework MILE. MILE first repeatedly coarsens the original graph, then applies an existing embedding method to the coarsened graph, and finally uses graph neural networks (GNN) to refine the embeddings. Experiments demonstrate that MILE can not only boost the embedding methods but also scale them to large networks. Methods using a similar framework include HARP [8] and GOSH [9].

A benefit of MILE is that it is agnostic to the underlying embedding methods, which is convenient for users. While HARP and GOSH are designed for specific embedding methods, MILE is compatible with any existing methods. For example, scientific or commercial projects [2], [10] usually have their own customized embedding system, and the expense of replacing it with a new one can be prohibitive. In contrast, wrapping the current embedding system in MILE can preserve the customized methodology and is relatively inexpensive.

Additionally, machine learning benefits from the usage of high-performance computing techniques. One might think of extending the multi-level embedding framework to parallel or distributed settings. However, it is non-trivial to leverage parallelism in graph algorithms due to the connectivity of nodes in a graph, which can cause race conditions if processed in parallel. For example, MILE and many multi-level graph partitioning algorithms [11]–[13] adopt heavy edge matching (HEM) and its variants for graph coarsening. HEM has different parallel implementations for distributed systems [12] and shared-memory machines [13]. These implementations, unfortunately, only emphasize handling race conditions rather than reducing them. Moreover, some approaches used in multi-level embedding do not have any parallel implementation, such as the structural equivalent matching (SEM) approach in MILE.

Another challenge of designing parallel multi-level embedding methods lies in model training. For example, MILE trains a GNN to learn embeddings on a finer-grained graph from a coarser one. As the growing scale of network data exceeds the limit of GPU memory, multiple GNN models [14]–[16] use minibatch to train on GPU and enable parallel training. On the other hand, distributed machine learning has attracted much interest [17]–[19]. Horovod [17] is a distributed training framework which scales the model training across multiple machines with a high scaling efficiency. It is also compatible with MPI implementations such as MVAPICH [20] for further enhancement.

In this paper, we proposed a Distributed MultI-Level

[1]Our code is available at https://github.com/heyuntian/DistMILE

2640-0316/21/$31.00 ©2021 IEEE
DOI 10.1109/HiPC53243.2021.00042

Embedding framework **DistMILE**, which extends MILE to a distributed setting. Our contributions can be summarized as follows:

- We propose DistMILE, a distributed multi-level embedding framework. To the best of our knowledge, this is the first model-agnostic embedding framework designed for distributed setting. DistMILE leverages a hybrid of parallel and distributed computing techniques to expedite the embedding process. It treats the base embedding phase as a black box, hence it is generalizable to any embedding method.
- We propose a parallel graph coarsening algorithm, which includes a novel locality-sensitive-hashing (LSH) based approach for node matching. We explore different design choices in this algorithm to reduce the synchronization costs and race conditions in parallel coarsening. The proposed algorithm can also be used in other graph problems such as graph partitioning.
- We propose a distributed training paradigm for the refinement of node embeddings. Specifically, this training paradigm is built on Horovod and leverages distributed learning and parallel computing technique for a maximum speedup.
- We conduct extensive experiments on real-world graph datasets to compare DistMILE with MILE. Through the task of node classification, our experiments show that DistMILE can learn embeddings of comparable quality with respect to MILE, but significantly reduce the running time. On YouTube, DistMILE achieves up to $28\times$ speedup. On the largest dataset Yelp, only DistMILE is able to scale the embedding methods to learn the representations.
- We also study the impact of different parameters on DistMILE's performance. The results not only show that DistMILE performs well under different settings, but also give an insight into the tradeoff between the cost of computing resources and the performance of our proposed framework.

The rest of this paper is organized as follows: Section 2 reviews previous work on distributed machine learning as well as graph embedding, especially multi-level embedding methods. Section 3 formulates the graph embedding problem on multiple machines. Section 4 introduces the methodology of DistMILE, including a shared-memory parallel graph coarsening algorithm and a distributed training paradigm. In Section 5, we evaluate DistMILE and MILE to observe their performance in terms of efficiency and quality. We also conduct drilldown experiments to understand how DistMILE works under different settings. We conclude this paper in Section 6.

II. Background and Related Work

In this section, we first introduce multi-level embedding, including a popular framework MILE [7] and other works from this group. We also review the related work of graph embedding and high-performance computing for machine learning.

A. Multi-level Embedding

The multi-level embedding framework is a powerful tool to mitigate the huge overhead of embedding large graphs. In general, a method using this framework first coarsens the input graph then learn the embeddings of the fine-grained graph from the compressed ones. By shrinking the graph size, the multi-level embedding framework can not only boost the embedding but also preserve higher-order structural features for a better quality.

One recent work adopting this framework is MILE [7]. MILE runs on a single machine with no parallel computing involved. It consists of three phases: graph coarsening, base embedding, and embedding refinement. Initially, given the original graph $\mathcal{G}$ (or $\mathcal{G}_0$), MILE repeatedly coarsens the graph m times and eventually obtains the coarsest graph $\mathcal{G}_m$. At each coarsen level, MILE merges groups of nodes into supernodes in the coarser graph, and the edges of a supernode are the union of edges incident to its nodes in the finer graph. MILE adopts a hybrid of two methods for node matching: Structural Equivalence Matching (SEM) and Normalized Heavy Edge Matching (NHEM). SEM aims to find nodes which have the same neighbors, while NHEM matches each unmatched node u with its unmatched neighbor v such that the normalized weight of edge (u, v) is maximized. In addition, NHEM visits the nodes in the ascending order of their number of neighbors in order to collapse more nodes.

After forming the coarse graph, MILE applies a graph embedding method on $\mathcal{G}_m$ to learn the embeddings $\mathcal{E}_m$. Due to the smaller size, embedding $\mathcal{G}_m$ is more efficient than directly embedding $\mathcal{G}$. MILE treats this phase as a black box, and it is generalizable to any embedding method.

The final phase of MILE is embedding refinement. Given the embeddings on the coarsest graph $\mathcal{E}_m$ and the series of graphs $\{\mathcal{G}_m, \mathcal{G}_{m-1}, \ldots, \mathcal{G}_1, \mathcal{G}_0\}$, MILE iteratively computes embeddings of each graph and finally gets $\mathcal{E}_0$. Specifically, MILE trains a GNN model (e.g., GraphSAGE [14] and GCN [21]) that refines the embeddings $\mathcal{E}_i$ to $\mathcal{E}_{i-1}$. At each level, MILE projects the embeddings of supernodes in $\mathcal{G}_i$ to their corresponding nodes in $\mathcal{G}_{i-1}$, then applies the trained model to refine the projected embeddings.

In addition to MILE, some other embedding methods using the multi-level framework have been proposed recently. HARP [8] has a similar paradigm that coarsens the input graph prior to embedding. Instead of directly refining $\mathcal{E}_i$, HARP learns $\mathcal{E}_{i-1}$ by embedding $\mathcal{G}_{i-1}$ with $\mathcal{E}_i$ used as initialization. Unlike MILE and HARP which do not have a parallel implementation, GOSH [9] is a multi-level embedding approach that runs in parallel. GOSH benefits from the high speed of GPU-based training and is able to embed large networks with limited GPU memory. However, HARP and GOSH are not model-agnostic because of their customized embedding modules that cannot extend to other embedding methods. Recently Deng et al. [22] proposed GraphZoom that relies on the multilevel framework to learn node embeddings. GraphZoom accelerated the coarsening process by utilizing the

optimized MATLAB modules written in C and C++.

B. Graph Embedding

Learning embeddings on graphs has been extensively studied in recent years, which aims to learn a vector representation for each node preserving its structural features. Inspired by the success of using the skip-gram model on word embedding, a plethora of methods such as DeepWalk [4] and node2vec [5] have been proposed. These methods generate random walks on graphs and feed them into a skip-gram model to learn node embeddings. Another group of embedding methods are based on matrix factorization, e.g., NetMF [3]. With the increasing popularity of deep learning, recent works have switched to leveraging deep neural networks to learn higher-order graph structures. Methods from this group include SDNE [6] and VAG [23].

Most embedding methods cannot efficiently scale to large-scale networks. In addition to the aforementioned multi-level framework, there are different attempts to improve the scalability of graph embedding. GraphVite [24] is a CPU-GPU hybrid system for faster training. LightNE [25] is a CPU-only parallel embedding algorithm based on matrix factorization. However, these algorithms are not model-agnostic and only designed for a single machine. Facebook released a distributed large-scale embedding system PyTorch-BigGraph [26] which is optimized for efficient training without exceeding the memory limit. However, our previous experiments show that PyTorch-BigGraph is outperformed by *our sequential implementation of MILE* in terms of efficiency and accuracy for the tasks of node classification and link prediction [7]. We therefore do not compare with it.

C. Distributed Machine Learning

Presently, the application of machine learning algorithms often involves a huge volume of data, hence the demand for high performance computing in machine learning has emerged. One common way to enhance the computational power is to add more computing nodes to the HPC system [27]. With an increasing popularity, distributed learning has been supported in various methodologies. Distributed ensemble learning is available in popular ML libraries such as TensorFlow [28] and PyTorch [29], which trains multiple models on distributed machines and aggregates their outcome for prediction. Another straightforward option is parallel synchronous training, which relies on the MPI libraries for communication. Baidu Ring Allreduce [18] and Horovod [17] exploit an efficient ring-like algorithm for communication efficiency. DistBelief [19] and DIANNE [30] are asynchronous alternatives for distributed training.

III. Problem Statement

A graph is denoted as $G = (V, E)$, where V is the set of nodes and E is the set of edges in G. Table I shows the notations used in the paper. We formulate the problem of graph embedding as follows:

TABLE I
NOTATIONS USED IN THIS PAPER

Symbol	Definitions
G	Input graph for embedding
V, E	Node set, edge set
W	Edge weights, $W_{u,v}$ denotes the weight of edge (u, v)
$\mathcal{N}(u)$	Neighbors of node u
$\sigma(u)$	Degree of node u
d	Embedding dimension
m	Coarsen depth
M	Number of machines
T	Number of threads
h	Number of hash functions
n_c	Threshold for parallel coarsening
n_m	Threshold for the size of coarsened graph
p	Large prime number used by $\mathcal{F}$
$\mathcal{F}$	Hash functions for SEM
Q	Node signatures for SEM
ϕ	Matching results, $\phi(u)$ denotes the node matched with u
b	Batch size in GraphSAGE
s	Number of sampled neighbors in GraphSAGE

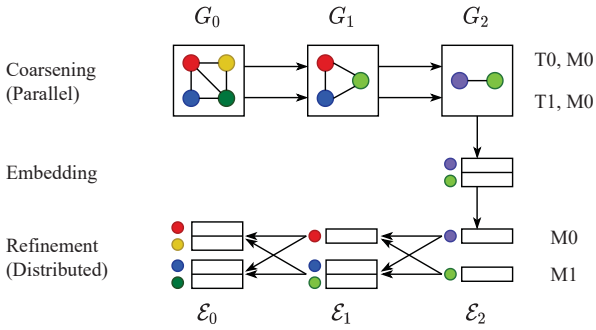


Fig. 1. Overview of DistMILE.

Definition 1 (Graph Embedding on Multiple Machines)**.** Given a graph $G = (V, E)$ and a dimension d where $d \ll |V|$, graph embedding aims to learn a function $f_e : V \rightarrow \mathcal{R}^d$ that maps each node to a d-dimensional space and learn a d-dimensional vector representation for each node in graph G. The learned representations, known as *embeddings*, should capture the structural properties of G. The similarity of embeddings of any two nodes should approximate their distance in the graph. Using M machines each with T threads, it should learn the embeddings in an efficient and scalable manner.

IV. Methodology

In this section, we present a distributed multi-level embedding framework DistMILE which leverages a hybrid of parallel and distributed computing techniques to boost the embedding efficiency. As a distributed version of MILE, DistMILE adopts the same scheme consisting of three phases: graph coarsening, base embedding, and refinement. Fig. 1 shows an example of DistMILE running on two machines each with two threads.

A. Graph coarsening

For graph coarsening, we present a new multi-threaded shared-memory graph coarsening algorithm (see in Algorithm

Algorithm 1 Parallel Graph Coarsening

Input: $G = (V, E)$, T, h, p
Output: Coarsened graph G'

1: Initialize shared arrays $\tilde{W}, Q, \phi$ ▷ Initialization
2: $\mathcal{F} \leftarrow$ Create h hash functions using p
3: **#pragma omp parallel**
4: $\quad i \leftarrow$ omp_get_thread_num()
5: $\quad V_i \leftarrow \{u \mid u < |V| \wedge u \bmod T = i\}$
6: $\quad$ **for all** $u \in V_i$ **do**
7: $\quad\quad$ **for all** edge $(u, v) \in E$ **do**
8: $\quad\quad\quad \tilde{W}_{u,v} \leftarrow W_{u,v} / \sqrt{\sigma(u) \cdot \sigma(v)}$
9: $\quad\quad Q[u] \leftarrow \{f(\mathcal{N}(u)) \mid f \in \mathcal{F}\}$ ▷ SEM
10: Update ϕ for nodes with same signatures
11: **#pragma omp parallel**
12: $\quad i \leftarrow$ omp_get_thread_num()
13: $\quad V_i \leftarrow \{u \mid u < |V| \wedge u \bmod T = i\}$
14: $\quad$ Sort V_i by the number of neighbors in ascending order
15: $\quad$ **for all** $u \in V_i$ **do** ▷ NHEM
16: $\quad\quad v \leftarrow \max_{v \in \mathcal{N}(u),\ \phi(v) = \varnothing} \tilde{W}_{u,v}$
17: $\quad\quad$ **if** $\phi(u) = \varnothing$ **and** $\phi(v) = \varnothing$ **then**
18: $\quad\quad\quad \phi(u) \leftarrow v,\ \phi(v) \leftarrow u$
19: $\quad$ Barrier.wait()
20: $\quad$ Correct matching conflicts in $\phi(V_i)$
21: $\quad$ Build G' in parallel
return G'

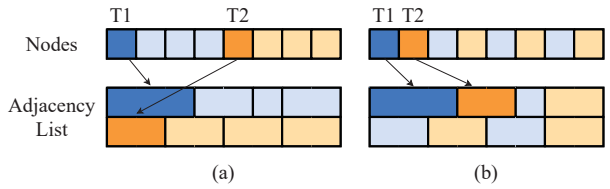


Fig. 2. Two Ways of Workload Distribution.

1). This algorithm shrinks the graph via two matching approaches (SEM and NHEM), which are also used in MILE. DistMILE extends both approaches to parallel execution with reduced workload and synchronization costs.

Choice of Parallel Formulation: Graph coarsening has been used in a variety of multi-level algorithms for other problems. KMetis [11] is a multi-level graph partitioning algorithm, which has two different parallel formulations, namely, ParMetis [12] for distributed-memory systems and mt-Metis [13] for shared-memory systems. Experiment results demonstrate that mt-Metis is more efficient due to low synchronization costs. We conduct a similar comparison in our drilldown experiments, and thus we adopt the multi-threaded shared-memory parallelism for coarsening.

Workload Distribution: A key problem for parallel computing is how to distribute the workload among multiple threads. Data tiling is commonly used for matrix computation. However, the adjacency list and edge weights of a large graph are usually stored in one-dimensional arrays. Note that the workload of SEM and HEM is proportional to the number of edges. We examine two ways for partitioning (see in Fig. 2): (a) divide the nodes into T consecutive chunks, or (b) divide the nodes into T interlaced chunks (lines 5 and 13 in Algorithm 1). Theoretically, the second option has a lower cache miss rate as the threads are likely to request the same data block at a moment. Our analysis reveals that on Yelp, the largest dataset in this work, the second option can save up to 27% of the running time than using the first one.

Initialization: Prior to coarsening, DistMILE first declares three arrays in shared memory. These arrays can be accessed and written by multiple threads at the same time. Specifically, $\tilde{W}$ is of size $|E|$ for normalized edge weights which can be used for NHEM, while Q is of size $|V| \times h$ for node signatures, where h is the number of hash functions used for SEM. ϕ is of size $|V|$ containing the node matching results.

SEM: The goal of SEM is to collapse nodes with the same neighbors. To perform SEM, MILE converts each node's neighbors into a string, then uses a dictionary with these strings as the keys to cluster the nodes. A naive way to parallelize SEM is to create a dictionary in each thread then merge them. However, the overhead of merging multiple dictionaries can be huge on large networks. For example, on Yelp, this parallel scheme is even slower than sequential SEM, while merging the dictionaries takes 70% of the execution time.

In DistMILE, we adopt a new locality-sensitive-hashing (LSH) based SEM approach to avoid the expensive synchronization. LSH is an effective technique that can hash similar items into same buckets with high probability, which has been successfully applied in data clustering [31] and neighbor similarity search [32]. To check if $\mathcal{N}(u) = \mathcal{N}(v)$ for nodes u and v, we follow [33] and estimate the similarity of their neighbors as:

$$\mathbf{Pr}_{f \in \mathcal{F}}\left[f(\mathcal{N}(u)) = f(\mathcal{N}(v))\right] = sim(\mathcal{N}(u), \mathcal{N}(v)) \quad (1)$$

Here $sim(\cdot, \cdot) \in [0, 1]$ is a similarity function and $\mathcal{F}$ is a family of hash functions.

Specifically, DistMILE adds h randomly sampled MinHash functions to $\mathcal{F}$ in line 2. Each function $f_i \in \mathcal{F}$ is defined as $f_i(U) = \min_{v \in U}\left[(a_i * v + b_i) \mod p\right]$ for $U \subseteq V$, where p is a large prime number and a_i, b_i are randomly sampled from $[0, p)$. After applying these functions in line 9, each node has a h-dimensional vector as its *signature*. If two nodes have the same signature, they are very likely to be structurally equivalent. Instead of performing $O(|V|^2)$ pairwise queries for similarity search in classic LSH, our proposed approach divides the signature matrix by node degrees and matches the nodes by comparing the signatures in parallel.

NHEM: Mt-Metis proposed a multi-threaded matching approach called *unprotected matching*. In this approach, each thread updates the matching result ϕ simultaneously, followed by checking if there is a *matching conflict*, i.e., $\phi(\phi(u)) \neq u$ for each node u. Nodes involved in a matching conflict will not be collapsed with other nodes. Unprotected matching outperforms the other two matching approaches in [13] in terms

of efficiency due to reduced synchronization and memory accesses.

To perform NHEM in parallel, DistMILE leverages unprotected matching with several optimizations. Race conditions that occur in line 18 of Algorithm 1 can cause matching conflicts, which can further induce a larger coarsened graph increasing the running time of base embedding. To reduce the matching conflicts, DistMILE additionally checks the matching status of both node u and its selected neighbor v in line 17 before updating ϕ. This optimization greatly reduces the matching conflicts with little additional overhead. For example, on Flickr, it helps decreasing the graph size by 13% after being coarsened 5 times.

To further avoid matching conflicts, we adopt a heuristic strategy to choose the mode of NHEM. Specifically, when the number of nodes in the graph is no less than a threshold n_c, DistMILE performs NHEM in parallel, otherwise it runs the serial version as MILE does.

B. Base Embedding

In the phase of base embedding, DistMILE calls a graph embedding algorithm to learn the node representations on the coarsened graph. Since the graph has been significantly shrunk in the prior phase, this is more efficient than embedding the original graph. Note that DistMILE is model-agnostic that means the user is able to determine which method is exploited for embedding and whether the model is trained on CPUs or GPUs.

C. Refinement

Distributed deep learning recently became a popular solution for machine learning on large graphs. By increasing the number of processing units and their computation power, GNN models can be trained on distributed machines efficiently. For embedding refinement, we propose a distributed training paradigm, as shown in Algorithm 2. It uses the base embeddings from the second phase as input and leverages a hybrid of distributed learning and parallel computing in order to achieve the maximum speedup.

Minibatch Training and GraphSAGE: In MILE, a refinement model is trained with the entire node data in the graph, which consumes too much memory and works only for training on a single machine. In order to train the model in parallel, it is necessary to adopt data parallelism and split the data into multiple *mini-batches*. Mini-batch training divides the training task into multiple sub-tasks that can be assigned to distributed machines, reducing memory usage. In our implementation of DistMILE, we use GraphSAGE as our GNN model for refinement, which is compatible with mini-batch training. A hyperparameter b is used to control the batch size, which is always no more than the number of nodes assigned to the current machine (see in line 4).

Distributed Learning with Horovod: In order to balance the workload on each machine, DistMILE initially allocates the equal number of nodes to the machines in line 3. Then each machine partitions its nodes into minibatches of size b in line 10. During model training, each time the device fetches one mini-batch and compute the gradients locally. To train across a cluster of machines, DistMILE is incorporated with Horovod for distributed training. Horovod provides an easy-to-use interface to scale a single-machine training program to run across multiple machines. Horovod can gather the gradients from different devices and apply the averaged gradients to each device, which follows [34] to normalize the loss on each machine by the total minibatch size. In addition to synchronizing the model parameters distributed on each machine, Horovod is able to achieve a high scaling efficiency which is appreciated in distributed training.

Algorithm 2 Distributed Training

Input: $G = (V, E)$, s, b
Output: Trained model
1: rank $\leftarrow$ horovod.rank()
2: procs $\leftarrow$ horovod.size()
3: $V_r \leftarrow \left[\frac{|V|}{\text{procs}} \cdot \text{rank},\ \frac{|V|}{\text{procs}} \cdot (\text{rank}+1)\right)$
4: $b \leftarrow \min\{|V_r|,\ b\}$
5: $\tilde{\mathcal{N}}_r \leftarrow$ shared array ▷ Sample neighbors
6: **#pragma omp parallel for**
7: **for all** $u \in V_r$ **do**
8: Sample s neighbors for node u, update $\tilde{\mathcal{N}}_r[u]$
9: $\tilde{\mathcal{N}} \leftarrow \text{AllGather}(\tilde{\mathcal{N}}_r)$
10: $B \leftarrow$ Partition V_r into mini-batches of size b
11: **for** each batch in B **do**
12: Synchronously train the model with current batch
return model

Optimizations: The implementation of DistMILE has been optimized for training on different hardwares, especially for GPU training. Although training on GPU is efficient, the small GPU memory usually limits the training performance. For example, MILE needs to transfer data of all nodes between CPU and GPU, which could be a bottleneck when the model is trained on a large graph. In order to reduce the memory usage, DistMILE is able to remove the unnecessary training data (i.e., embeddings and sampled neighbors) for the current batch in addition to leveraging mini-batch training. Furthermore, for CPU computations such as neighbor sampling (in lines 6-8), DistMILE leverages a multi-threaded shared-memory parallelism when it runs on multi-core machines. With these optimizations, DistMILE can scale better across different systems.

V. Experiments

A. Experiment setup

1) Datasets: We use datasets that have been used in MILE and other prior work of network embedding. Statistics of these datasets are shown in Table II.

2) Baselines: Our experiments evaluate the performance of different multi-level model-agnostic embedding frameworks, namely, MILE and DistMILE.

TABLE II
DATASET INFORMATION

Dataset	# Nodes	# Edges	# Classes
Blog	10.3K	334.0K	39
Flickr	80.5K	5.9M	195
Youtube	1.1M	3.0M	47
Yelp	8.9M	39.8M	22

MILE: We run MILE with multiple graph embedding methods as follows:

- NetMF [3]: NetMF is a representative matrix-factorization based method for network embedding. In our experiment, NetMF is trained on CPU with the rank set to 1024 and the window size set to 10.
- DeepWalk [4]: DeepWalk is a popular random-walk based embedding method. It runs on CPU with parallelism available for sampling random walks. We set the length of random walks as 80, the number of walks for each node to 10, and the window size to 10.
- SDNE [6]: SDNE utilizes deep neural networks to perform graph embedding. The model is trained with 5 epochs with the sizes of hidden layers set to $[300, 500]$. We set $\alpha = 0.2$ and $\beta = 10.0$. In this experiment, the model is trained and applied on a single GPU.

The experiments of MILE are conducted on a single machine. MILE only utilizes CPU parallelism to train the refinement model. As MILE treats the phase of base embedding as a black box, the selected base embedding method can determine whether it is trained on CPU or GPU and whether or not to run in parallel. For GraphSAGE, we set $s = 10^2$, $b = 10^5$, and the learning rate to 10^{-3} if not specified.

DistMILE: Following the setup for MILE, DistMILE runs with the three embedding methods above and utilizes inherent parallelism. The differences are that DistMILE leverages shared-memory CPU parallelism for graph coarsening on a single machine, and it uses both CPUs and GPUs on all machines for embedding refinement. We empirically set $n_c = 10^4$, $h = 16$, and $p = 2^{31} - 1$ for coarsening. we use the same parameters of GraphSAGE in MILE.

Fair comparison of MILE and DistMILE: To study and compare the performance of both frameworks in a fair situation, several adaptions are made in our experiments. In the phase of embedding refinement, we add the mini-batch technique into MILE. The benefits are two-fold: it reduces the memory usage for refinement so MILE can scale to larger dataset, and MILE with mini-batch training is equivalent to the serial version of DistMILE which makes the comparison more fair. Furthermore, we adopt a new comparison paradigm. Due to matching conflicts caused by the multi-threaded coarsening, the coarsened graph in DistMILE is consequently larger than that in MILE. Therefore, we use a new threshold *coarsen depth* to control graph coarsening. Specifically, both DistMILE and MILE repeatedly coarsen the graph until the number of nodes is no more than a given threshold n_m. In our experiment, we set $n_m = |V| \cdot 2^{-m}$ where m is the coarsen depth varying within different ranges depending on the network size.

3) Metrics: Following the experiment setting of MILE, we evaluate the quality of the embeddings through the task of node classification. More specifically, we use the node embeddings as the features for classification, and report the average F1-scores after a 10-fold cross validation.

4) System specification: We conduct the experiments on a cluster of four Linux machines with a 28-core Intel Xeon E5-2680 CPU, an NVIDIA Tesla v100 GPU, and 128GB of RAM on each machine. Our method is implemented in Python, where we use the pymp[2] and horovod[3] packages for parallel computing and distributed learning, respectively. In addition, we use MVAPICH2-GDR[4] for MPI communication. Our code

B. Analysis

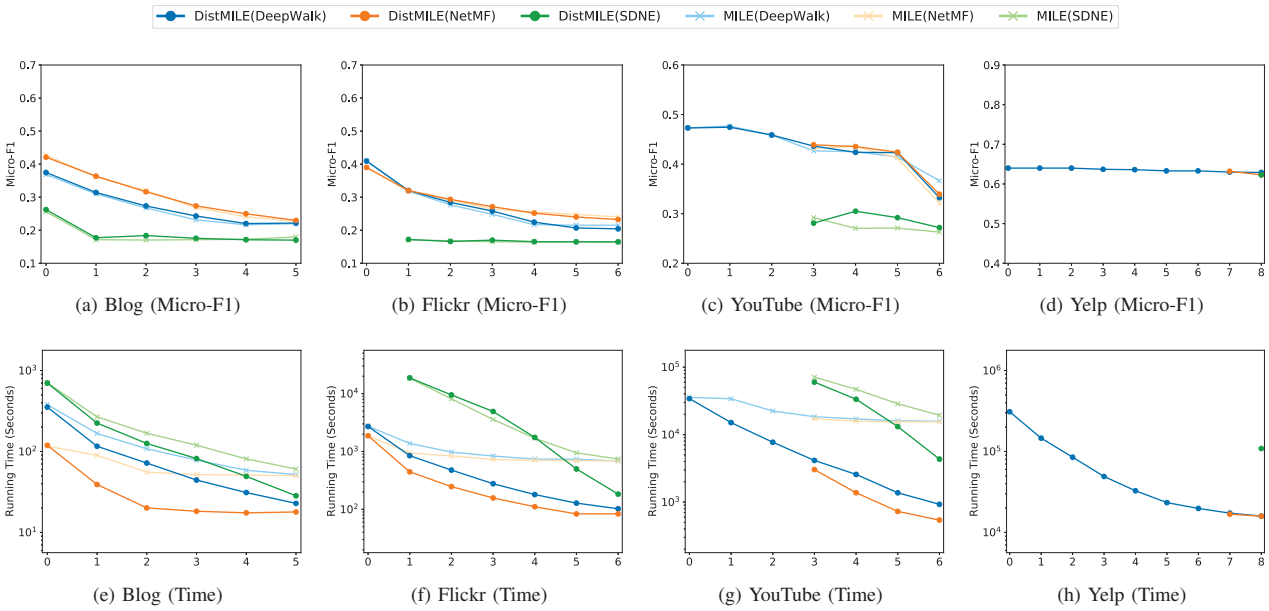
This section of experiments compares DistMILE with MILE through node classification to see their quality and running time at different coarsen depths. Fig. 3 shows the results, where we assign dark/light colors to DistMILE/MILE with the same base embedding method. Note that setting coarsen depth $m = 0$ means that it directly applies the original embedding method with no coarsening and refinement. We report the running time in seconds.

Both DistMILE and MILE expedite embedding: Multi-level embedding framework can help existing embedding methods scale to larger networks. For example, NetMF cannot be applied directly on YouTube/Yelp due to memory constraints. However, by using the coarsening-refinement strategy, DistMILE and MILE can overcome the limitation of CPU memory and embed large networks, which demonstrates that DistMILE and MILE have a relatively lower demand of computing hardware compared to the original embedding methods.

In addition, DistMILE and MILE are able to reduce the time for embedding on large networks. For example, the execution of SDNE on Flickr cannot finish within 2 days, while DistMILE and MILE using SDNE with coarsen depth $m = 1$ can finish in 5 hours. Increasing the coarsen depth can further boost both frameworks. For $m = 6$ on Flickr, the execution of MILE(SDNE) takes around 12 minutes, and DistMILE(SDNE) can even finish in only 3 minutes. Similar improvements are also observed in the results on YouTube and Yelp.

DistMILE has comparable quality: The embedding qualities of both frameworks with various embedding methods are evaluated through Micro-F1 scores in the node classification task, as shown in Fig. 3(a)-3(d). It can be seen that the impact of embedding on coarsened graphs varies with different combinations of base embedding methods and datasets. For example, the quality of MILE/DistMILE with DeepWalk declines a little after each time of coarsening ($m > 0$ on Blog/Flickr), while the Micro-F1 scores of SDNE in both frameworks are still comparable ($m > 1$ on all graphs) with respect to the original version.

[2]https://github.com/classner/pymp
[3]https://github.com/horovod/horovod
[4]https://mvapich.cse.ohio-state.edu



(a) Blog (Micro-F1) (b) Flickr (Micro-F1) (c) YouTube (Micro-F1) (d) Yelp (Micro-F1)

(e) Blog (Time) (f) Flickr (Time) (g) YouTube (Time) (h) Yelp (Time)

Fig. 3. Comparison of Overall Performance of MILE and DistMILE

In comparison of DistMILE and MILE, we observe that DistMILE and MILE with the same base embedding method always have similar Micro-F1 scores. For example, on Flickr, the distance of any two lines denoting the same base embedding method is at most 0.01. This demonstrate that the employment of hybrid high-performance computing techniques in DistMILE does not hurt the embedding quality while effectively boosting the embedding process.

DistMILE is significantly faster: In Fig. 3(e)-3(h), we plot the running time of all baselines in a logarithmic scale. This includes the running time of all three phases, and we later compare MILE and DistMILE's time in the phases of coarsening and refinement respectively. The results show that DistMILE is faster than MILE in almost every case. Compared to MILE, DistMILE achieves up to $3\times$ speedup on Blog with little loss of quality. On Flickr, DistMILE's speedup is up to $8\times$ (NetMF, $m = 6$). On small datasets, the speedup of DistMILE is mainly contributed by parallel coarsening and distributed training of refinement model, but limited by the data volume. When it comes to larger datasets, DistMILE is able to completely show the advantage of high-performance computing. On YouTube ($m = 6$), the speedup of DistMILE using NetMF with respect to MILE increases to 28. The speedup on large networks comes from the distributed parallel computing for sampling neighbors during the training of GraphSAGE.

The speedup of DistMILE with respect to MILE is affected by the running time of base embedding. Since both frameworks provides the user with full control of this phase, DistMILE's speedup decreases when the graph is not sufficiently coarsened. When we decrease the coarsen depth, each pair of lines corresponding to the same embedding method becomes closer. We notice that on Flickr with $m = 2$ or 3,

TABLE III
COMPARISON OF SHARED-MEMORY AND DISTRIBUTED-MEMORY COARSENING.

m	Time for Coarsening (sec)	
	Shared-Memory	Distributed-Memory
1	8.39	120.60
2	12.07	172.91
3	15.46	214.72
4	26.70	243.59
5	33.84	258.11
6	37.83	264.35

DistMILE using SDNE is slightly slower than MILE because the coarsened graph in DistMILE is much larger and SDNE is inefficient for large graphs. However, in most cases, DistMILE is much faster than MILE.

DistMILE scales better over Yelp: With the employment of high-performance computing techniques, DistMILE has an improved scalability than MILE on the largest dataset Yelp. Considering the huge amount of time for sampling in MILE, we reduce the number of sampled neighbors s to 10. Unfortunately, MILE is unable to finish within the time limit even under the relaxed setting due to the extremely long time on sampling. In contrast, DistMILE can embed Yelp with all embeding methods. The running time of DeepWalk on Yelp is about 4 days, while DistMILE reduces it to only 4 hours with coarsen depth 8. Furthermore, the embedding quality of DistMILE does not decline much when we increase the coarsen depth, which demonstrates that DistMILE can learn embeddings with a comparable quality while significantly reducing the running time.

Choice of Parallel Formulation for Coarsening: In addi-

TABLE IV
SPEEDUP FOR COARSENING

Dataset	m	MILE (sec)	DistMILE (sec)	Speedup
Flickr	1	81.40	8.39	**9.70**
	2	112.52	12.07	9.32
	3	135.19	15.46	8.74
	4	152.50	26.70	5.71
	5	163.22	33.84	4.82
	6	164.32	37.83	4.34
Yelp	1	731.46	105.84	6.91
	2	2099.41	339.17	6.19
	3	2988.71	474.86	6.29
	4	3593.85	496.04	7.25
	5	3990.16	533.19	**7.48**
	6	4292.87	574.92	7.47
	7	4446.70	624.47	7.12
	8	4869.37	653.31	7.45

TABLE V
SPEEDUP FOR REFINEMENT

Dataset	m	MILE (sec)	DistMILE (sec)	Speedup
Flickr	1	567.91	30.90	18.38
	2	525.58	27.40	19.18
	3	517.26	24.98	**20.70**
	4	509.94	25.24	20.20
	5	506.80	30.18	16.79
	6	490.62	30.87	15.89
YouTube	1	16973.99	582.04	29.16
	2	15512.14	355.87	43.59
	3	14836.22	277.62	53.44
	4	14541.72	252.90	57.50
	5	14608.88	249.96	**58.44**
	6	14851.52	264.16	56.22

tion to the shared-memory formulation of parallel graph coarsening, we implement a distributed-memory version to explore different choices of parallel formulations. In the distributed-memory formulation, the graph is initially partitioned into M subgraphs, where M is the number of machines. During the matching process, each machine communicates the matching information of nodes they share with other machines.

Table III shows the running time of shared-memory and distributed-memory formulations on Flickr, respectively. We observe that the shared-memory formulation is always faster than the distributed version regardless of coarsen depths. At coarsen depth $m = 1$, distributed coarsening finishes in 120 seconds, while shared-memory coarsening only needs 8 seconds. Hence we adopt the shared-memory parallelism on a single machine in the phase of coarsening.

DistMILE's speedup in different phases: Next we study the improvements in running time of DistMILE with respect to MILE in the phase of graph coarsening and embedding refinement. In this experiment, we use DeepWalk as the base embedding method for both frameworks. Note that as the first phase, graph coarsening is not affected by the choice of embedding method. Although the phase of refinement takes the learned embeddings as its input, the workload in this phase does not vary. Therefore, the choice of base embedding method has little impact on the speedup of DistMILE in these two phases.

TABLE VI
IMPACT OF VARYING T ON COARSENING AND FOLLOW-UP EMBEDDING

Dataset	T	Coarsening (sec)	Graph Size	Embedding (sec)	Total (sec)
Flickr	1	162.35	5043.0	189.60	351.95
	2	88.20	5071.4	199.40	287.60
	4	48.84	5164.2	211.10	259.94
	8	29.73	5312.4	217.69	247.43
	16	19.07	5510.0	224.37	243.44
	28	15.46	5727.8	225.95	**241.38**
Yelp	1	5553.74	30985.0	849.85	6403.60
	2	3099.86	30080.2	714.39	3814.25
	4	1753.55	30290.0	734.23	2487.78
	8	1076.99	31918.2	874.68	1951.67
	16	743.54	30885.6	834.03	1577.57
	28	653.31	30247.4	730.36	**1383.64**

Table IV shows the running time of both frameworks and the speedup of DistMILE for coarsening. We can observe that DistMILE achieves up to $10\times$ speedup with respect to MILE. For coarsen depth $m = 1$ on Flickr, both DistMILE and MILE coarsen the original network twice, as MILE spends 81 seconds and DistMILE only takes 8 seconds. On Yelp, the speedup of DistMILE is up to $7.5\times$. It can decrease to around $5\times$ when the dataset is sparse or the graph has been coarsened multiple times ($m \geq 4$ on Flickr) since there is relatively less parallel computation on these networks. On the other hand, given a time limit for coarsening, DistMILE is able to coarsen the original network more times, which can boost the following two phases. For example, MILE coarsens Flickr into a graph containing 20000 nodes ($m = 1$) within 81 seconds, while DistMILE can reduce the number of nodes to 715 ($m = 6$) using only half of the time.

DistMILE has a more considerable speedup in the phase of refinement (see in Table V). The speedup of DistMILE is up to $21\times$ on Flickr and increases to $58\times$ on YouTube. For coarsen depth $m \geq 3$ on YouTube, MILE needs 4 hours to train and apply the GraphSAGE model for embedding refinement, but DistMILE only takes about 4 minutes. Thanks to the leverage of distributed learning and parallel sampling in the GraphSAGE model, DistMILE is able to learn the refined representations within minutes.

C. DistMILE Drilldown

This section of experiments explores the performance of DistMILE under different setups. We first evaluate the coarsening performance with different numbers of used threads and observe its impact on follow-up embedding. In addition, we study how DistMILE works on different computing clusters by varying the size of machines. We also explore the impact of learning rate and batch size on distributed learning for refinement. Without loss of generality, here we use DeepWalk as the base embedding method.

Varying the Number of Threads for Coarsening: DistMILE leverages multi-threading shared-memory parallel computing for graph coarsening. Now we vary the number of

TABLE VII
IMPACT OF VARYING THE NUMBER OF MACHINES ON REFINEMENT

M	Training (sec)	Refinement (sec)	Total (sec)	Scaling Efficiency
1	1060.92	62369.38	63430.30	100
2	558.29	31557.25	32115.54	98.75
3	371.61	21267.34	21638.95	97.71
4	286.54	16324.09	16610.64	95.47
5	259.92	13226.12	13486.04	94.07
6	217.01	11131.82	11348.83	93.15
7	186.70	9506.89	9693.59	93.48
8	184.97	8795.72	8980.69	88.22

threads T used for coarsening to see how it affects the first phase and the follow-up base embedding. With increasing T from 1 to 28, we measure the running time of coarsening and embedding as well as the size of coarsened graph, as shown in Table VI. Note that we keep the 28-core parallelism in DeepWalk unchanged for fair comparison.

We first observe that using more threads results in faster graph coarsening, which is expected. For example, coarsening Flickr with only one thread takes 162 seconds, while increasing T to 28 can reduce the running time to only 15 seconds, which is an 11-fold speedup. On Yelp, using 28 threads can save 1.4 hours over single-threaded coarsening.

On the other hand, increasing the number of used threads may lead to slower base embedding by increasing the size of the coarsened graph. This is because matching conflicts are more likely to happen when we use more threads, hence more nodes remain unmatched after coarsening. With varying T from 1 to 28 on Flickr, the graph size increases by 700, which leads to 35 more seconds in the follow-up embedding. We notice that the sizes of coarsened graphs do not always strictly decrease with the increasing T, which is because DistMILE coarsens the graph for different times when T varies. Using less threads can reduce matching conflicts so DistMILE can more quickly meet the requirement of coarsening. Examples include $T = 8$ and 16 (one more time of coarsening) on Yelp.

To better understand the tradeoff between the efficiencies of coarsening and base embedding, Table VI shows the total running time of both phases. We observe that using less than 28 threads is relatively inefficient on both datasets, therefore we recommend setting the number of threads equal to the number of CPU cores for maximum overall speedup.

Varying the Number of Machines: DistMILE exploits distributed learning to train the refinement model across multiple machines. To learn how DistMILE leverages computing resources, we measure the running time and scaling efficiency of the refinement phase on Yelp, as shown in Table VII.

We first observe that the phase of refinement in DistMILE is dramatically accelerated thanks to the leverage of high-performance computing techniques. On a single machine, DistMILE needs around 18 hours to learn the representations for the fine-grained graph. With four machines used, it finishes within 4.7 hours. The running time is further reduced to 2.4 hours when 8 machines are used. Additionally, DistMILE achieves a high scaling efficiency. With 4 machines or less, its scaling efficiency is at least 95%. Even when 8 machines are used, the scaling efficiency declines only slightly to 88%.

TABLE VIII
IMPACT OF VARYING THE LEARNING RATE ON MICRO-F1

Learning Rate	Blog	Flickr	YouTube	Yelp
0.01	0.228	0.234	0.427	0.623
0.001	**0.243**	**0.260**	**0.439**	**0.633**
0.0001	0.212	0.237	0.409	0.626

TABLE IX
IMPACT OF VARYING THE BATCH SIZE

b	Training (sec)	Applying (sec)	Memory Usage	Micro-F1
500	284.05	12132.22	1.20	0.627
1,000	334.12	6443.27	1.46	0.627
2,000	328.80	4288.91	1.96	0.629
5,000	323.21	2929.13	2.97	0.631
10,000	311.44	2468.45	3.18	0.631
20,000	302.98	2249.71	4.98	0.631
50,000	292.15	2019.47	16.99	0.632
100,000	279.90	1868.11	30.82	0.632

Varying the Learning Rate: We evaluate the quality of learned embeddings with different learning rates from $\{0.01, 0.001, 0.0001\}$ that are also used by the authors of GraphSAGE [14]. In Table VIII, we observe that setting the learning rate to 0.001 outperforms the other two choices on Flickr by 11% in terms of Micro-F1. While the three options achieve similar Micro-F1 scores on Yelp, using 0.001 is slightly better. Similar results are shown on Blog and YouTube.

Varying the Batch Size: Moving the batch size b from 500 to 100000 on Yelp, we observe that large batch sizes can reduce the exact time of training and applying the refinement model (see in Table IX). Our choice of $b = 100,000$ in the main experiment achieves the best efficiency among all choices. More specifically, the change in batch sizes has limited impact on the training time, which ranges from 280 to 330 seconds. However, using large batch sizes can dramatically reduces the time of applying the model. By increasing the batch size, the time of applying can be reduced from 3.4 hours ($b = 500$) to only 30 minutes ($b = 100,000$).

On the other hand, using a large batch size leads to a higher memory usage. For example, when $b \leq 10,000$, the refinement phase uses up to 3GB GPU memory, while the usage increase to about 30 GB if the batch size increase to $100,000$. This is because most GPU memory is consumed by the embeddings of sampled neighbors, which is $O(bsd)$. We observe that there is no quality loss when the model is trained with large batch sizes, which is consistent with previous work [34].

VI. CONCLUSION AND FUTURE WORK

In this paper, we propose a distributed multi-level embedding framework DistMILE to further enhance the scalability of graph embedding. First, we present a new multi-threaded

parallel algorithm for graph coarsening which reduces both synchronization cost and race conditions in coarsening. Second, DistMILE keeps the follow-up phase of base embedding a black box, and it is compatible with any embedding method trained on CPUs or GPUs. Third, DistMILE adopts a distributed training paradigm for embedding refinement with a high scaling efficiency, which takes full advantages of high-performance computing techniques. Our framework can learn the representations of comparable quality while achieving a high speedup with respect to MILE, significantly improving the scalability of graph embedding methods. An interesting direction for future research is implement other distributed refinement models to further enhance the efficiency.

ACKNOWLEDGEMENTS

This material is supported by the National Science Foundation (NSF) under grants OAC-2018627, CCF-2028944, and CNS-2112471. Any opinions, findings, and conclusions in this material are those of the author(s) and may not reflect the views of the respective funding agency.

REFERENCES

[1] R. Hu, C. C. Aggarwal, S. Ma, and J. Huai, "An embedding approach to anomaly detection," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 385–396.

[2] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee, "Billion-scale commodity embedding for e-commerce recommendation in alibaba," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 839–848.

[3] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, and J. Tang, "Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec," in *Proceedings of the eleventh ACM international conference on web search and data mining*, 2018, pp. 459–467.

[4] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.

[5] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[6] D. Wang, P. Cui, and W. Zhu, "Structural deep network embedding," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 1225–1234.

[7] J. Liang, S. Gurukar, and S. Parthasarathy, "Mile: A multi-level framework for scalable graph embedding," *In ICWSM, also available as arXiv preprint:1802.09612*, 2021.

[8] H. Chen, B. Perozzi, Y. Hu, and S. Skiena, "Harp: Hierarchical representation learning for networks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[9] T. A. Akyildiz, A. A. Aljundi, and K. Kaya, "Gosh: Embedding big graphs on small hardware," in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–11.

[10] R. Ramanath, H. Inan, G. Polatkan, B. Hu, Q. Guo, C. Ozcaglar, X. Wu, K. Kenthapadi, and S. C. Geyik, "Towards deep and representation learning for talent search at linkedin," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, ser. CIKM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2253–2261. [Online]. Available: https://doi.org/10.1145/3269206.3272030

[11] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed computing*, vol. 48, no. 1, pp. 96–129, 1998.

[12] G. Karypis and V. Kumar, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *Siam Review*, vol. 41, no. 2, pp. 278–300, 1999.

[13] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 225–236.

[14] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1025–1035.

[15] P. Veličković, W. Fedus, W. L. Hamilton, P. Liò, Y. Bengio, and R. D. Hjelm, "Deep graph infomax," in *International Conference on Learning Representations*, 2019.

[16] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," in *International Conference on Learning Representations*, 2020.

[17] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.

[18] A. Gibiansky, "Bringing hpc techniques to deep learning," http://research.baidu.com/bringing-hpc-techniques-deep-learning/, 2017.

[19] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker *et al.*, "Large scale distributed deep networks," 2012.

[20] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The mvapich project: Transforming research into high-performance mpi library for hpc community," *Journal of Computational Science*, p. 101208, 2020.

[21] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[22] C. Deng, Z. Zhao, Y. Wang, Z. Zhang, and Z. Feng, "Graphzoom: A multi-level spectral approach for accurate and scalable graph embedding," in *The International Conference on Learning Representations (ICLR)*, 2020.

[23] N. Kipf Thomas and W. Max, "Variational graph auto-encoders," in *NeurIPS Workshop on Bayesian Deep Learning*, 2016.

[24] Z. Zhu, S. Xu, J. Tang, and M. Qu, "Graphvite: A high-performance cpu-gpu hybrid system for node embedding," in *The World Wide Web Conference*, 2019, pp. 2494–2504.

[25] J. Qiu, L. Dhulipala, J. Tang, R. Peng, and C. Wang, "Lightne: A lightweight graph processing system for network embedding," in *Proceedings of the 2021 International Conference on Management of Data (SIGMOD 2021)*, June 2021.

[26] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich, "Pytorch-biggraph: A large-scale graph embedding system," *arXiv preprint arXiv:1903.12287*, 2019.

[27] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, no. 2, Mar. 2020. [Online]. Available: https://doi.org/10.1145/3377454

[28] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[29] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[30] E. De Coninck, S. Bohez, S. Leroux, T. Verbelen, B. Vankeirsbilck, P. Simoens, and B. Dhoedt, "Dianne: a modular framework for designing, training and deploying deep neural networks on heterogeneous distributed infrastructure," *Journal of Systems and Software*, vol. 141, pp. 52–65, 2018.

[31] H. Koga, T. Ishibashi, and T. Watanabe, "Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing," *Knowledge and Information Systems*, vol. 12, no. 1, pp. 25–53, 2007.

[32] V. Satuluri and S. Parthasarathy, "Bayesian locality sensitive hashing for fast similarity search," *arXiv preprint arXiv:1110.1328*, 2011.

[33] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, 2002, pp. 380–388.

[34] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.