Distributed Graph Embedding with Information-Oriented Random Walks

Peng Fang
Huazhong University of Science and Technology
China
fangpeng@hust.edu.cn

Arijit Khan
Aalborg University
Denmark
arijitk@cs.aau.dk

Siqiang Luo*
Nanyang Technological University
Singapore
siqiang.luo@ntu.edu.sg

Fang Wang*
Huazhong University of Science and Technology
China
wangfang@hust.edu.cn

Dan Feng
Huazhong University of Science and Technology
China
dfeng@hust.edu.cn

Zhenli Li
Huazhong University of Science and Technology
China
lizhenli@hust.edu.cn

Wei Yin
Huazhong University of Science and Technology
China
weiyin_hust@hust.edu.cn

Yuchao Cao
Huazhong University of Science and Technology
China
caoyuchao@hust.edu.cn

ABSTRACT

Graph embedding maps graph nodes to low-dimensional vectors, and is widely adopted in machine learning tasks. The increasing availability of billion-edge graphs underscores the importance of learning efficient and effective embeddings on large graphs, such as link prediction on Twitter with over one billion edges. Most existing graph embedding methods fall short of reaching high data scalability. In this paper, we present a general-purpose, distributed, information-centric random walk-based graph embedding framework, DistGER, which can scale to embed billion-edge graphs. DistGER incrementally computes information-centric random walks. It further leverages a multi-proximity-aware, streaming, parallel graph partitioning strategy, simultaneously achieving high local partition quality and excellent workload balancing across machines. DistGER also improves the distributed Skip-Gram learning model to generate node embeddings by optimizing the access locality, CPU throughput, and synchronization efficiency. Experiments on real-world graphs demonstrate that compared to state-of-the-art distributed graph embedding frameworks, including KnightKing, DistDGL, and Pytorch-BigGraph, DistGER exhibits 2.33×–129× acceleration, 45% reduction in cross-machines communication, and >10% effectiveness improvement in downstream tasks.

PVLDB Reference Format:
Peng Fang, Arijit Khan, Siqiang Luo, Fang Wang, Dan Feng, Zhenli Li, Wei Yin, Yuchao Cao. Distributed Graph Embedding with Information-Oriented Random Walks. PVLDB, 16(7): 1643 - 1656, 2023.
doi:10.14778/3587136.3587140

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/RocmFang/DistGER.

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 16, No. 7 ISSN 2150-8097.
doi:10.14778/3587136.3587140

1 INTRODUCTION

Graph embedding is a widely adopted operation that embeds a node in a graph to a low-dimensional vector. The embedding results are used in downstream machine learning tasks such as link prediction [60], node classification [5], clustering [34], and recommendation [45]. In these applications, graphs can be huge with millions of nodes and billions of edges. For instance, the Twitter graph includes over 41 million user nodes and over one billion edges, and it has extensive requirements for link prediction and classification tasks [19]. The graph of users and products at Alibaba also consists of more than two billion user-product edges, which forms a giant bipartite graph for its recommendation tasks [56]. A plethora of random walk-based graph embedding solutions [17, 18, 39, 47, 49] are proposed. A random walk is a graph traversal that starts from a source node, jumps to a neighboring node at each step, and stops after a few steps. Random-walk-based embeddings are inspired by the well-known natural language processing model, word2vec [32]. By conducting sufficient random walks on graphs, substantial graph structural information is collected and fed into the word2vec (Skip-Gram) to generate node embeddings. Compared with other graph embedding solutions such as graph neural networks [20, 51, 53–55] and matrix factorization techniques [40, 41, 58, 64, 66], random walk-based methods are more flexible, parallel-friendly, and scale to larger graphs [62].

While graph embedding is crucial, the increasing availability of billion-edge graphs underscores the importance of scaling graph embedding. The inherent challenge is that the number of random walks required increases with the graph size. For example, one representative work, node2vec [18] needs to sample many node pairs to ensure the embedding quality, it takes months to learn node embeddings for a graph with 100 million nodes and 500 million edges by 20 threads on a modern server [66]. Some very recent work, e.g., HuGE [17] attempts to improve the quality of random walks according to the importance of nodes. Though this method can remove redundant random walks to a great extent, the inherent complexity remains similar. It still requires more than one week to learn embeddings for a billion-edge Twitter graph on a modern server, hindering its adoption to real-world applications. Another line of work turns to use GPUs for efficient graph embedding. For example, some recent graph embedding frameworks (e.g., [59, 70]) simultaneously perform graph random walks on CPUs and embedding training on

GPUs. However, as the computing power between GPUs and CPUs differ widely, it is typically hard for the random walk procedure performed on CPUs to catch up with the embedding computation performed on GPUs, causing bottlenecks [44, 67]. Furthermore, this process is heavily related to GPUs' computing and memory capacity, which can be drastically different across different servers.

Recently, distributed graph embedding, or computing graph embeddings with multiple machines, has attracted significant research interest to address the scalability issue. Examples include KnightKing [63], Pytorch-BigGraph [26], and DistDGL [68]. KnightKing [63] optimizes the walk-forwarding process for node2vec and brings up several orders of magnitude improvement compared to a single-server solution. However, it may suffer from redundant or insufficient random walks that are attributed to a routine random walk setting, resulting in low-quality training information for the downstream task [17]. Moreover, the workload-balancing graph partitioning scheme that it leverages fails to consider the randomness inherent in random walks, introducing higher communication costs across machines and degrading its performance. Facebook proposes Pytorch-BigGraph [26] that leverages graph partitioning technique and parameter server to learn large graph embedding on multiple CPUs based on PyTorch. However, the parameter server used in this framework needs to synchronize embeddings with clients, which puts more load on the communication network and limits its scalability. Amazon has recently released DistDGL [68], a distributed graph embedding framework for graph neural network model. However, its efficiency is bogged down by the graph sampling operation, e.g., more than 80% of the overhead is for sampling in the GraphSAGE model [20], and the mini-batch sampling used may trigger delays in gradient updates causing inefficient synchronization. In conclusion, although the distributed computation frameworks have shown better performance than the single-server and CPU-GPU-based solutions, significant rooms exist for further improvement.

Our DistGER system. We present a newly designed distributed graph embedding system, DistGER, which incorporates more effective information-centric random walks such as HuGE [17] and achieves super-fast graph embedding compared to state-of-the-arts. As a preview, compared to KnightKing, Pytorch-BigGraph, and DistDGL, our DistGER achieves $9.3\times$, $26.2\times$, and $51.9\times$ faster embedding on average, and easily scales to billion-edge graphs (§6). Due to information-centric random walks, DistGER embedding also shows higher effectiveness when applied to downstream tasks.

Three novel contributions of DistGER are as follows. **First** and foremost, since the information-centric random walk requires measuring the effectiveness of the generated walk on-the-fly during the walking procedure, it inevitably introduces higher computation and communication costs in a distributed setting. DistGER resolves this by showing that the effectiveness of a walking path can be measured through incremental information, avoiding the need for full-path information. DistGER invents incremental information-centric computing (InCoM), ensuring $O(1)$ time for on-the-fly measurement and maintains constant-size messages across computing machines. **Second**, considering the randomness inherent in random walks and the workload balancing requirement, DistGER proposes multi-proximity-aware, streaming, parallel graph partitioning (MPGP) that is adaptive to random walk characteristics, increasing the local partition utilization. Meanwhile, it uses a dynamic workload constraint for the partitioning strategy to ensure load-balancing. **Finally**, different from the existing random walk-based embedding techniques, DistGER designs a distributed Skip-Gram learning model (DSGL) to generate node embeddings and implements an end-to-end distributed graph embedding system. Precisely, DSGL leverages global-matrices and two local-buffers for node vectors to improve the access locality, thus reducing cache lines ping-ponging across multiple cores during model updates; then develops multi-windows shared-negative samples computation to fully exploit the CPU throughput. Moreover, a hotness block-based synchronization mechanism is proposed to synchronize node vectors efficiently in a distributed setting.

Our contributions and roadmap. We propose an efficient, scalable, end-to-end distributed graph embedding system, DistGER, which, to our best knowledge, is the first general-purpose, information-centric random walk-based distributed graph embedding framework.

• We introduce incremental information-centric computing (InCoM) to address computation and communication overheads due to on-the-fly effectiveness measurements during information-oriented random walks in a distributed setting (§3.1).

• We propose multi-proximity-aware, streaming, parallel graph partitioning (MPGP) that achieves both higher local partition utilization and load-balancing (§3.2).

• We develop a distributed Skip-Gram learning model (DSGL) to generate node embeddings by improving the access locality, CPU throughput, and synchronization efficiency (§4).

• We conduct extensive experiments on five large, real-world graphs to demonstrate that DistGER achieves much better efficiency, scalability, and effectiveness over existing popular distributed frameworks, e.g., KnightKing [63], DistDGL [68], and Pytorch-BigGraph [26]. In addition, DistGER generalizes well to other random walk-based graph embedding methods (§6).

We discuss preliminaries and a baseline approach in §2, related work in §5, and conclude in §7. Additional related work and empirical results, proof of theorems are given in our full version [15].

2 PRELIMINARIES AND BASELINE

We design an end-to-end distributed system for effective and scalable embedding of large graphs via random walks. To this end, we first discuss relevant works on random walk-based sequential graph embedding (§2.1) and distributed systems for random walks on graphs (§2.2). Then, we propose a baseline distributed system for random walk-based graph embedding by combining the above two methods (§2.3), discuss its limitations and scopes of improvements, which leads to introducing our ultimate system, DistGER in §3 and §4. Table 1 explains the most important notations. DistGER handles undirected and unweighted graphs by default, but can support directed and weighted graphs (higher edge weights imply stronger connectivity) [15]. DitsGER uses the *Compressed Sparse Row* (CSR) [38] format to store graph data, where directed edges are stored with their source nodes and undirected edges are stored twice for both directions. For each weighted edge, CSR stores a tuple containing its destination node and edge weight.

2.1 Random-walks Based Graph Embedding

These graph embedding algorithms are inspired by the well-known natural language processing model, word2vec [32]: They transform

Table 1: Frequently used notations

Notation	Meaning
$G = (V, E)$	G: undirected, unweighted graph; V: set of nodes; E: set of edges
$\varphi(u)$	embedding or vector representation of node u, having dimension d
w	window size of context in the Skip-Gram
$N(u)$	neighbors of node u
L	random walk length starting from a node
r	number of random walks per node
$H(X)$	entropy of random variable X with possible values $x_1, x_2, \ldots, x_n$

a graph into a set of random walks through sampling methods, treat each random walk as a sentence, and then adopt word2vec (Skip-Gram) to generate node embeddings from the sampled walks.

Node2vec. A most representative algorithm in the aforementioned category is node2vec [18], as given below.

Random walk method. Given a graph $G = (V, E)$, two nodes $u, v \in V$, and we suppose a walker is currently at node u. Node2vec defines the transition probability from u to v as $P(u, v) = \frac{\pi_{uv}}{Z}$, where π_{uv} is the unnormalized transition probability from u to v, and Z is the normalization constant defined as $\sum_{v \in N(u)} \pi_{uv}$. Node2vec defines a second-order random walk. Assume that a walker just traversed node t and now resides at node u (u is a neighbor of t). Next, it will select a node v from u's neighbors. The un-normalized transition probability π_{uv} is defined by d_{tv}, which is the shortest path distance between nodes t and v: If d_{tv} is 0, 1, 2, respectively, then the corresponding π_{uv} is $1/p$, 1, $1/q$. Hyperparameters p and q are called return and in-out parameters, respectively. $d_{tv} = 0$ means that t and v are the same node, i.e., the walker goes back to t, which is a BFS-like exploration, thus setting a small p obtains a "local view" in the graph with respect to the start node. $d_{tv} = 1$ means that v is a neighbor of t, and $d_{tv} = 2$ denotes a DFS-like exploration to get a "global view" in the graph, which can be attained by a small q.

Features learning for graph embedding. Features learning maps $\varphi : V \longrightarrow R^d$ from nodes to feature representations (node embeddings). Since node2vec captures node representations based on the Skip-Gram model [32] that maximizes the co-occurrence probability between words within a window w in a sentence, the objective is:

$$\underset{\varphi}{\operatorname{argmax}} \frac{1}{|V|} \sum_{j=1}^{|V|} \sum_{-w \leq i \leq w} \log p(u_{j+i}|u_j) \tag{1}$$

The generated walks are used as a corpus with vocabulary V, where u_{j+i} denotes a context node in a window w, and $p(u_{j+i}|u_j)$ indicates the probability to predict the context node. The basic Skip-Gram formulates $p(u_j|u_{j+i})$ as the softmax function. Existing methods generally speed-up training with negative sampling [31].

$$\log p(u_j|u_{j+i}) \approx \log \sigma(\varphi_{in}(u_{j+i}) \cdot \varphi_{out}(u_j)) + \sum_{k=1}^{K} \mathbb{E}_{u_k \sim Pn(u)} [\log \sigma(-\varphi_{in}(u_{j+i}) \cdot \varphi_{out}(u_k))] \tag{2}$$

Here, $\sigma(x) = \frac{1}{1+exp(-x)}$ is the sigmoid function, and the expectations are computed by drawing random nodes from a sampling distribution $Pn(u), \forall u \in V$. Typically, the number of negative samples K is much smaller than $|V|$ (e.g., $K \in [5, 20]$).

Complexity analysis. Assume that the number of walks per node is r, walk length L, embedding dimensions d, window size w, and the number of negative samples K. The time complexity of node2vec random-walk procedure is $O(r \cdot L \cdot |V|)$. For feature learning, the corpus size $C = r \cdot L$. Let us denote the complexity of the unit operation of predicting and updating one node's embedding as o. The Skip-Gram with the negative sampling only needs $K + 1$ words to obtain a probability distribution (Eq. 2), thus the time complexity of node2vec feature learning is $O(C \cdot w \cdot (K+1) \cdot o)$. Since each node in the Skip-Gram model needs to maintain two embeddings φ_{in} and φ_{out} for the parameter updates, the space complexity of node2vec, which refers to the parameter sizes, is $O(|V|d)$.

Drawbacks. Despite the flexibility in exploring node representations (local-view vs. global-view), node2vec incurs high time overhead. It leverages a routine random walk configuration (usually, L=80 and r=10) to generate walks, similar to most existing random walk-based graph embedding methods, which limits the efficiency and scalability on large-scale graphs. Indeed, this *one-size-fits-all* strategy cannot meet the specific requirements of different real-world graphs. For instance, the high-degree nodes are usually located in dense areas of a graph, they might require longer and more random walks to capture more comprehensive features; while for the low-degree nodes, if treated equally, it may introduce redundancy into generated walks, thus limiting the scalability.

HuGE. The recent work, HuGE [17] attempts to resolve the routine random walk issue of node2vec and proposes a novel information-oriented random walk mechanism to achieve a concise and comprehensive representation in the sampling procedure.

Random walk method. First, HuGE leverages a hybrid random walk strategy, which considers both node degree and the number of common neighbors in each walking step. Common neighbors represent potential information between nodes, e.g., node similarity [47]. For random walks, high-degree nodes are revisited more, and walks starting from them can obtain richer information by traveling around their local neighbors [27]. The un-normalized transition probability from node u to the next-hop node v is:

$$\alpha(u, v) = \frac{1}{deg(u) - Cm(u, v)} \times \max\left\{\frac{deg(u)}{deg(v)}, \frac{deg(v)}{deg(u)}\right\} \tag{3}$$

where $deg(u)$ is the degree of u, and $Cm(u, v)$ denotes the number of their common neighbors. Thus, $\frac{1}{deg(u)-Cm(u,v)}$ indicates the similarity between the current node u and the next-hop node v, the ratio grows with higher $Cm(u, v)$, since $deg(u)$ is fixed. The max function assigns a weight to the transition probability from u to v, indicating the influence of a high degree node on its neighbors.

At the current node u, HuGE randomly chooses v from $N(u)$ as a candidate node, the acceptance probability for v as the next-hop node is $P(u, v)$, and if v is rejected, which happens with probability $1-P(u, v)$, the walker backtracks to u and repeats a random selection again from $N(u)$, known as the *walking-backtracking* strategy [27]. $P(u, v)$ is defined as $Z(\alpha(u, v))$, where HuGE normalizes $\alpha(u, v)$ via $Z(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, which is widely-applied in machine learning. For edge weight $w(u, v)$, we define $P(u, v) = Z(\alpha(u, v) \cdot w(u, v))$.

Second, in contrast to a one-size-fits-all strategy, HuGE proposes a heuristic walk length strategy to measure the effectiveness of information during walk based on entropy (H). Mathematically, let us denote the random walk starting at the source node u as $W_u^L = \{v_u^1, v_u^2, v_u^3, \ldots, v_u^L\}$, where v_u^k denotes the k-th node on the walk. The probability of the occurrence of a specific node v on the walk is $\frac{n(v)}{L}$, where $n(v)$ is the number of occurrences of v on the generated walk. The information entropy of the generated walk is:

$$H\left(W_u^L\right) = -\sum_{v \in W_u^L} \frac{n(v)}{L} \log \frac{n(v)}{L} \tag{4}$$

With increasing L, as the occurrence probability for a specific node in a generated walk gradually stabilizes, $H\left(W_u^L\right)$ initially grows with L until it converges. HuGE characterizes the correlation between $H\left(W_u^L\right)$ and L by linear regression and calculates the coefficient of determination (R^2) to determine the termination of a random walk.

$$R\left(H(W_u^L), L\right) = \frac{\sum_{i=1}^{L^*}\left(H\left(W_u^{L(i)}\right) - \overline{H(W_u^L)}\right)\left(L(i) - \overline{L}\right)}{\sqrt{\sum_{i=1}^{L^*}\left(H\left(W_u^{L(i)}\right) - \overline{H(W_u^L)}\right)^2}\sqrt{\sum_{i=1}^{L^*}\left(L(i) - \overline{L}\right)^2}} \tag{5}$$

$\overline{H\left(W_u^L\right)}$ and $\overline{L}$ are the mean of the respective series for $1 \leq i \leq L^*$, and L^* is the optimal walk length for the current walk. As L grows and $H(W_u^L)$ stabilizes, $R^2(H, L)$ also decreases and converges to 0, since their linear correlation diminishes. HuGE sets $R^2(H, L) < \mu$ as the walk termination condition. Setting a smaller μ generates longer walks, introducing redundant information; while too large μ may not ensure good coverage of graph properties during sampling, since too short walks are generated. Based on our experimental results, good quality walk lengths are attained with $\mu = 0.995$.

Third, HuGE also proposes a heuristic number of walks strategy. The corpus is generated by multiple (r) random walks from each node. Following [39], if the degree distribution of a connected graph follows power-law, the frequency in which nodes appear in short random walks will also follow a power-law distribution. Inspired by this observation, HuGE empirically analyzes the similarity between the two distributions via relative entropy. Formally, the node degree distribution is expressed as $p(v) = \frac{deg(v)}{\sum_{v \in V} deg(v)}$. We denote the number of occurrences of v in the generated corpus as $ocn(v)$. The probability distribution for such appearances in the corpus is $q(v) = ocn(v)/\sum_{v \in V} ocn(v)$. The relative entropy from p to q is:

$$D(q\|p) = \sum_{i=1}^{r^*} \frac{deg(v)}{\sum deg(v)} \log \frac{deg(v) \sum ocn(v)}{ocn(v) \sum deg(v)} \tag{6}$$

Here, r^* is the optimal number of walks from a source node. With increasing r, the difference $D_r(q\|p)$ gradually converges, which means that the probability distribution of nodes' occurrences in the generated corpus has stabilized.

$$\Delta D_r(q\|p) = |D_r(q\|p) - D_{r-1}(q\|p)| \tag{7}$$

HuGE leverages $\Delta D_r(q\|p) \leq \delta$ as the termination condition. Based on our experimental results, $\delta = 0.001$ usually produces a good number of random walks per source node.

Features learning for graph embedding. The features learning uses the Skip-Gram model, and similar to node2vec, follows Eq. 1 and 2.

Complexity analysis. The time complexity of HuGE random-walk procedure is $O(r' \cdot L' \cdot |V|)$, where the optimal number of walks per node is r' (decided by $\Delta D_r(q\|p) \leq \delta$) and the average walk length is L' (decided by $R^2(H, L) < \mu$ for each walk). However, the complexity of measuring $H(W)$ and $R\left(H(W), L\right)$ at each step of a walk is $O(L)$, where L is the current walk length. Thus, the overall computational workload of HuGE becomes quadratic in the walk length, though the average walk length can be smaller than that in Node2Vec. The time complexity of the feature learning phase

Algorithm 1 HuGE-D walking procedure

```
Input: current node $u$, candidate node $v$, Walker $W$, HuGE parameter $\mu$
Output: walker state updates
    sendStateQuery($u$, $v$, $W$)
 1: $P(u,v) = Z\left(\frac{1}{deg(u) - Cm(u,v)} \cdot \max\left\{\frac{deg(u)}{deg(v)}, \frac{deg(v)}{deg(u)}\right\}\right)$ // Eq. 3
    getStateQueryResult($W$, $P(u,v)$)
 2: generate a random number $\eta \in [0,1]$
 3: if $P(u,v) > \eta$ then
 4:     $W.path$.append($v$), $W.cur = v$, $W.steps$ ++
 5:     $L = W.steps$
 6:     compute $H(W)$ and $R\left(H(W), L\right)$ // Eq. 4, 5
 7:     if $R^2\left(H(W), L\right) < \mu$ then
 8:         terminate the walk
 9:     else
10:         generate another candidate node $t$ of $v$
11:         sendStateQuery($v$, $t$, $W$)
12: else
13:     backtrack to $u$ and generate another candidate node $v'$ of $u$
14:     sendStateQuery($u$, $v'$, $W$)
```

remains $O(C' \cdot w \cdot (K+1) \cdot o)$, but the average corpus size $C' = r' \cdot L'$ is smaller, since generally $r' < r$ and $L' < L$.

Drawbacks. The workload of HuGE is quadratic in the walk length. Moreover, HuGE [17] embedding method is sequential, and there is yet no end-to-end distributed system to support graph embedding via information-oriented random walks. Being sequential, HuGE requires more than one week to learn embeddings for a billion-edge Twitter graph on a modern server.

2.2 Distributed Random Walks on Graphs

KnightKing [63] is a recent general-purpose, distributed graph random walk engine. Its key components are introduced below.

Walker-centric programming model. For higher-order walks (e.g., second-order random walks in node2vec), KnightKing assumes a walker-centric view. KnightKing implements each step of walks by two rounds of message passing, one round for walkers to submit the walker-to-node query messages to check the distance between the previous node $W.prev$ and the candidate node v and to generate the un-normalized transition probability π_{uv}; another round of message passing returns results to walkers about their states, and then the walkers decide the next steps based on the sampling outcome.

KnightKing coordinates many walkers simultaneously based on the Bulk Synchronous Parallel (BSP) model [52]. Walkers are assigned to some computing machine/thread. KnightKing leverages rejection sampling to eliminate the need of scanning all out-edges at the walker's current node. Suppose a walker is currently at node u, the sampling method generates a candidate node v (a neighbor of u) with a transition probability $p(u,v)$. The key idea of rejection sampling is to find an envelop $Q(u) = max(\frac{1}{p}, 1, \frac{1}{q})$: Within the rectangular area covered by the lines $y = Q(u)$ and $x = |N(u)|$, it randomly samples a location (x, y) from this area, if $p(u,v) \geq y$, v is accepted as a successfully sampled node, otherwise, v is rejected, and the method conducts more sampling trials until success.

Workload-balancing graph partition. KnightKing adopts a node-partitioning scheme – each node (together with its edges) is assigned to one computing machine in a distributed setting. It roughly estimates the workload as the sum of the number of edges in each computing machine, and ensures a balance of workloads across computing machines by appropriately distributing the nodes.

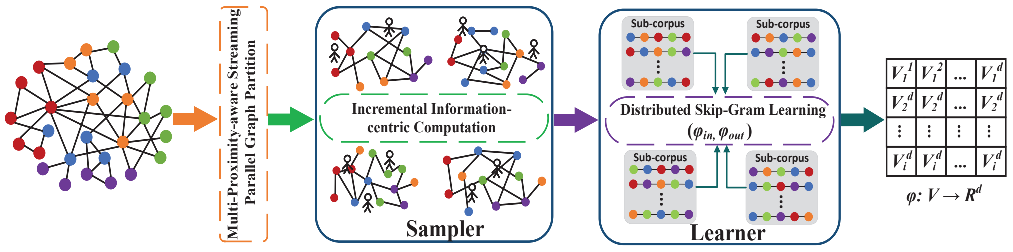

Multi-Proximity-aware Streaming Parallel Graph Partition
Incremental Information-centric Computation
Sampler
Sub-corpus
Sub-corpus
Distributed Skip-Gram Learning
$(\varphi_{in}, \varphi_{out})$
Sub-corpus
Sub-corpus
Learner
V_1^1 V_1^2 ... V_1^d
V_2^d V_2^d ... V_2^d
V_i^d V_i^d ... V_i^d
$\varphi: V \rightarrow R^d$

Figure 1: The workflow of our proposed system: DistGER

Complexity analysis. For walk forwarding, each trial of rejection sampling needs $O(1)$ time. With a reasonable $Q(u)$, there is a good chance for the sampling to succeed within a few trials, thus the walk forwarding computation for one step can be achieved in near $O(1)$ complexity. For communication, each message consists of $[walk\_id, steps, node\_id, previous\_node\_id]$ for node2vec. When a walk crosses a computing machine, it sends $M(1)$, i.e., one constant-length message to another machine. In a distributed environment, assuming P processors and the network bandwidth B, as analyzed in node2vec, the total workload for KnightKing is $O(r \cdot L \cdot |V|)$, with near $O(1)$ computation complexity for each step. Thus, the average time spent in each processor is $O(r \cdot L \cdot |V|/P)$. The communication cost is $O(N \cdot M(1)/B)$, where N is the count of cross-machine messages. Thus, the time spent for KnightKing is: $O(r \cdot L \cdot |V|/P + N \cdot M(1)/B)$.

Drawbacks. KnightKing provides distributed system support for traditional random walks, e.g., the one in node2vec. For information-oriented random walks in HuGE, the walkers need to additionally maintain the generated walking path at each step, and thus the message requires to carry the path information. The message length increases with the length of the walks, thus the efficiency of KnightKing reduces due to extra overheads of computation and communication (elaborated in §2.3). The purely load-balancing partition scheme in KnightKing also introduces high communication cost.

2.3 Baseline: HuGE-D

Our distributed baseline approach, HuGE-D replaces the traditional random walking method in KnightKing with the information-oriented scheme of HuGE, and leverages a full-path computation mechanism.

Full-path computation mechanism. As previously stated, to meet the information measurement requirements, HuGE-D stores the full-path in the message, in addition to the necessary fields such as *walk_id*, *steps*, and *node_id*. Algorithm 1 shows the walking procedure, where we retain the *walking-backtracking* strategy of HuGE, which is similar to rejection sampling in KnightKing.

Complexity analysis. HuGE-D measures the information effectiveness of generated walks at each step. The complexity of measuring $H(W)$ and $R\,(H(W), L)$ at each step is $O(L)$, where L is the current walk length. For the communication cost due to messages, HuGE-D additionally requires carrying the generated walking path information in contrast to KnightKing, so the message cost is also linear in the walk length L, which we denote as $M(L)$. Similar to KnightKing, since the total workload is related to the average walk length L' and the optimal number of walks per node r', the time spent for HuGE-D in distributed setting is: $O(r' \cdot (L')^2 \cdot |V|/P + N \cdot M(L')/B)$, where P, B, and N are the number of processors, network bandwidth, and the count of cross-machine messages, respectively.

Drawbacks. **(1) Computation:** HuGE-D measures the effectiveness of generated walks at each step, which has $O(L)$ complexity, where L is the current walk length. Thus, unlike KnightKing, the computational workload of HuGE-D is quadratic in walk length. **(2) Communication:** KnightKing only sends constant-length messages, e.g., for node2vec each message has $[walk\_id, steps, node\_id, prev\_node\_id]$, but the messages in HuGE-D carry the full-path information (i.e., $[walk\_id, steps, node\_id, path\_info]$) for information measurements, thus the message cost is linear in the walk length. **(3) Partitioning:** The workload-balancing partition in KnightKing fails to consider the large amount of cross-machine communications introduced by the randomness inherent in random walks.

3 THE PROPOSED SYSTEM: DistGER

To address the aforementioned computing and communication challenges of the baseline HuGE-D (§2.3), we ultimately design an efficient and scalable distributed information-centric random walks engine, DistGER, aiming to provide an end-to-end support for distributed random walks-based graph embedding (Figure 1). We discuss our distributed information-centric random walk component (sampler) in §3.1 and multi-proximity-aware, streaming, parallel graph partitioning scheme (MPGP) in §3.2, while our novel distributed graph embedding (learner) is given in §4. Besides providing systemic support for the information-oriented method HuGE, DistGER can also extend its information-centric measurements to traditional random walk-based approaches via the general API to get rid of their routine configurations (§6.6).

3.1 Incremental Information-centric Computing

DistGER introduces incremental information-centric computing (InCoM) to reduce redundant computations and the costs of messages. Recall that the baseline HuGE-D computes $H(W)$ and $R(H(W), L)$ via the full-path computation mechanism to measure the information effectiveness of a walk W at each step, requiring $O(L)$ time at every step of the walk. We show that instead it is possible to update $H(W)$ and $R(H(W), L)$ incrementally with $O(1)$ time cost at every step of the walk. We store local information about ongoing walks at the respective machines, which reduces cross-machine message sizes.

Incremental computing of walk information. Each computing machine stores a partition of nodes from the input graph. For every ongoing walk W, each machine additionally maintains in its *local frequency list* the set of nodes v present in the walk which are also

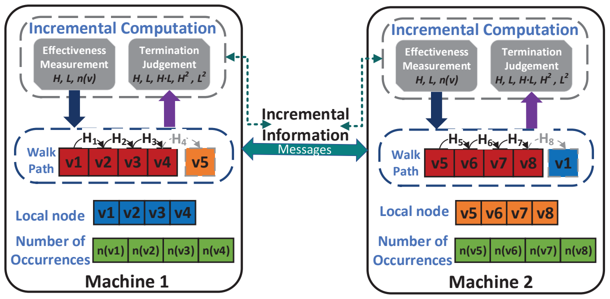


Figure 2: Incremental computing for information-centric random walk

local to that machine, together with their number of occurrences $n(v)$ on the walk. When W terminates, the local frequency list for W is no longer required and is deleted. Theorem 1 summarizes our incremental computing of walk information. The proof is given in our full version [15] due to lack of space.

THEOREM 1. *Consider an ongoing walk W^L with the current length $L \geq 0$, the next accepted node to be added in W^L is v, and $n(v) \geq 0$ is the number of occurrences of v in the walk. In addition to v, both L and $n(v)$ would increase by 1. For clarity, we denote $n(v)$ in W^L and W^{L+1} as $n^L(v)$ and $n^{L+1}(v)$, respectively. The information entropy $H\left(W^{L+1}\right)$ is related to $H\left(W^L\right)$ as follows.*

$$H\left(W^{L+1}\right) = \frac{H\left(W^L\right) \times L - \log T}{L+1}$$
$$\text{where } T = \begin{cases} \dfrac{L^L}{(L+1)^{L+1}} \cdot \dfrac{\left(n^{L+1}(v)\right)^{n^{L+1}(v)}}{\left(n^L(v)\right)^{n^L(v)}}, & \text{if } v \in W^L \\ \dfrac{L^L}{(L+1)^{L+1}}, & \text{if } v \notin W^L \end{cases} \tag{8}$$

Incremental computing for walk termination. To terminate a random walk, HuGE computes and verifies the linear relation between information entropy H and walk length L at every step of the walk. From Eq. 5, $R(H, L)$ can be expressed as:

$$R(H, L) = \frac{E(HL) - E(H)E(L)}{\sqrt{(E(H^2) - E(H)^2)(E(L^2) - (E(L)^2)}} \tag{9}$$

The mean function $E(\ )$ can be computed incrementally.

$$E_p(X) = \frac{1}{p}\sum_{i=1}^{p} X_i = \left(\frac{p-1}{p}\right) E_{p-1}(X) + \frac{X_p}{p} \tag{10}$$
$$E_p(XY) = \frac{(p-1)^2 E_{p-1}(XY) + (p-1)[X_p E_{p-1}(Y) + Y_p E_{p-1}(X)] + X_p Y_p}{p^2}$$

Message size reduction. Due to incremental computation, instead of full-path information, only constant-length messages need to be sent across computing machines: [*walker_id*, *steps*, *node_id*, H, L, $E(H)$, $E(L)$, $E(HL)$, $E(H^2)$, $E(L^2)$].

EXAMPLE 1. *Figure 2 exhibits incremental information-centric computing (*InCoM*). The graph is partitioned into two machines: $\{v_1, v_2, v_3, v_4\}$ in M_1 and $\{v_5, v_6, v_7, v_8\}$ in M_2. A local frequency list is maintained for each ongoing walk at every machine, having # occurrences information only about nodes that are local to a machine. When a local node is added to the walk, # occurrences for this node is updated in this local list. Using the local frequency list,* DistGER *incrementally computes information entropy H, as well as the linear relation R between H and the current path length L to decide on walk termination. If the next accepted node is not at the current machine, the walker only needs to carry the necessary incremental (constant-length) information as a message including walker_id, steps, node_id, H, L, $E(H)$, $E(L)$, $E(HL)$, $E(H^2)$, and $E(L^2)$ to the other machine, and then generate H and R in that machine. Given an 8 bytes space to store a variable, since the messages in baseline* HuGE-D *carry the full-path information (i.e., [walk_id, steps, node_id, path_info]),* HuGE-D *needs $24 + 8L$ bytes per message, where L is the walk length, while* DistGER *only requires the constant size of 80 bytes. If the maximum path length is 80 (commonly used), one message in* DistGER *is up to 8.3× smaller than that in* HuGE-D.

Complexity analysis. The pseudocode of InCoM remains similar to that in Algorithm 1, except that H and R are computed incrementally in Line 6, via Eq. 8 and 10, and we require only $O(1)$ time at each step of the walk. Furthermore, when a walk crosses a machine, it sends a constant-length message to another machine. Following similar analysis as HuGE-D, the time spent for InCoM is: $O(r' \cdot L' \cdot |V|/P + N \cdot M(1)/B)$, where P, B, and N are # processors, network bandwidth, and # cross-machine messages, respectively.

3.2 Multi-Proximity-aware Streaming Partitioning

Graph partitioning aims at balancing workloads across machines in a distributed setting, while also reducing cross-machine communications. Balanced graph partitioning with the minimum edge-cut is an **NP**-hard problem [8]. Instead, our system, DistGER develops multi-proximity-aware, streaming, parallel graph partitioning (MPGP): By leveraging first and second-order proximity measures, we select a good-quality partitioning to ensure that each walker stays in a local computing machine as much as possible, thereby reducing the number of cross-machine communications. The partitioning is conducted in a node streaming manner, hence it scales to larger graphs [1, 36]. We also ensure workload balancing among the computing servers.

Partitioning method. Given a set of partially computed partitions, $P_1, P_2, ..., P_m$, where m denotes # machines, an un-partitioned node v is placed in one of these partitions based on the following objective:

$$\arg\max_{i \in \{1...m\}} \left(PS_1(v, P_i) + PS_2(v, P_i)\right) \times \tau(P_i) \tag{11}$$

$$\text{where } \tau(P_i) = 1 - \frac{|P_i|}{\gamma \times (\sum_{i=1}^{m} |P_i|)/m} \tag{12}$$

$PS_1(v, P_i) = |\{u \in N(v) \cap P_i\}|$ and $PS_2(v, P_i) = \sum_{u \in P_i} |\{N(v) \cap N(u)\}|$ represent the first- and second-order proximity scores, respectively. For edge weight $w(v, u)$, $PS_1(v, P_i) = \sum_{u \in N(v) \cap P_i} w(v, u)$ and $PS_2(v, P_i) = \sum_{u \in P_i} |\{N(v) \cap N(u)\}| \cdot w(v, u)$. Intuitively, PS_1 denotes # neighbors of an unpartitioned node in the target partition, thus a higher value of PS_1 implies that the unpartitioned node should have a higher chance to be assigned to the target partition. Since PS_2 is defined by # common neighbors, which are widely used to measure the similarity of node-pairs during the random walk, a higher value of PS_2 is also in line with the characteristics of random walk. Higher first- and second-order proximity scores increase the chance that a random walker stays in a local computing machine, thereby reducing cross-machine communication. MPGP also introduces a dynamic load-balancing term $\tau(P_i)$, where $|P_i|$ denotes the current number of nodes in P_i, hence it is updated after every un-partitioned node's assignment; and γ is a slack parameter that allows deviation from the exact load balancing. Setting $\gamma = 1$ ensures strict load balancing but hampers partition quality. Meanwhile, setting a larger γ relaxes the load-balancing constraint, and creates a skewed partitioning.

MPGP differs from some of the existing node streaming-based graph partition schemes, e.g., LDG [46] and FENNEL [50] in several ways. **First,** LDG and FENNEL set a maximum size for each partition in advance based on the total number of nodes and tend to assign nodes to a partition until the maximum possible size is reached for that partition. In contrast, we ensure good load balancing across partitions at all times during the partitioning phase. **Second,** we find that LDG and FENNEL cannot partition larger graphs in a reasonable time, for example, they consume more than one day to partition the Youtube [48] graph with 1M nodes and 3M edges, while our proposed MPGP requires only a few tens of seconds (§6), which is due to our optimization methods discussed below.

Optimizations and parallelization. **First,** to measure first-order proximity scores, we apply the Galloping algorithm [12] that can speed-up intersection computation between two unequal-size sets. During our streaming graph partitioning, the partition size gradually increases, hence the Galloping algorithm is quite effective. **Second,** during a second-order proximity score computation, i.e., $PS_2(v, P_i) = \sum_{u \in P_i} |\{N(v) \cap N(u)|$, we only consider those nodes $u \in P_i$ whose contributions to first-order proximity score are non-zero, i.e., $u \in N(v)$. This is because if u is not a neighbor of v, the random walk cannot reach u. **Third,** nodes streaming order could impact the partitioning time and effectiveness. We compare a number of streaming orders [46, 50], e.g., random, BFS, DFS, and their variations. Based on empirical results (§6), we recommend DFS+degree-based streaming for sequential MPGP: Among the un-explored neighbors of a node during DFS, we select the one having the highest degree. This strategy improves the efficiency of the Galloping algorithm. **Fourth**, with large-scale graphs (e.g., Twitter), sequential MPGP, still requires considerable time to partition. Thus, we implement a simple parallelization scheme parallel MPGP (MPGP-P), as follows: We divide the stream into several segments and independently partition the nodes of each segment in parallel via MPGP; finally, we combine the partitioning results of all segments. Based on our empirical results (§6), we recommend BFS+Degree for MPGP-P since it reduces the partition time greatly and the random walk time on a partitioned graph is comparable to that obtained from the sequential version of MPGP.

Complexity analysis. The running time of MPGP is dominated by first- and second-order proximity scores computation for each node, which are computed in parallel for all m partitions. For a first-order proximity score computation via the Galloping algorithm, let the smaller set size be S_1. In the early stages of partitioning, the larger set constitutes the neighbors of an un-partitioned node v, then the time complexity is $O(S_1 \cdot \log |N(v)|)$; while at later stages, the larger set is the partition, thus it takes $O(S_1 \cdot \log(|V|/m))$ time. For the second-order proximity score computing of v, let S_2 be the intersection set size generated by the first-order proximity scores computing of v. As the number of common neighbors for each edge (u, v) is processed in parallel by the Galloping algorithm, the second-order proximity score computing requires $O\left(\frac{S_2}{T} \cdot |N(v)| \cdot \log N^v_{max}\right)$ time, with T threads, where $N^v_{max} = \max_{u \in N(v)} |N(u)|$.

4 DISTRIBUTED EMBEDDING LEARNING

Our learner in DistGER supports distributed learning of node embeddings using the random walks generated by the sampler (Figure 1). We first discuss the shortcomings of state-of-the-art methods on distributed Skip-Gram with negative sampling and provide an overview of our solution (§4.1), then elaborate our three novel improvements (§4.2), and finally summarize our overall approach (§4.3).

4.1 Challenges and Overview of Our Solution

Skip-Gram uses the stochastic gradient descent (SGD) [43]. Parameters update from one iteration and gradient computation in the next iteration may touch the same node embedding, making it inherently sequential. The original word2vec leverages Hogwild [35] for parallelizing SGD. It asynchronously processes different node pairs in parallel and ignores any conflicts between model updates on different threads. However, there are shortcomings in this approach [21, 42]. (1) Since multiple threads can update the same cache lines, updated data needs to be communicated between threads to ensure cache-coherency, introducing cache lines ping-ponging across multiple cores, which results in high access latency. (2) Skip-Gram with negative sampling randomly selects a set of nodes as negative samples for each context node in a context window. Thus, there is a certain locality in model updates for the same target node, but this feature has not been exploited in the original scheme. The randomly generated set of negative samples also introduces random access for parameter updates in each iteration, degrading performance.

As shown in Figure 3(a), a walk denoted as W_1 is assigned to a thread, and there is a sliding window w_1 containing context nodes $\{v_1, v_2, v_4, v_5\}$ and the target node v_3. The Skip-Gram computes the dot-products of word vectors $\varphi_{in}(v_i)$ for a given word $v_i \in \{v_1, v_2, v_4, v_5\}$ and $\varphi_{out}(v_3)$, as well as for a set of K negative samples, $\varphi_{out}(v_k)$ $(v_k \in V)$. Notice that $\varphi_{out}(v_3)$ will be computed four times with four different context words and sets of negative samples, and these dot-products are level-1 BLAS operations [6], which are limited by memory bandwidth. Some state-of-the-art work, e.g., Pword2vec [21] shares negative samples with all other context nodes in each window (Figure 3(b)), and thus converts level-1 BLAS vector-based computations into level-3 BLAS matrix-based computations to efficiently utilize computational resources. However, such matrix sizes are relatively small and still have a gap to reach the peak CPU throughput. Besides, this way of sharing negative samples cannot significantly reduce the ping-ponging of cache lines across multiple cores. To improve CPU throughput, pSGNScc [42] combines context nodes of the negative sample from another window into the current window; and together with the current context nodes, it generates larger matrix batches (Figure 3(c)). Nevertheless, pSGNScc needs to maintain a pre-generated inverted index table to find related windows, resulting in additional space and lookup overheads.

To address the above limitations, we propose a distributed Skip-Gram learning model, named DSGL (Figure 3(d)). DSGL leverages global-matrices and local-buffers for the vectors of context nodes and target/negative sample nodes during model updates to improve the locality and reduce the ping-ponging of cache lines across multiple cores. We then propose a multi-windows shared-negative samples computation mechanism to fully exploit the CPU throughput. Last but not least, DSGL uses a hotness-block based synchronization

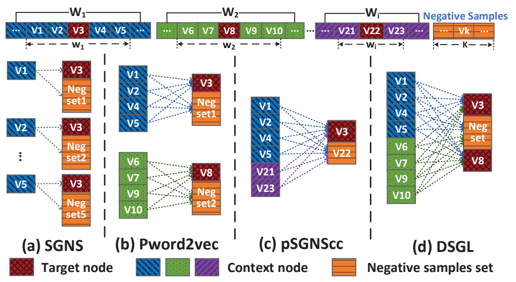


Figure 3: Schematic diagram of (a) Skip-Gram **with negative samples** (SGNS)**, (b)** Pword2vec **[21], (c)** pSGNScc **[42], and (d)** DSGL **(our method)**

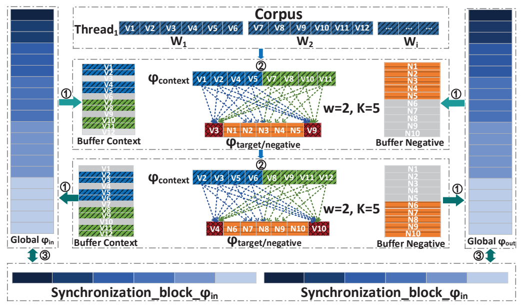


Figure 4: Workflow of our DSGL**: distributed Skip-Gram learning**

mechanism to synchronize the node vectors in a distributed setting. While we design them considering information-oriented random walks, they are generic and have the potential to improve any random walk and distributed Skip-Gram based graph embedding model (§6).

4.2 Proposed Improvements: DSGL

Improvement-I: Global-matrix and local-buffer. As reasoned above, the parallel Skip-Gram with negative sampling suffers from poor locality and ping-ponging of cache lines across multiple cores. Since the model maintains two matrices to update the parameters through forward and backward propagations during training, where the input matrix φ_{in} and output matrix φ_{out} store the vectors of context nodes and target/negative sample nodes, respectively – how to construct these two matrices are critical to the locality of data access.

Since most of the real-world graphs follow a power-law degree distribution [2, 3], we find that the generated corpus sampled from those graphs also has this feature – a few nodes occupy most of the corpus, which are updated frequently during training. In light of this observation, we construct φ_{in} and φ_{out} as *global matrices* in descending order of node frequencies from the generated corpus. It ensures that the vectors of high-frequency nodes stay in the cache lines as much as possible. Recall that node frequencies in the corpus were already computed in the random walk phase (§2.1).

In addition, to avoid cache lines ping-ponging, DSGL uses *local buffers* to update the context and negative sample nodes for each thread, thus the node vectors are first updated in the local buffers within a lifetime (i.e., when a thread processes a walk during training) and are then synchronized to the global matrix. Precisely, since a sliding window w always shifts the boundary and the target node by one node (Eq. 1), each node in a walk will appear as a context node in up to $2w$ sliding windows and as a target node once, thus a context node can be reused up to $2w + 1$ times in the lifetime. It is necessary to consider this temporal locality and build a *local context buffer* for each node vector in a walk. Meanwhile, on-chip caches where randomly-generated negative sample nodes are located, have a higher chance to be updated by multiple threads, and updating their vectors to the global φ_{out} requires low-level memory accesses. To alleviate this, DSGL also constructs a *local negative buffer* for vectors of negative samples during one lifetime. It randomly selects K (negative sample set size) $\times$ L (walk length, i.e., total steps) negative samples into the negative buffer from φ_{out}. Thus, DSGL can use a different negative sample set with the same size (K) at each step. For the target node, due to its lower re-usability compared to context nodes, it is only updated once in global φ_{out} in the lifetime; hence, we do not create a buffer for it. The constructed local buffers require small space, since the sizes of buffers are related to the walk length. The length of the information-centric random walk is much smaller than that of traditional random walks (§2.1).

Improvement-II: Multi-windows shared-negative samples. CPU throughput of Skip-Gram model can be significantly improved by converting vector-based computations into matrix-based computations [10]; however, the batch matrix sizes are relatively small for existing methods [21, 42] and cannot fully utilize CPU resources. Based on this consideration, we design a multi-windows shared-negative samples mechanism: To increase the batch matrix sizes, we batch process the vector updates of context windows from multiple ($\geq$ 2) walks allocated to the same thread.

Consider Figure 3(d), two walks W_1 and W_2 are assigned to the same thread, where w_1 and w_2 are two context windows in W_1 and W_2; v_3 and v_8 are target nodes in w_1 and w_2, respectively. DSGL simultaneously batch updates the node vectors in two context windows w_1 and w_2 based on level-3 BLAS matrix-based computations, the set of negative samples is shared across the batch context nodes, v_3 and v_8 are used as additional negative samples for w_2 and w_1, respectively. After the lifetime of W_1 and W_2, the updated parameters of all context nodes and target/negative samples are written back to φ_{in} and φ_{out}, respectively. Assuming the set of negative samples $K = 5$, the batch matrix sizes of one iteration for Pword2vec [21] (Figure 3(b)) is 4×6, while that of DSGL is extended to 8×7. This utilizes higher CPU throughput and accelerates training without sacrificing accuracy(§ 6). In practice, the number ($\geq$ 2) of multi-windows can be flexibly set according to available hardware resources.

Improvement-III: Hotness-block based synchronization. In a distributed setting, the updated node vectors need to be synchronized with each computing machine. Assuming that the generated corpus is partitioned into m machines, each machine independently processes the local corpus and periodically synchronizes the local parameters with other $m - 1$ machines. For a full model synchronization across computing machines, the communication cost is $O(|V| \cdot d \cdot m)$, where $|V|$ is the total number of nodes and d denotes the dimension of vectors, e.g., for 100 million nodes with 128 dimensions, the full model synchronization needs $\approx$ 102.4 billion messages across 4 machines – it will be difficult to meet the efficiency requirement.

Notice that the node occurrence counts in generated corpus also follows a power-law distribution, implying that high-frequency nodes have a higher probability of being accessed and updated during training. Following this, we propose a hotness-block based synchronization mechanism in DSGL. Instead of full synchronization, we only conduct more synchronization for hot nodes than that for low-frequency nodes. Since our global matrices are constructed based on the descending order of node frequencies in the generated corpus (Improvement-I), it provides a favorable condition to achieve hotness-block based synchronization. The global matrices are partitioned into several blocks (i.e., hotness-blocks) based on the same frequency of nodes in the corpus, and are denoted as $B(i)$, $0 < i \leq ocn_{max}$, where ocn_{max} is the largest number of occurrences of any node in the corpus. We randomly sample one node from each hotness-block; for all these sampled nodes, we synchronize their vectors across all computing machines during one synchronization period. Due to the node frequency skewness in the corpus, for each node in $B(i)$, the probability of it being sampled for synchronization is inversely proportional to $|B(i)|$. Thus, we ensure that the hot nodes are synchronized more during the entire training procedure, while the low-frequency nodes would have relatively less synchronization. Compared to the full model synchronization mechanism, our synchronization cost is $O(ocn_{max} \cdot d \cdot m)$, where $ocn_{max} << |V|$, indicating that it can significantly reduce the load on network, while keeping the parameters on each computing machine updated aptly.

4.3 Putting Everything Together

First, DSGL constructs two global matrices φ_{in} and φ_{out} in descending order of node frequencies from the generated corpus. It leverages a pipelining construction during the random walk phase, one thread is responsible for counting the frequency of each node on its local walk; at the end of all random walks, these counts from computing machines are aggregated to construct global φ_{in} and φ_{out}. Next, DSGL uses a multi-windows shared-negative samples computation for each thread. As shown in Figure 4, two walks W_1 and W_2 are assigned to $Thread_1$, suppose # multi-windows = 2 and negative sample set size $K = 5$. For maintaining access locality and reducing cache lines ping-ponging, two local buffers per thread are constructed for the context (blue and green blocks) and negative sample (orange blocks) nodes. The target nodes are denoted as red blocks. Initially, the two buffers will respectively load the vectors associated with nodes from φ_{in} and φ_{out}, then $Thread_1$ proceeds over W_1 and W_2 with matrix-matrix multiplications, and the updated vectors are written to the buffers at each step. After the lifetime of W_1 and W_2, the latest node vectors in buffers are written back to global φ_{in} and φ_{out}. DSGL periodically synchronizes the updated vectors across multiple computing machines by hotness-block based synchronization.

5 RELATED WORK

Graph embedding algorithms. Sequential graph embedding techniques [9] fall into three categories. Matrix factorization-based algorithms [40, 41, 58, 64, 66] construct feature representations based on the adjacency or Laplacian matrix, and involve spectral techniques [4]. Graph neural networks (GNNs)-based approaches [20, 51, 53–55] focus on generalizing graph spectra into semi-supervised or supervised graph learning. Both techniques incur high computational overhead and DRAM dependencies, limiting their scalability to large graphs. Random-walk methods [16–18, 39, 47] transform a graph into a set of random walks through sampling and then adopt Skip-Gram to generate embeddings. They are more flexible, parallel-friendly, and scale to larger graphs [62]. HuGE+ [16] is a recent extension of HuGE [17], which considers the information content of a node during the next-hop node selection to improve the downstream task accuracy. It uses the same HuGE information-oriented method to determine the walk length and number of walks per node (§2.1), and hence the efficiency is similar to that of HuGE.

Graph embedding systems and distributed embedding. To address efficiency challenges with large graphs, recently proposed GraphVite [70] and Tencent's graph embedding system [59] follows sampling-based techniques on a CPU-GPU hybrid architecture, simultaneously performing graph random walks on CPUs and embedding training on GPUs. Marius [33] optimizes data movements between CPU and GPU on a single machine for large-scale knowledge graphs embedding. Seastar [61] develops a novel GNN training framework on GPUs with a vertex-centric [30] programming model. We deployed DistGER on GPU, but it does not provide a significant improvement, especially for large-scale graphs. Similar to the above works, computing gaps between CPUs and GPUs and the limited memory of GPUs still plague the efficiency of graph embedding.

Other approaches attempt to scale graph embeddings from a distributed perspective. HET-KG [13] is a distributed system for knowledge graph embedding. It introduces a cache embedding table to reduce communication overheads among machines. AliGraph [69] optimizes sampling operators for distributed GNN training and reduces network communication by caching nodes on local machines. Amazon has released DistDGL [68], a distributed graph embedding framework for GNN model with mini-batch training based on the Deep Graph Library [57]. Pytorch-Biggraph [26] leverages graph partitioning and parameter servers to learn large graph embeddings on multiple CPUs in a distributed environment based on PyTorch. However, the efficiency and scalability of DistDGL and Pytorch-BigGraph are affected by parameter synchronization (as demonstrated in §6). ByteGNN [67] is a recently proposed distributed system for GNNs, with mini-batch training and two-level scheduling to improve parallelism and resource utilization, and tailored graph partitioning for GNN workloads. Since ByteGNN is not publicly available yet, we cannot compare it in our experiments. There are also approaches attempting to address computational efficiency challenges with new hardware [25, 29, 37]. Although these approaches open up opportunities for training larger datasets or providing more accelerations, their programming compatibility and prohibitive expensive still pose challenges.

6 EXPERIMENTAL RESULTS

We evaluate the efficiency (§6.2) and scalability (§6.3) of our proposed method, DistGER by comparing with HuGE-D (baseline), KnightKing [63], PyTorch-BigGraph (PBG) [26], and Distributed DGL (DistDGL) [68]. We also compare the effectiveness (§6.4) of generated embeddings on link prediction. Finally, we analyze efficiency due to individual parts of DistGER (§6.5) and the generality of DistGER for other random walk-based embeddings (§6.6). Our codes and datasets are at [14].

Table 2: **Datasets statistics** ($K = 10^3, M = 10^6, B = 10^9$)

Graph	#nodes	#edges
FL	80.51 K	5.90 M
YT	1.14 M	2.99 M
LJ	2.24 M	14.61 M
OR	3.07 M	117.19 M
TW	41.65 M	1.47 B

Table 3: **Avg. memory footprint (GB) of** DistGER **and** KnightKing **on each machine**

	Sampling		Training	
Graph	KnightKing	DistGER	KnightKing	DistGER
FL	0.66	**0.41**	1.31	**0.86**
YT	4.11	**1.36**	4.73	**4.26**
LJ	7.65	**1.95**	6.38	**5.49**
CO	10.98	**3.27**	8.52	**6.86**
TW	out-of-memory	**20.18**	out-of-memory	**67.16**

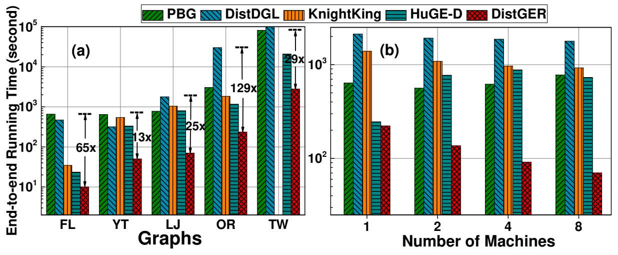


Figure 5: **Efficiency (a) and Scalability (b):** PBG **[26],** DistDGL **[68],** KnightKing **[63],** HuGE-D **(baseline),** DistGER **(ours)**

6.1 Experimental Setup

Environment. We conduct experiments on a cluster of 8 machines with 2.60GHz Intel ® Xeon ® Gold 6240 CPU with 72 cores (hyper-threading) in a dual-socket system, and each machine is equipped with 192GB DDR4 memory and connected by a 100Gbps network. The machines run Ubuntu 16.04 with Linux kernel 4.15.0. We use GCC v9.4.0 for compiling DistGER, KnightKing, and HuGE-D, and use Python v3.6.15 and torch v1.10.2 as the backend deep learning framework for Pytorch-BigGraph and DistDGL.

Datasets. We employ five widely-used, real-world graphs (Table 2): *Flickr* (FL) [48], *Youtube* (YT) [48], *LiveJournal* (LJ) [65], *Com-Orkut* (OR) [24], and *Twitter* (TW) [23]. The first two graphs are selected for multi-label node classification (empirical results given in [15]) with distinct number of node labels 195 and 47, respectively, where labels in *Flickr* represent interest groups of users, and *Youtube*'s labels represent groups of viewers that enjoy common video genres. The last four graphs are used in link prediction. We also use synthetic graphs [11] (up to 1 billion nodes, 10 billion edges) and a real-world *UK graph* [7] (100M nodes, 3.7B edges) to assess the scalability of DistGER. Considering the default settings of popular random walk-based methods (e.g., Deepwalk, node2vec, HuGE), we use their undirected version.

Competitors. We compare DistGER against three state-of-the-art distributed graph embedding frameworks: the distributed random walk engine, KnightKing https://github.com/KnightKingWalk/KnightKing [63]; the distributed multi-relations based graph embedding system, PyTorch-BigGraph (PBG) https://github.com/facebookresearch/PyTorch-BigGraph [26] – designed by Facebook; and the distributed graph neural networks-based system, DistDGL https://github.com/dmlc/dgl [68] – recently proposed by Amazon. We also implement HuGE-D, a distributed version of information-centric random walk-based graph embedding (HuGE [17]), on top of KnightKing, served as our baseline. Since KnightKing and HuGE-D provide distributed support only for random walk without that for embedding learning, we generate their node embeddings using Pword2vec https://github.com/IntelLabs/pWord2Vec [21], the most popular distributed Skip-Gram system released by Intel.

Parameters. For DistGER and HuGE-D random walks, we set parameters μ=0.995, δ=0.001 based on information measurements (§2), while KnightKing uses L=80 and r=10 that are routine configurations in the traditional random walk-based graph embedding [18, 39, 63]. For DistGER, KnightKing, and HuGE-D training, we set the sliding window size w=10, number of negative samples K=5, and synchronization period=0.1 sec [21], and additionally, multi-windows number=2, γ=2 for DisrGER. For fair comparison across all systems, we set the embedding dimension d=128 that is commonly used [17, 18, 39, 47, 49, 66], and report the average running time for each epoch. For task effectiveness evaluations, we find the best results from a grid search over learning rates from 0.001-0.1, # epochs from 1-30, and # dimensions from 128-512.

6.2 Efficiency and Memory Use w.r.t. Competitors

We report the end-to-end running times of PBG, DistDGL, KnightKing, HuGE-D, and DistGER on five real-world graphs with the cluster of 8 machines in Figure 5 (a). The reported end-to-end time includes the running time of partitioning, random walks (for random walk-based frameworks), and training procedures. DistGER significantly outperforms the competitors on all these graphs, achieving a speedup ranging from 2.33× to 129×. Recall that DistGER is a similar type of system as KnightKing and HuGE-D, and our key improvements are discussed in §3 and in §4. Analogously, Figure 5 (a) exhibits that our system, DistGER achieves an average speedup of 9.25× and 6.56× compared with KnightKing and HuGE-D. Notice that we fail to run KnightKing on the largest *Twitter* dataset because its routine random walk strategy requires more main memory space. The advantage of information-centric random walk in HuGE is almost wiped out in HuGE-D due to on-the-fly information measurements and the higher communication costs in a distributed setting. The multi-relation-based PBG leverages a parameter server to synchronize embeddings between clients, resulting in more load on the communication network. As a result, PBG is on average 26.22× slower than DistGER. For graph neural network-based system DistDGL, due to the long running time of graph sampling (e.g., taking 80% of the overhead for the GraphSAGE), it is highly inefficient than other systems. For the billion-edge *Twitter* graph, it does not terminate in 1 day. Table 3 shows DistGER's average memory footprint on each machine of the 8-machine cluster. Compared to same type of system KnightKing, DistGER requires less memory for sampling and training.

6.3 Scalability w.r.t. Competitors

Figure 5 (b) shows end-to-end running times of all competing systems on the *LiveJournal* graph, as we increase # machines from 1 to 8 to evaluate scalability. DistGER achieves better scalability than the other four distributed systems. PBG leverages a parameter server and a shared network filesystem to synchronize the parameters in the distributed model. When the number of machines increases, PBG puts

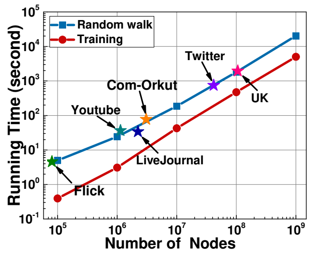


Figure 6: Scalability of DistGER **on synthetic graphs, where Y-axis is in log-scale**

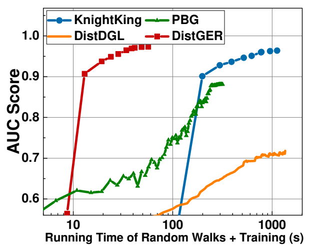


Figure 7: The influence of running time on embedding quality for DistGER **and competitors**

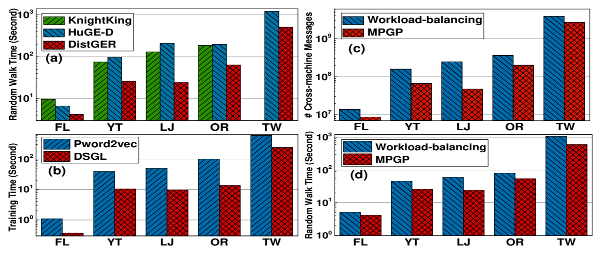


Figure 8: (a) Random walk efficiency, (b) training efficiency, (c) # cross-machine messages, (d) random walk efficiency for MPGP **(ours) and workload-balancing scheme** (KnightKing)

more load on the communications network, resulting in poor scalability. Likewise, DistDGL is bounded by the synchronization overhead for gradient updates, limiting its scalability. Both KnightKing and HuGE-D suffer from higher communication costs during random walks, due to their only workload-balancing partitioning scheme (§2.2, §6.5). Since HuGE-D is implemented on top of KnigtKing, it exhibits worse scalability due to high communication costs and on-the-fly information measurements in a distributed setting (§2.3). In comparison, DistGER incorporates multi-proximity-aware streaming graph partitioning and incremental computations to reduce both communication and computation costs, it also employs hotness-block based parameters synchronization during training to dramatically reduce the pressure on network bandwidth. Hence, DistGER achieves better scalability than other systems. Due to space limitations, we omit DistGER's scalability results on other graphs, which exhibit similar trends. On *Twitter*, the end-to-end running times DistGER on 1, 2, 4, and 8 machines are 3090s, 1739s, 1197s, and 746s, respectively, while on *Com-Orkut*, the results are 304s, 204s, 149s, and 89s, respectively. The results show a good linear relationship.

To further assess the scalability of DistGER, we generate synthetic graphs [11] with a fixed node degree of 10 and the number of nodes from 10^5 to 10^9. Figure 6 presents the running times for random walks and training on these synthetic graphs using a cluster of 8 machines, suggesting that the running time increases linearly with the size of a graph, and DistGER has the capability to handle even billion-node graphs. Moreover, the running times for six real-world graphs (including the *UK graph* with $|E| = 3.7B$, $|V| = 100M$, for which the competing systems do not terminate in 1 day or crash due to hardware and memory limitation) are inserted into the plot, which is consistent with the trend on synthetic data.

6.4 Effectiveness w.r.t. Competitors

Link prediction. To perform link prediction on a given graph G, following [17, 18, 49, 64], we first uniformly at random remove 50% edges as positive test edges, and the rest are used as positive training edges. We also provide negative training and test edges by considering those node pairs between which no edge exists in G. We ensure that the positive and negative set sizes are similar. The link prediction is conducted as a classification task based on the similarity of u and v, i.e., $\varphi(u) \cdot \varphi(v)$. The effectiveness of link prediction is measured via the *AUC* (Area Under Curve) score [28] – the higher the better. We repeat this procedure 50 times to offset the randomness of edge removal and report the average *AUC* in Table 4. DistGER outperforms all competitors on these graphs, except for PBG on *Com-Orkut*, where DistGER ranks second. On average, DistGER has an 11.7% higher *AUC* score compared with the other three systems, thanks to our information-centric random walks. PBG is the best on *Com-Orkut* because this graph is much denser and is friendly to the multi-relationship-based model in PBG. Figure 7 exhibits accuracy-efficiency tradeoffs of DistGER and competitors, i.e., their *AUC* convergence curves w.r.t. increasing running times of random walks and training, over *LiveJournal*, further indicating that DistGER has better efficiency and effectiveness than the competitors.

Table 4: *AUC* **scores of** DistGER **and competitors for link prediction**

Method	Youtube	LiveJournal	Com-Orkut	Twitter
PBG	0.753	0.882	**0.955**	0.912
DistDGL	0.894	0.718	0.815	running time > 1 day
KnightKing	0.904	0.963	0.918	out-of-memory
DistGER	**0.966**	**0.976**	0.921	**0.919**

6.5 Efficiency due to Individual Parts of DistGER

Random walk and training efficiency. To evaluate the system design of DistGER (§3, §4), we first compare the efficiency of random walks and training with those of KnighKing and HuGE-D. For random walks (Figure 8(a)), DistGER significantly outperforms KnightKing and HuGE-D on all our graph datasets, achieving an average speedup of 3.32× and 3.88×, respectively. Although HuGE-D implements information-oriented random walks on KnightKing, due to additional computation and communication overheads during on-the-fly information measurements (§2.3), its efficiency can be lower than that of KnightKing. We also notice that the random walk lengths (L) and the number of random walks (r) reduce (on average) 63.2% and 18%, respectively, in our information-oriented random walks, compared to KnightKing's routine random walk configuration.

Another benefit of information-centric random walks is that it generates concise and effective corpus to improve training efficiency. Compared to KnightKing, DistGER achieves 17.37×-27.95× acceleration in training over all our graphs. Next, considering the same corpus size, we compare the training efficiency of Pword2vec and DSGL (trainer in DistGER). Figure 8(b) shows that DSGL achieves 4.31× average speedup compared to Pword2vec. We also notice that

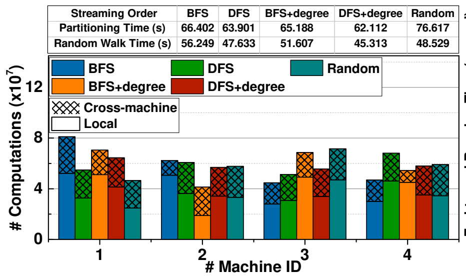
Table 5: Performance evaluation of partitioning for DistGER and Competitors

(a) Partitioning time for DistGER and competitors

graph	PBG	DistDGL (METIS)	DistGER (MPGP)
FL	383.28 s	127.72 s	**15.96 s**
YT	349.15 s	116.30 s	**13.56 s**
LJ	458.52 s	425.19 s	**36.42 s**
OR	2662.62 s	2761.25 s	**294.68 s**
TW	22 hour s	> 1 day	**9 hours**

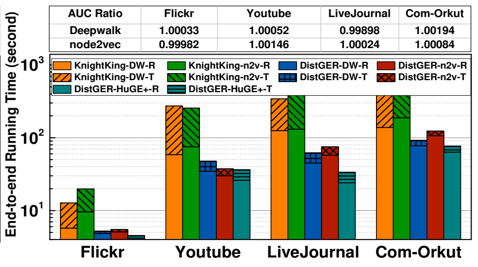
(b) Evaluation of Parallel MPGP

graph	Streaming	Partitioning	Walking
LJ	DFS+deg	21.86 s	**23.78 s**
	BFS+deg	**21.25 s**	24.79 s
OR	DFS+deg	**151.29 s**	77.12 s
	BFS+deg	156.37 s	**46.55 s**
TW	DFS+deg	**1940.65 s**	683.81 s
	BFS+deg	2034.21 s	**590.36 s**



Streaming Order	BFS	DFS	BFS+degree	DFS+degree	Random
Partitioning Time (s)	66.402	63.901	65.188	62.112	76.617
Random Walk Time (s)	56.249	47.633	51.607	45.313	48.529

Figure 9: The distribution of local computations and cross-machine communications for different streaming orders on *LiveJournal*. The top table reports their running times for partitioning and random walks



AUC Ratio	Flickr	Youtube	LiveJournal	Com-Orkut
Deepwalk	1.00033	1.00052	0.99898	1.00194
node2vec	0.99982	1.00146	1.00024	1.00084

Figure 10: Generality of DistGER vs. KnightKing. The bars show random walk efficiency ($-R$) and training efficiency ($-T$) for Deepwalk (DW), node2vec (n2v) and HuGE+. The top table shows the ratio $\frac{AUC \text{ for DistGER}}{AUC \text{ of KnightKing}}$, with DW and n2v, task: link prediction

the average throughput (number of nodes processed per second) for DSGL is up to 49.5 million/s, while that of Pword2vec is only up to 16.1 million/s. These results indicate that our distributed Skip-Gram learning model (§4) is more efficient than Pword2vec.

Partitioning efficiency. Considering the randomness inherent in random walks, the partitioning scheme is critical to overall efficiency. For DistGER, Figure 8(c) exhibits that our multi-proximity-aware streaming graph partitioning (MPGP) significantly reduces (avg. reduction 45%) the number of cross-machine messages than the workload-balancing partition of KnightKing on five graphs. Moreover, it improves the efficiency by 38.9% for the random walking procedure (Figure 8(d)) over the same set of walks. We report in Table 5(a) the time required for graph partitioning in competing systems, where DistDGL uses the METIS algorithm [22] for partitioning. The results show that MPGP performs partitioning with very little overhead in most cases, and the partitioning efficiency is on average 25.1× faster than competitors. In Figure 9, we exhibit the distribution of local computations and cross-machine communications on four machines for different streaming orders, and the top table reports their running times for partitioning and random walks. For sequential MPGP, we find that the DFS+degree-based streaming order (§3.2) is more efficient than other streaming orders, and it also strikes the best balance between cross-machine communications reduction and workload balancing. Table 5(b) exhibits the performance evaluation of parallel MPGP on the small- (*LiveJournal*), medium- (*Com-Orkut*) and large-scale (*Twitter*) graphs. The results show that DFS+Degree in parallel MPGP is still the best or comparable in terms of partition time, due to the same reason as stated in our third optimization scheme (§3.2). On the other hand, BFS+Degree in parallel MPGP works the best in terms of random walk time due to preserving the locality of the graph structure (our fourth optimization scheme in §3.2). We ultimately recommend BFS+Degree for parallel MPGP, since it reduces the partition time greatly, while the random walk time is comparable to that obtained from sequential MPGP.

6.6 Generality of DistGER

To demonstrate the generality of DistGER, we deploy Deepwalk [39], node2vec [18] and HuGE+ [16] on DistGER. While the original Deepwalk and node2vec follow traditional random walks, in DistGER the walk length and the number of walks are decided via information-centric measurements. Next, we also deploy both Deepwalk and node2vec on KnightKing which supports the routine configuration random walk. Figure 10 illustrates that DistGER reduces the random walks time by 41.1% and 51.6% on average for Deepwalk and node2vec, respectively. For training, DistGER is on average 17.7× and 21.3× faster than KnightKing+Pword2vec for Deepwalk and node2vec, respectively. Moreover, we also show the *AUC* ratio of DistGER and KnightKing, considering Deepwalk and node2vec, for link prediction. Our results depict that DistGER has comparable (in most cases, higher) *AUC* scores, while it improves the efficiency significantly even for traditional random walk-based graph embedding methods. HuGE+ is an extension of HuGE, and it uses the same HuGE information-centric method to determine the walk length and the number of walks per node. Figure 10 exhibits the compatibility of HuGE+ on DistGER via its general API.

7 CONCLUSION

We proposed DistGER, a novel, general-purpose, distributed graph embedding framework with improved effectiveness, efficiency, and scalability. DistGER incrementally computes information-centric random walks and leverages multi-proximity-aware, streaming, parallel graph partitioning to achieve high local partition quality and excellent workload balancing. DistGER also designs distributed Skip-Gram learning, which provides efficient, end-to-end distributed support for node embedding learning. Our experimental results demonstrated that DistGER achieves much better efficiency and effectiveness than state-of-the-art distributed systems KnightKing, DistDGL, and Pytorch-BigGraph, and scales easily to billion-edge graphs, while it is also generic to support traditional random walk-based graph embeddings.

ACKNOWLEDGMENTS

This work was supported by the NSFC (No. U22A2027, 61832020, 61821003). Arijit Khan acknowledges support from the Novo Nordisk Foundation grant NNF22OC0072415. Siqiang Luo acknowledges support from Singapore MOE Tier 2 (T2EP20122-0003), Tier-1 seed (RS05/21) and NTU SUG-NAP. Peng Fang acknowledges support from CSC under grant 202106160066.

REFERENCES

[1] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov. 2018. Streaming Graph Partitioning: An Experimental Study. *Proc. VLDB Endow.* 11, 11 (2018), 1590–1603.

[2] L. Adamic, O. Buyukkokten, and E. Adar. 2003. A Social Network Caught in the Web. *First Monday* 8, 6 (2003).

[3] A.-L. Barabasi and R. Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512.

[4] M. Belkin and P. Niyogi. 2001. Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering. In *NeurIPS*.

[5] S. Bhagat, G. Cormode, and S. Muthukrishnan. 2011. Node Classification in Social Networks. In *Social Network Data Analytics*, Charu C. Aggarwal (Ed.). Springer, 115–148.

[6] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. 2002. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2 (2002), 135–151.

[7] P. Boldi, M. Santini, and S. Vigna. 2008. A Large Time-Aware Graph. *SIGIR Forum* 42, 2 (2008), 33–38.

[8] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. 2016. Recent Advances in Graph Partitioning. In *Algorithm Engineering - Selected Results and Surveys*. Lecture Notes in Computer Science, Vol. 9220. 117–158.

[9] H. Cai, V. W. Zheng, and K. C.-C. Chang. 2018. A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications. *IEEE Trans. Knowl. Data Eng.* 30, 9 (2018), 1616–1637.

[10] J. Canny, H. Zhao, B. Jaros, C. Ye, and J. Mao. 2015. Machine Learning at the Limit. In *IEEE International Conference on Big Data*.

[11] D. Chakrabarti, Y. Zhan, and C. Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SDM*.

[12] E. D. Demaine, A. López-Ortiz, and J. I. Munro. 2000. Adaptive Set Intersections, Unions, and Differences. In *SODA*.

[13] S. C. Dong, X. P. Miao, P. K. Liu, X. Wang, B. Cui, and J. X. Li. 2022. HET-KG: Communication-Efficient Knowledge Graph Embedding Training via Hotness-Aware Cache. In *ICDE*. IEEE, 1754–1766.

[14] P. Fang, A. Khan, S. Luo, F. Wang, D. Feng, Z. Li, W. Yin, and Y. Cao. 2022. Our code and datasets. https://github.com/RocmFang/DistGER.

[15] Peng Fang, Arijit Khan, Siqiang Luo, Fang Wang, Dan Feng, Zhenli Li, Wei Yin, and Yuchao Cao. 2023. Distributed Graph Embedding with Information-Oriented Random Walks. arXiv:2303.15702 [cs.DC]

[16] P. Fang, F. Wang, Z. Shi, H. Jiang, D. Feng, X. Xu, and W. Yin. 2022. How to Realize Efficient and Scalable Graph Embeddings via an Entropy-driven Mechanism. *IEEE Transactions on Big Data* (2022).

[17] P. Fang, F. Wang, Z. Shi, H. Jiang, D. Feng, and L. Yang. 2021. HuGE: An Entropy-driven Approach to Efficient and Scalable Graph Embeddings. In *ICDE*.

[18] A. Grover and J. Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. In *KDD*.

[19] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. 2013. WTF: The Who to Follow Service at Twitter. In *WWW*.

[20] W. L. Hamilton, Z. Ying, and J. Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NeurIPS*.

[21] S. Ji, N.r Satish, S. Li, and P. K. Dubey. 2019. Parallelizing Word2vec in Shared and Distributed Memory. *IEEE Transactions on Parallel and Distributed Systems* 30, 9 (2019), 2090–2100.

[22] G. Karypis and V. Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[23] H. Kwak, C. Lee, H. Park, and S. Moon. 2010. What is Twitter, a Social Network or a News Media?. In *WWW*.

[24] H. Kwak, C. Lee, H. Park, and S. Moon. 2012. Defining and Evaluating Network Communities based on Ground-truth. In *ICDM*.

[25] Y. Lee, J. Chung, and M. Rhu. 2022. SmartSAGE: Training Large-Scale Graph Neural Networks Using in-Storage Processing Architectures. In *ISCA* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 932–945.

[26] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich. 2019. Pytorch-BigGraph: A Large Scale Graph Embedding System. In *MLSys*.

[27] Y. Li, Z. Wu, S. Lin, H. Xie, M. Lv, Y. Xu, and J. C. S. Lui. 2019. Walking with Perception: Efficient Random Walk Sampling via Common Neighbor Awareness. In *ICDE*.

[28] R. N. Lichtenwalter, J. T. Lussier, and N. V. Chawla. 2010. New Perspectives and Methods in Link Prediction. In *KDD*. 243–252.

[29] C. Liu, H. K. Liu, H. Jin, X. F. Liao, Y. Zhang, Z. H. Duan, J. H. Xu, and H. Z. Li. 2022. ReGNN: a ReRAM-based Heterogeneous Architecture for General Graph Neural Networks. In *DAC*. 469–474.

[30] S. Q. Luo, Z. C. Zhu, X. K. Xiao, Y. Yang, C. B. Li, and B. Kao. 2023. Multi-Task Processing in Vertex-Centric Graph Systems: Evaluations and Insights. (2023).

[31] T. Mikolov, K. Chen, G. Corrado, and J. Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR*.

[32] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NeurIPS*.

[33] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman. 2021. Marius: Learning Massive Graph Embeddings on a Single Machine. In *OSDI*. 533–549.

[34] F. Nie, W. Zhu, and X. Li. 2017. Unsupervised Large Graph Embedding. In *AAAI*.

[35] F. Niu, B. Recht, C. Re, and S. J. Wright. 2011. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*. 693–701.

[36] A. Pacaci and M. T. Özsu. 2019. Experimental Analysis of Streaming Algorithms for Graph Partitioning. In *SIGMOD*.

[37] Y. Park, S. Min, and J. W. Lee. 2022. Ginex: SSD-Enabled Billion-Scale Graph Neural Network Training on a Single Machine via Provably Optimal in-Memory Caching. *Proc. VLDB Endow.* 15, 11 (2022), 2626–2639.

[38] R. A. Pearce, M. B. Gokhale, and N. M. Amato. 2010. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *SC*.

[39] B. Perozzi, R. Al-Rfou, and S. Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *KDD*.

[40] J. Qiu, Y. Dong, H. Ma, J. Li, C. Wang, K. Wang, and J. Tang. 2019. NetSMF: Large-Scale Network Embedding as Sparse Matrix Factorization. In *WWW*.

[41] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, and J. Tang. 2018. Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec. In *WSDM*.

[42] V. Rengasamy, T. Y. Fu, W. C. Lee, and K. Madduri. 2017. Optimizing Word2Vec Performance on Multicore Systems. In *IA3*.

[43] H. Robbins and S. Monro. 1951. A Stochastic Approximation Method. *Annals of Mathematical Statistics* 22, 3 (1951), 400–407.

[44] M. Serafini. 2021. Scalable Graph Neural Network Training: The Case for Sampling. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 68–76.

[45] C. Shi, B. Hu, W. X. Zhao, and P. S. Yu. 2019. Heterogeneous Information Network Embedding for Recommendation. *IEEE Trans. Knowl. Data Eng.* 31, 2 (2019), 357–370.

[46] I. Stanton and G. Kliot. 2012. Streaming Graph Partitioning for Large Distributed Graphs. In *KDD*.

[47] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. 2015. LINE: Large-scale Information Network Embedding. In *WWW*.

[48] L. Tang and H. Liu. 2009. Scalable Learning of Collective Behavior based on Sparse Social Dimensions. In *CIKM*.

[49] A. Tsitsulin, D. Mottin, P. Karras, and E. Müller. 2018. VERSE: Versatile Graph Embeddings from Similarity Measures. In *WWW*.

[50] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. 2014. FENNEL: Streaming Graph Partitioning for Massive Scale Graphs. In *WSDM*.

[51] K. Tu, P. Cui, X. Wang, P. S. Yu, and W. Zhu. 2018. Deep Recursive Network Embedding with Regular Equivalence. In *KDD*.

[52] L. G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (1990), 103–111.

[53] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. 2018. Graph Attention Networks. In *ICLR*.

[54] D. Wang, P. Cui, and W. Zhu. 2016. Structural Deep Network Embedding. In *KDD*.

[55] H. Wang, J. Wang, J. Wang, M. Zhao, W. Zhang, F. Zhang, X. Xie, and M. Guo. 2018. GraphGAN: Graph Representation Learning With Generative Adversarial Nets. In *AAAI*.

[56] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. In *SIGKDD*.

[57] 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. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[58] X. Wang, P. Cui, J. Wang, J. Pei, W. Zhu, and S. Yang. 2017. Community Preserving Network Embedding. In *AAAI*.

[59] W. Wei, Y. Wang, P. Gao, S. Sun, and D. Yu. 2020. A Distributed Multi-GPU System for Large-Scale Node Embedding at Tencent. *CoRR* abs/2005.13789 (2020).

[60] X. Wei, L. Xu, B. Cao, and P. S. Yu. 2017. Cross View Link Prediction by Learning Noise-resilient Representation Consensus. In *WWW*.

[61] Y. Wu, K. Ma, Z. Cai, T. Jin, B. Li, C. Zheng, J. Cheng, and F. Yu. 2021. Seastar: Vertex-centric Programming for Graph Neural Networks. In *EuroSys*. 359–375.

[62] K. Yang, X. Ma, S. Thirumuruganathan, K. Chen, and Y. Wu. 2021. Random Walks on Huge Graphs at Cache Efficiency. In *SOSP*.

[63] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang. 2019. KnightKing: A Fast Distributed Graph Random Walk Engine. In *SOSP*.

[64] R. Yang, J. Shi, X. Xiao, Y. Yang, and S. S. Bhowmick. 2020. Homogeneous Network Embedding for Massive Graphs via Reweighted Personalized PageRank. *Proc. VLDB Endow.* 13, 5 (2020), 670–683.

[65] R. Zafarani and H. Liu. 2009. Social Computing Data Repository at ASU. In *http://socialcomputing.asu.edu*.

[66] J. Zhang, Y. Dong, Y. Wang, J. Tang, and M. Ding. 2019. ProNE: Fast and Scalable Network Representation Learning. In *IJCAI*.

[67] C. Zheng, H. Chen, Y. Cheng, Z. Song, Y. Wu, C. Li, J. Cheng, H. Yang, and S. Zhang. 2022. ByteGNN: Efficient Graph Neural Network Training at Large Scale. *Proc. VLDB Endow.* 15, 6 (2022), 1228–1242.

[68] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *IA3@SC*.

[69] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* (2019), 2094–2105.

[70] Z. Zhu, S. Xu, J. Tang, and M. Qu. 2019. GraphVite: A High-Performance CPU-GPU Hybrid System for Node Embedding. In *WWW*.