SCARA: Scalable Graph Neural Networks with Feature-Oriented Optimization

Ningyi Liao*
Nanyang Technological University
liao0090@e.ntu.edu.sg

Dingheng Mo*
Nanyang Technological University
dingheng001@e.ntu.edu.sg

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

Xiang Li
East China Normal University
xiangli@dase.ecnu.edu.cn

Pengcheng Yin
Google Research
pcyin@google.com

ABSTRACT

Recent advances in data processing have stimulated the demand for learning graphs of very large scales. Graph Neural Networks (GNNs), being an emerging and powerful approach in solving graph learning tasks, are known to be difficult to scale up. Most scalable models apply node-based techniques in simplifying the expensive graph message-passing propagation procedure of GNN. However, we find such acceleration insufficient when applied to million- or even billion-scale graphs. In this work, we propose SCARA, a scalable GNN with feature-oriented optimization for graph computation. SCARA efficiently computes graph embedding from node features, and further selects and reuses feature computation results to reduce overhead. Theoretical analysis indicates that our model achieves sub-linear time complexity with a guaranteed precision in propagation process as well as GNN training and inference. We conduct extensive experiments on various datasets to evaluate the efficacy and efficiency of SCARA. Performance comparison with baselines shows that SCARA can reach up to 100× graph propagation acceleration than current state-of-the-art methods with fast convergence and comparable accuracy. Most notably, it is efficient to process precomputation on the largest available billion-scale GNN dataset Papers100M (111M nodes, 1.6B edges) in 100 seconds.

PVLDB Reference Format:
Ningyi Liao, Dingheng Mo, Siqiang Luo, Xiang Li, and Pengcheng Yin. SCARA: Scalable Graph Neural Networks with Feature-Oriented Optimization. PVLDB, 15(11): 3240-3248, 2022.
doi:10.14778/3551793.3551866

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gdmnl/SCARA-PPR.

1 INTRODUCTION

Recent years have witnessed the burgeoning of online services based on data represented by graphs, which leads to rapid increase in the amount and complexity of such graph data. Graph Neural Networks (GNNs) are specialized neural models designed to represent and process graph data, and have achieved strong performance on graph understanding tasks such as node classification [7, 11, 16, 19], link prediction [5, 27, 36, 39], and community detection [2, 10, 28].

One of the most widely adopted GNN models is the Graph Convolutional Network (GCN) [19] which learns graph representations by leveraging information of topological structure. Specifically, the GCN represents each node state by a feature vector, successively propagates the state to neighboring nodes, and updates the neighbor features using a neural network. This interleaved process of graph propagation and state update can proceed for multiple iterations.

While being able to effectively gather state information from the graph structure, GCNs are known to be resource-demanding, which implies limited scalability when deployed to large-scale graphs [34, 41]. It is also non-trivial to fit the node features of large graphs into the memory of hardware accelerators like GPUs. However, it is increasingly demanding to apply these effective models to modern real-world graph datasets. Recent studies have attempted to learn representations of large graphs such as the Microsoft Academic Graph (MAG) of 100 million entries [24, 35]. Nonetheless, directly fitting the basic GCN model to such data would easily cause unacceptable training time or out-of-memory error. Hence, how to adopt the GCN model efficiently to these very large-scale graphs while benefiting from its performance becomes a challenging yet important problem in realistic applications.

Existing Approaches are Not Scalable Enough. Several techniques have been proposed towards more efficient learning for GNN, addressing the scalability issues. One optimization is to decouple graph propagation from the feature update and employ linear models to simplify computation [20, 32]. There is no need to store the whole graph in the GPU and the memory footprint is thence reduced. Such methods exploit graph data management techniques such as Personalized PageRank [23] to calculate the graph representation used in the model. Another direction is easing node interdependence, which enables training on smaller batches and is achieved by neighbor sampling [8, 15], layer sampling [7, 13, 16], and subgraph sampling [11, 18, 37]. Various sampling schemes have been applied to restrain the number of nodes contained in GNN learning pipelines and reduce computational overhead. Other algorithms are also utilized in simplifying graph propagation and learning in order to improve efficiency and efficacy, including diffusion [4, 20], self-attention [26, 27, 38], and quantization [12].

Unfortunately, such methods are still not efficient enough when applied to million-scale or even larger graphs. According to [29], the very recent state-of-the-art algorithm GBP [9] still consumes more

*Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551866

Table 1: Time and memory complexity of scalable GNN models. Precomputation memory complexity indicates the usage of intermediate variables, while the training and inference memory refer to the GPU usage for storing and updating representation and weight matrices in each training iteration. Time complexity is for the full training and/or inference node set.

Model	Precomp. Mem.	Training Mem.	Inference Mem.	Precomp. Time	Training Time	Inference Time
GCN [19]	–	$O(LnF + LF^2)$	$O(LnF + LF^2)$	–	$O(ILmF + ILnF^2)$	$O(LmF + LnF^2)$
GraphSAINT [37]	–	$O(L^2bF + LF^2)$	$O(LnF + LF^2)$	–	$O(IL^2nF^2)$	$O(LmF + LnF^2)$
GAS [13]	$O(LnF)$	$O(LdbF + LF^2)$	$O(LdbF + LF^2)$	$O(m + LnF)$	$O(ILmF + ILnF^2)$	$O(nF)$
APPNP [20]	$O(m)$	$O(LbF + LF^2 + db)$	$O(LbF + LF^2 + db)$	$O(m)$	$O(ITmF + ILnF^2)$	$O(TmF + LnF^2)$
PPRGo [6]	$O(n/r_{max})$	$O(LbF + LF^2 + Kb)$	$O(LbF + LF^2 + Kb)$	$O(m/r_{max})$	$O(IKnF + ILnF^2)$	$O(KnF + LnF^2)$
SGC [32]	$O(m)$	$O(LbF + LF^2)$	$O(LbF + LF^2)$	$O(LmF)$	$O(ILnF^2)$	$O(LnF^2)$
GBP [9]	$O(nF)$	$O(LbF + LF^2)$	$O(LbF + LF^2)$	$O(LF\sqrt{Lm\log(Ln)}/\epsilon)$	$O(ILnF^2)$	$O(LnF^2)$
SCARA (ours)	$O(nF)$	$O(LbF + LF^2)$	$O(LbF + LF^2)$	$O(F\sqrt{m\log n}/\lambda)$	$O(ILnF^2)$	$O(LnF^2)$

than 10^4 seconds solely for precomputation on the Papers100M graph (111M nodes, 1.6B edges, generated from MAG) to reach proper accuracy. In our experiments, the same model even exceeds the 160GB RAM bound on a single worker during learning. Such cost is still too high for the method to be applied in practice.

Our Contributions. In this paper, we propose SCARA, a scalable Graph Neural Network algorithm with low time complexity and high scalability on very large datasets. On the theoretical side, the time complexity of SCARA for precomputation/training/inference matches the same sub-linear level with the state of the art, as shown in Table 1. On the practical side, to our knowledge, SCARA is the first GNN algorithm that can be applied to billion-scale graph Papers100M with a precomputation time less than 100 seconds and complete training under a relatively strict memory limit.

Particularly, SCARA employs several feature-oriented optimizations. First, we observe that most current scalable methods repetitively compute the graph propagation information from the node-based dimension, which results in complexity at least proportional to the number of graph nodes. To address this issue, we design a Feature-Push method that realizes the information propagation from the feature vectors, which removes the linear dependency on the number of nodes in the complexity while maintaining the same precision of corresponding graph propagation values. Second, as we mainly process the feature vectors, we discover that there is significant room to reuse the computation results across different feature dimensions. Hence we propose the Feature-Reuse algorithm. Through compositing the calculation results, SCARA can efficiently prevent time-consuming repetitive propagation. By such means, SCARA outperforms all leading competitors in our experiments in all 6 GNN learning tasks in regard to model convergence time, i.e., the sum of precomputation and training time, with highly efficient inference speed, significantly better memory overhead, and comparable or better accuracy.

In summary, we have made the following contributions:

- We present the Feature-Push algorithm which propagates the graph information from the feature vectors with forward push and random walk. Our method realizes a sub-linear complexity for precomputation running time along with efficient model training and inference implemented in the mini-batch approach.
- We propose the Feature-Reuse mechanism which further utilizes the feature-oriented optimizations to improve the efficiency of feature propagation while maintaining precision. The technique is able to half the overhead for several graph representations.
- We conduct comprehensive experiments to evaluate the efficiency and effectiveness of the SCARA model on various datasets and with benchmark methods. Our model is efficient to process the billion-scale dataset Papers100M. It also achieves up to 100× faster in precomputation time than the current state of the art.

2 PRELIMINARIES AND RELATED WORKS

Notations. Consider a graph $G = \langle V, E\rangle$ with node set V and edge set E. Let $n = |V|$, $m = |E|$, and $d = m/n$. The graph connectivity is represented by the adjacency matrix with self-loops as $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, while the diagonal degree matrix is $\boldsymbol{D} \in \mathbb{R}^{n\times n}$. Following [9, 29], we normalize the adjacency matrix by $\boldsymbol{D}$ with convolution coefficient $r \in [0, 1]$ as $\tilde{\boldsymbol{A}}_{(r)} = \boldsymbol{D}^{r-1}\boldsymbol{A}\boldsymbol{D}^{-r}$. For each node $v \in V$, denote the set of the out-neighbors by $\mathcal{N}(v) = \{u|(v, u) \in E\}$, and the out-degree of v by $d(v) = |\mathcal{N}(v)|$. Each v has an F-dimension attribute vector $\boldsymbol{x}(v)$, which composes the attribute matrix $\boldsymbol{X} \in \mathbb{R}^{n\times F}$.

A GNN recurrently computes the node representation matrix $\boldsymbol{H}^{(l)}$ as current state in the l-th layer. The model input feature matrix is $\boldsymbol{H}^{(0)} = \boldsymbol{X}$ in particular. For a conventional L-layer GCN [19], the $(l + 1)$-th representation matrix $\boldsymbol{H}^{(l+1)}$ is updated as:

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\tilde{\boldsymbol{A}}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = 0, 1, \cdots, L-1, \tag{1}$$

where $\boldsymbol{W}^{(l)}$ is the trainable weight matrix of the l-th layer, $\tilde{\boldsymbol{A}} = \tilde{\boldsymbol{A}}_{(1/2)}$ is the normalized adjacency matrix, and $\sigma(\cdot)$ is the activation function such as ReLU or softmax. For analysis simplicity we keep the feature size F unchanged in all layers.

Summarized in Table 1, we present an analysis on the complexity bounds of GCN in Eq. (1) to explain the restraints of its efficiency. We focus on the training phase as it updates the model for I epochs and requires most resources. For the L-layer GCN model, the multiplication of *graph propagation* $\tilde{\boldsymbol{A}}\boldsymbol{H}^{(l)}$ is bounded by a complexity of $O(LmF)$ giving the adjacency matrix $\tilde{\boldsymbol{A}}$ with m entries and the propagation is conducted for L iterations. The overhead for *feature transformation* by multiplying $\boldsymbol{W}^{(l)}$ is $O(LnF^2)$. In the *training* stage, the above procedure is repeated to iteratively update the model weights $\boldsymbol{W}^{(l)}$. As discovered by previous studies [9, 11], the dominating term is $O(LmF)$ when the graph is large, since the latter transformation can be accelerated by parallel computation. Hence, the full graph propagation becomes the scalability bottleneck. For memory usage, the GCN typically takes $O(LnF + LF^2)$ space to store layer-wise node representation and weight matrices.

Post-Propagation Model. As the graph propagation possesses the major computation overhead when the graph is scaled-up, a

straightforward idea is to simplify this step and prevent it from being repetitively included in each layer. Such approaches are regarded as propagation decoupling models [21, 40]. We further classify them into post- and pre-propagation variants based on the presence stage of propagation relative to feature transformation.

The post-propagation decoupling methods apply propagation only on the last model layer, enabling efficient and individual computation of the graph propagation matrix, as well as the fast and simple model training. The APPNP model [20] introduces the personalized PageRank (PPR) [23] algorithm in the propagation stage. The iterative graph propagation in the GCN updates is replaced by multiplying the PPR matrix after the feature transformation layers:

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = 0, 1, \cdots, L-2, \tag{2}$$

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\hat{\boldsymbol{\Pi}}\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = L-1, \tag{3}$$

where $\hat{\boldsymbol{\Pi}} = \sum_{l=0}^{L} \alpha(1-\alpha)^l \tilde{\boldsymbol{A}}^l$ is the PPR matrix.

In this design, the feature transformation benefit from the mini-batch scheme in both training and inference stages, hence reducing the demand for GPU memory. In Table 1, the batch size is b. Regarding computation speed, a T-round Power Iteration computation on the PPR matrix [23] leads to $O(TmF + LnF^2)$ time per epoch. The PPRGo model [6] further improves the efficiency of precomputing the PPR matrix $\boldsymbol{\Pi}$ by the Forward Push algorithm [3] with an error threshold r_{max} and only records the top-K entries. However, it demands $O(n/r_{max})$ space to store the dense PPR matrix.

Pre-Propagation Model. Another line of research, namely the pre-propagation models such as SGC [32], chooses to propagate graph information in advance and encode it to the attributes matrix $\boldsymbol{X}$, forming an embedding matrix $\boldsymbol{P}$ that is utilized as the input feature to the neural network layers. In a nutshell, we summarize the model updates in the following scheme:

$$\boldsymbol{H}^{(0)} = \boldsymbol{P} = \sum_{l=0}^{L_P} a_l \tilde{\boldsymbol{A}}^l_{(r)} \cdot \boldsymbol{X}, \tag{4}$$

$$\boldsymbol{H}^{(l+1)} = \sigma\left(\boldsymbol{H}^{(l)}\boldsymbol{W}^{(l)}\right), \quad l = 0, 1, \cdots, L-1, \tag{5}$$

where L_P denotes the depth of precomputed propagation and a_l is the layer-dependent diffusion weight.

The line of Eq. (4) corresponds to the *precomputation section* and is calculated only once for each graph. The complexity of this stage is solely related to the precomputation techniques applied in the model. In SGC, the equation is given by an L-hop multiplication of $\boldsymbol{P} = \tilde{\boldsymbol{A}}^L\boldsymbol{X}$, taking $O(LmF)$ time. A recent work GBP [9] employs a PPR-based bidirectional propagation with $L_P = L$ and tunable a_l and r. Under an approximation of relative error ϵ, it improves precomputation complexity to $O(LF\sqrt{Lm\log(Ln)}/\epsilon)$ in the best case. It is notable that since GBP contains a node-based traverse scheme, it is sensitive to the scale of n in practice.

Eq. (5) follows the neural network *feature transformation*, taking $\boldsymbol{P}$ as input feature. Compared to Eq. (3), it completely removes the need for additional multiplication, hence both training and inference are reduced to $O(LnF^2)$. The simple GNN provides scalability in both resource-demanding training and frequently-queried inference, with the ease to employ techniques such as mini-batch training, parallel computation, and data augmentation.

Other Methods. There is a large scope of GNNs related to sampling techniques, which simplifies the propagation by replacing the full-batch graph updates with sampled nodes in mini-batches. A popular direction is graph-wise sampling, such as Cluster-GCN [11] exploiting clustering structure and GraphSAINT [37] using various levels of information. The representative GraphSAINT-RW integrates L-hop random walk graph sampling with a training complexity $O(L^2nF+LnF^2)$. It is however not applicable in the full graph inference stage, causing the inference time and memory overheads to be identical to the vanilla GCN. GAS [13] samples layer-wise neighbors and consumes great memory for historical embedding. It has $O(LmF + LnF^2)$ training overhead, while the optimal inference complexity is benefited by the cached embedding.

In our experiments, we compare our GNN algorithm with the state of the arts from each of the aforementioned categories, to demonstrate the scalability and effectiveness of our algorithm.

3 SCARA FRAMEWORK

We propose our SCARA framework composing Feature-Push and Feature-Reuse. The Feature-Push algorithm conducts propagation from the aspect of feature, while Feature-Reuse is a novel technique that reuses columns in the feature matrix. We present analysis on the algorithmic complexity and precision guarantee to demonstrate the theoretical validity and effectiveness of SCARA.

3.1 Overview

To realize scalability in the network training and inference stage, and to better employ advanced Personalized PageRank (PPR) algorithms to optimize graph diffusion, we apply the backbone of propagation decoupling approach in our GNN design. Similar to the idea of pre-propagation models [9, 32], in precomputation stage we follow Eq. (4) to compute the graph information $\boldsymbol{P}$ in advance together with the node attributes $\boldsymbol{X}$. Then, a simple yet effective feature transformation is conducted as given in Eq. (5).

The propagation stage is the complexity bottleneck, mentioned earlier. Hence, we focus on reducing its computation complexity. We rewrite Eq. (4) in our propagation as:

$$\boldsymbol{P} = \sum_{l=0}^{\infty} \alpha(1-\alpha)^l \tilde{\boldsymbol{A}}^l_{(r)} \cdot \boldsymbol{X} = \sum_{l=0}^{\infty} \alpha(1-\alpha)^l \left(\boldsymbol{D}^{r-1}\boldsymbol{A}\boldsymbol{D}^{-r}\right)^l \boldsymbol{X}, \tag{6}$$

where α is the teleport probability as we set $a_l = \alpha(1-\alpha)^l$ to be associated with the form in the PPR calculation. Compared with APPNP and PPRGo, we adopt a generalized graph adjacency $\tilde{\boldsymbol{A}}_{(r)}$ with an adjustable convolution factor $r \in [0, 1]$ to fit different scales of graphs. The upper bound is set to $L_P = \infty$ to better capture the whole graph information without efficiency degeneration.

Our computation of Eq. (6) is displayed in Algorithm 1 (Feature-Push) and explained in detail in Section 3.2. The highlight of Feature-Push is the application of propagating from features, which differs from prior works. In many real-world tasks, when a graph is scaled-up, its numbers of nodes (n) and edges (m) increase, but the node attributes dimension (F) usually remains unchanged. Thus, an algorithm with complexity mainly dependent on F enjoys better scalability than those dominated by n or m.

As the attribute matrix $\boldsymbol{X}$ is included in our computation, we then investigate how to fully utilize its information contained to further accelerate our algorithm, which leads to the Algorithm 2 (Feature-Reuse). The motivation is to reduce the expensive iterative computation of $\boldsymbol{P}$ components by exploiting the previous

results based on attribute vectors $\boldsymbol{x}$ on selected dimensions f. We apply a linear combination scheme with precision guarantee to lighten the constraints of Algorithm 1 while improving speed. We further describe this methodology in Section 3.3.

3.2 Feature-Push

Examining Eq. (6), the embedding matrix $\boldsymbol{P}$ is the composition of graph diffusion matrix $\tilde{\boldsymbol{A}}_{(r)}$ and node attributes $\boldsymbol{X}$. Most scalable methods such as APPNP [20] and SGC [32] compute the propagation part separately from network training, resulting in a complexity at least proportional to m. GBP [9] discusses a bidirectional propagation with both node-side random walk on $\boldsymbol{D}^{-1}\boldsymbol{A}$ and feature-side reverse push on $\boldsymbol{D}^{-r}\boldsymbol{X}$. Although the random walk step ensures precision guarantee, it requires long running time when not being accelerated by other methods [30, 31].

We propose the Feature-Push approach that propagates graph information from the feature dimension, which is capable to utilize efficient single-source PPR algorithms through a simple but surprisingly effective transformation. Intuitively, the Feature-Push is first initialized by the normalized features $\boldsymbol{D}^{1-r}\boldsymbol{X}$. Then, single-source PPR algorithms, which compute the PPR values from a source node to other nodes, are applied to iteratively propagate the information with $\left(\boldsymbol{A}\boldsymbol{D}^{-1}\right)^l$. It achieves the embedding matrix based on that:

$$\tilde{\boldsymbol{A}}^l_{(r)} \cdot \boldsymbol{X} = \left(\boldsymbol{D}^{r-1}\boldsymbol{A}\boldsymbol{D}^{-r}\right)^l \boldsymbol{X} = \boldsymbol{D}^{r-1}\left(\boldsymbol{A}\boldsymbol{D}^{-1}\right)^l \boldsymbol{D}^{1-r}\boldsymbol{X}, \quad (7)$$

In order to better derive Feature-Push, we borrow the Personalized PageRank (PPR) notations to describe our technique manipulating feature vectors. On a graph G, given a source node $s \in V$ and a target node $t \in V$, the PPR $\pi(s,t)$ represents the probability of a random walk with teleport factor $\alpha \in (0,1)$ which starts at node s and stops at t. In general, *forward* PPR algorithms, often categorized as *single-source* PPR, start the computation from s, contrasted to *backward* or *reverse* alternatives that are developed from t [29].

Algorithm 1 Feature-Push

Input: Graph G, node set U, feature vector $\boldsymbol{x}$, probability α, convolution factor r, push coefficient β
Output: Approximate embedding vector $\hat{\boldsymbol{\pi}}(\boldsymbol{x})$

1 **for all** $u \in U$ **do**
2 $r'(\boldsymbol{x};u) \leftarrow x(u) \cdot d(u)^{1-r}$
3 $r(\boldsymbol{x};u) \leftarrow r'(\boldsymbol{x};u)/\sum_{u\in U} r'(\boldsymbol{x};u)$
4 $\hat{\pi}(\boldsymbol{x};t) \leftarrow 0$ for all $t \in U$
5 **while** exist $u \in U$ such that $r(\boldsymbol{x};u) > r_{max}/d(u)$ **do**
6 **for all** $v \in \mathcal{N}(u)$ **do**
7 $r(\boldsymbol{x};v) \leftarrow r(\boldsymbol{x};v) + (1-\alpha)\cdot r(\boldsymbol{x};u)/d(u)$
8 $\hat{\pi}(\boldsymbol{x};u) \leftarrow \hat{\pi}(\boldsymbol{x};u) + \alpha \cdot r(\boldsymbol{x};u)$
9 $r(\boldsymbol{x};u) \leftarrow 0$
10 $r_{sum} \leftarrow \sum_{u\in U} r(\boldsymbol{x};u), \quad N_W \leftarrow r_{sum}/\beta$
11 **for all** $u \in U$ such that $r(\boldsymbol{x};u) \neq 0$ **do**
12 perform $\frac{r(\boldsymbol{x};u)}{r_{sum}} \cdot N_W$ random walks from u
13 **for all** random walk stopping at t **do**
14 $\hat{\pi}(\boldsymbol{x};t) \leftarrow \hat{\pi}(\boldsymbol{x};t) + r_{sum}/N_W$
15 $\hat{\pi}(\boldsymbol{x};t) \leftarrow \hat{\pi}(\boldsymbol{x};t)\cdot d(t)^{r-1}$ for all $t \in U$
16 **return** $\hat{\boldsymbol{\pi}}(\boldsymbol{x}) \leftarrow \left(\hat{\pi}(\boldsymbol{x};t_1), \cdots, \hat{\pi}(\boldsymbol{x};s_{n_U})\right)$

When the PPR calculation is integrated with features, it shares similarities in forms but with different interpretation. Consider the PPR problem with nodes in a set $U \subseteq V$ as the source nodes. Let n_U be the size of set U. Denote a matrix $\boldsymbol{X} = (\boldsymbol{x}_1, \cdots, \boldsymbol{x}_F)$, where $\boldsymbol{x}_f$ $(1 \le f \le F)$ is the f-th column vector that is of length n_U and the sum of elements is 1. Following [29], we assume all the entries $x_f(u) \ge 0$ for each $u \in U$. We use $\pi(\boldsymbol{x};t)$ to represent the PPR for feature vector $\boldsymbol{x}$, and can be defined as the probability of the event that a random walk which starts at a node $s \in U$ with probability distribution $\boldsymbol{x}$ and stops at t. It can be derived from the definition that, each feature PPR $\pi(\boldsymbol{x};t)$ can be interpreted as a generalized integration of normal PPR value $\pi(s,t)$, hence the properties and operations of common PPR are still valid. The embedding matrix is $\boldsymbol{P} = (\boldsymbol{\pi}(\boldsymbol{x}_1), \cdots, \boldsymbol{\pi}(\boldsymbol{x}_F))$, where $\boldsymbol{\pi}(\boldsymbol{x}_f) = \boldsymbol{\pi}_f$ is the f-th column of feature PPR vector corresponding to vector $\boldsymbol{x}_f$, and is composed by $\boldsymbol{\pi}_f = \left(\pi(\boldsymbol{x}_f;t_1), \cdots, \pi(\boldsymbol{x}_f;t_{n_U})\right)$. We here look into the redefined problem for approximating feature PPR:

*Definition 3.1 (**Approximate PPR for Feature**).* Given an absolute error bound $\lambda > 0$, a PPR threshold $0 < \delta < 1$, and a failure probability $0 < \phi < 1$, the approximate PPR query for feature vector $\boldsymbol{x}$ computes an estimation $\hat{\pi}(\boldsymbol{x};t)$ for each $t \in U$ with $\pi(\boldsymbol{x};t) > \delta$, such that with probability at least $1-\phi$,

$$|\pi(\boldsymbol{x};t) - \hat{\pi}(\boldsymbol{x};t)| \le \lambda. \quad (8)$$

Recognizing that GNNs require less precise propagation information to achieve proper performance [25, 42], the approximate feature PPR enables employing efficient computation based on forward PPR algorithms without loss in eventual model effectiveness [31, 33]. We employ a scalable algorithm Feature-Push to compute the embedding matrix combining Forward Push [3] and Random Walk techniques that both operate feature vectors. The algorithm makes use of both approaches, that random walk is accurate but less efficient, while forward push is fast with a loose precision guarantee. Algorithms exploiting such combination have been the state of the arts in various PPR benchmarks [22, 31]. We highlight that the differences between Algorithm 1 and [22, 31] are two-fold. First, the push starts from the feature vector, which can be seen as a generalized PPR operation taking probability distribution $\boldsymbol{x}$ into account. Second, the feature-based query facilitates subsequent transformation in Eq. (7) and reusing in Eq. (9).

As shown in Algorithm 1, the Feature-Push algorithm outputs the approximation of embedding vector $\hat{\boldsymbol{\pi}}(\boldsymbol{x})$ for input feature $\boldsymbol{x}$. Repeating it for F times with all features $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_F$ produces all columns compositing the estimate of embedding matrix $\hat{\boldsymbol{P}}$. The algorithm first computes the approximation $\hat{\pi}(\boldsymbol{x};t)$ for each node $t \in U$ through forward push (line 2-9 in Algorithm 1), then conducts compressed random walks to save computation (line 10-14). We analyze each method and their combination respectively.

Forward Push on Feature Value. Instead of calculating the PPR value $\pi(s,t)$, the forward push method in Feature-Push maintains a *reserve value* $\hat{\pi}(\boldsymbol{x};t)$ directly for node $t \in U$ and feature $\boldsymbol{x}$ as the estimation of $\pi(\boldsymbol{x};t)$. An auxiliary *residue value* $r(\boldsymbol{x};t)$ is recorded as the intermediate result for each node-feature pair. The residue is initialized by the L_1-normalized feature vector $\boldsymbol{x}$. to transfer node attributes to distributions in line with $\pi(\boldsymbol{x};t)$ that stands for the probability with a sum of 1 for all nodes $t \in U$. The forward push

algorithm subsequently updates the residue of target node t from the source node s to propagate the information. The threshold r_{max} controls the terminating condition so that the process can stop early. Eventually, the forward push transfers α portion of node residue $r(\boldsymbol{x};t)$ into reserve value, while distributing the remaining $(1-\alpha)$ to the neighbors of s.

Random Walk on Feature Residue. Feature-Push then performs random walks with decay factor α to propagate the residue feature value. Compared with the pure random walk approach, Feature-Push only requires $\frac{r(\boldsymbol{x};t)}{r_{sum}} \cdot N_W$ number of walks per node with the same precision guarantee, benefiting from the Forward Push results. The estimation of $\hat{\pi}(\boldsymbol{x};t)$ is achieved by implementing the Monte-Carlo method [14, 30], and is updated according to the fraction of random walks terminating at t.

Combination and Normalization. To depict the combination of forward push and random walk, we define the coefficient β:

*Definition 3.2 (**Push Coefficient**).* The push coefficient β is the scale between the total left residual r_{sum} and the total number of sampled random walks N_W in Feature-Push.

The scale β is the key coefficient of Feature-Push, which balances absolute error guarantee and time complexity. Referencing the trade-off in [31], we set β to a specific value, namely standard push coefficient $\beta_s = \frac{\lambda^2}{(2\lambda/3+2)\cdot\log(2/\phi)}$, to satisfy the guarantee of $\hat{\pi}(\boldsymbol{x};t)$ in Definition 3.1. In Algorithm 1, the forward push and random walk are combined as line 14. Derived from the single-source PPR analysis [3, 31], we state that our Feature-Push algorithm provides an unbiased estimation $\hat{\pi}(\boldsymbol{x};t)$ of the value $\pi(\boldsymbol{x};t)$:

Lemma 3.3. *Algorithm 1 produces an unbiased estimation $\hat{\pi}(\boldsymbol{x};t)$ of the value $\pi(\boldsymbol{x};t)$ satisfying Eq. (8). Repeating it for F times produces an unbiased estimation $\hat{\boldsymbol{P}}$ of the embedding matrix $\boldsymbol{P}$.*

The combination of forward push and random walk generates the approximate $\boldsymbol{\Pi}^{(l)} = \alpha(1-\alpha)^l \left(\boldsymbol{A}\boldsymbol{D}^{-1}\right)^l$ for a certain l. To be aligned with the embedding matrix $\boldsymbol{P}^{(l)}$ in Eq. (6), we apply the normalization by degree vector (lines 2 and 15 in Algorithm 1) to achieve the transformation in Eq. (7). These operations on embedding values can be efficiently implemented in vector-based schemes.

3.3 Feature-Reuse

A key difference between the feature PPR and the classic single-source PPR is that, in single-source PPR, queries on nodes are orthogonal to each other, while in feature PPR there is similarity between different features. Calculating based on features in Algorithm 1 enables taking advantage of such property and utilizing computed values to estimate the PPR of another similar feature.

We propose Feature-Reuse algorithm that speeds up the feature PPR computation by leveraging and reusing the similarity between different feature vectors. We select a set of vectors as the base vectors from all features and compute their PPR values by Feature-Push. When querying the PPR value on a non-base feature vector, Feature-Reuse separates a segment of the vector that can be obtained by combining the base vectors, and estimate the PPR value of this segment directly with the PPR value of the base vectors without additional Feature-Push computation overhead.

As a toy example, if we have the PPR $\boldsymbol{\pi}(\boldsymbol{b})$ for base feature vector $\boldsymbol{b} = (0.5, 0.5)$, and need to compute the PPR for $\boldsymbol{x} = (0.4, 0.6)$, we can firstly decompose $\boldsymbol{x} = (0.4, 0.4) + (0, 0.2)$. We then acquire the PPR for $(0.4, 0.4)$ directly by $0.8\boldsymbol{\pi}(\boldsymbol{b})$, and just need to compute the PPR value of the residue $(0, 0.2)$. The latter PPR calculation is faster due to the reduced dimension and a loose precision bound.

Base Selection. Algorithm 2 shows the pseudo code of Feature-Reuse. To represent the similarity between feature vectors, we design a simple yet effective metric, namely the minimum L1 distance counter $M(\cdot)$. Feature-Reuse chooses $n_B \ll n_U$ feature vectors with the highest minimum L1 distance counter as the base vectors $\boldsymbol{b}_i$ to compose the base set $\boldsymbol{X}_B = \{\boldsymbol{b}_1, \cdots, \boldsymbol{b}_{n_B}\}$ (line 2-8). Feature-Push is then invoked to compute the PPR value $\hat{\boldsymbol{\pi}}(\boldsymbol{b}_i, \beta^*)$ of the base vectors with push coefficient $\beta^* = \gamma\beta_s = \frac{\gamma\lambda^2}{\log(2/\phi)\cdot(2\lambda/3+2)}$, where $0 < \gamma \leq 1$ is a tunable precision factor.

Residue Calculation. Algorithm 2 then computes the approximate values of the rest features (line 11-20). Given selected base vectors $\boldsymbol{b}_i \in \boldsymbol{X}_B$, a feature vector can be written in a linear decomposition with residue as $\boldsymbol{x}_f = \sum_{i=1}^{n_B} \theta_i \cdot \boldsymbol{b}_i + \boldsymbol{x}'$, where $0 \leq \theta_i < 1$ and $\boldsymbol{x}'$ is the residue feature vector. We compute the PPR vector $\hat{\boldsymbol{\pi}}(\boldsymbol{x}', \beta')$ according to the remaining part left in linear decomposition $\boldsymbol{x}'$ by Feature-Push with a less precise push coefficient $\beta' = \left(1 - \gamma\sum_{i=1}^{n_B}\theta_i\right)\beta_s$. Finally, we constitute the estimation as:

$$\boldsymbol{\pi}^*(\boldsymbol{x}_f) = \sum_{i=1}^{n_B} \theta_i \cdot \hat{\boldsymbol{\pi}}(\boldsymbol{b}_i, \beta^*) + \hat{\boldsymbol{\pi}}(\boldsymbol{x}', \beta'). \tag{9}$$

The PPR of base vectors $\hat{\boldsymbol{\pi}}(\boldsymbol{b}_i, \beta^*)$ acquired by Feature-Push has its own accuracy guarantee as stated in Lemma 3.3. However, how to assure the other vectors composed by Eq. (9) satisfy the

Algorithm 2 Feature-Reuse

Input: Graph G, feature set $\boldsymbol{X} = \{\boldsymbol{x}_f\}$, base set size n_B, decomposition threshold δ_0, precision factor γ, error bound λ
Output: Approximate embedding matrix $\hat{\boldsymbol{P}}$

1 $\beta_s \leftarrow \frac{\lambda^2}{(2\lambda/3+2)\cdot\log(2n)}$
2 $M(\boldsymbol{x}_f) \leftarrow 0$ for all $\boldsymbol{x}_f \in \boldsymbol{X}$, $\boldsymbol{X}_B = \emptyset$
3 **for all** $\boldsymbol{x}_f \in \boldsymbol{X}$ **do**
4 $\boldsymbol{x}_{f^*} \leftarrow \arg\min_{\boldsymbol{x}_{f^*} \in \boldsymbol{X}} \|\boldsymbol{x}_{f^*} - \boldsymbol{x}_f\|_1$
5 $M(\boldsymbol{x}_{f^*}) \leftarrow M(\boldsymbol{x}_{f^*}) + 1$
6 **for** i from 1 to n_B **do**
7 $\boldsymbol{b}_i \leftarrow \arg\max_{\boldsymbol{b}_i \in \boldsymbol{X}} M(\boldsymbol{b}_i)$
8 $\boldsymbol{X}_B \leftarrow \boldsymbol{X}_B \cup \boldsymbol{b}_i$, $\boldsymbol{X} \leftarrow \boldsymbol{X} - \boldsymbol{b}_i$
9 **for** i from 1 to n_B **do**
10 $\hat{\boldsymbol{\pi}}_i \leftarrow$ Apply Alg. 1 on $\boldsymbol{b}_i$ with $\beta^* = \gamma\beta_s$
11 **for all** $\boldsymbol{x}_f \in \boldsymbol{X}$ **do**
12 $\theta_i \leftarrow 0$ for i from 1 to n_B
13 $\boldsymbol{x}' \leftarrow \boldsymbol{x}_f$, $\delta \leftarrow 1$, $\vartheta \leftarrow 1$
14 **while** $\vartheta \cdot \delta > \delta_0$ **do**
15 $\boldsymbol{b}_i \leftarrow \arg\min_{\boldsymbol{b}_i \in \boldsymbol{X}_B} \|\boldsymbol{x}' - \boldsymbol{b}_i\|_1$
16 $\vartheta \leftarrow \arg\min_{\vartheta} \|\boldsymbol{x}' - \vartheta\boldsymbol{b}_i\|_1$, $\delta \leftarrow \delta/2$
17 $\boldsymbol{x}' \leftarrow \boldsymbol{x}' - \vartheta\boldsymbol{b}_i$, $\theta_i \leftarrow \theta_i + \vartheta$
18 $\boldsymbol{\pi}_f^* \leftarrow$ Apply Alg. 1 on $\boldsymbol{x}'$ with $\beta' = \left(1 - \gamma\sum_{i=1}^{n_B}\theta_i\right)\beta_s$
19 **for** i from 1 to n_B **do**
20 $\boldsymbol{\pi}_f^* \leftarrow \boldsymbol{\pi}_f^* + \theta_i \cdot \hat{\boldsymbol{\pi}}_i$
21 **return** $\hat{\boldsymbol{P}} = \left(\boldsymbol{\pi}_1^*, \cdots, \boldsymbol{\pi}_F^*\right)$

approximation in Definition 3.1? To investigate the estimation error of feature vectors, we propose the following lemma. All the missing proofs in this paper can be found in [1].

Lemma 3.4. *Given a feature vector $\boldsymbol{x}_f$, the ground truth of PPR vector is $\boldsymbol{\pi}(\boldsymbol{x}_f)$, and the estimation output by Eq. (9) is $\boldsymbol{\pi}^*(\boldsymbol{x}_f)$. For any respective element $\pi(\boldsymbol{x}_f; t)$ and $\pi^*(\boldsymbol{x}_f; t)$, $|\pi(\boldsymbol{x}_f; t) - \pi^*(\boldsymbol{x}_f; t)| \leq \lambda$ holds with probability at least $1 - \phi$, for β' such that $\beta' > \beta^*$ and*

$$\beta' \leq \frac{\lambda^2/\log(2/\phi) - 2\sum_{i=1}^{n_B} \theta_i \beta^*}{2\lambda/3 + 2}. \tag{10}$$

Lemma 3.4 indicates that, when choosing a smaller push coefficient β^* for base vectors, the coefficient β' can be larger and reduce the cost of PPR computation on most feature vectors. We can thence derive the following lemma, which states that the setting in Algorithm 2 satisfies Definition 3.1:

Lemma 3.5. *Given a feature set $\boldsymbol{X}$, for any feature vector $\boldsymbol{x}_f \in \boldsymbol{X}$, Algorithm 2 returns an approximate PPR vector $\hat{\boldsymbol{\pi}}(\boldsymbol{x}_f)$, that any of its elements $\hat{\pi}(\boldsymbol{x}_f; t)$ satisfies Eq. (8) with at least $1 - \phi$ probability.*

Proof. In Algorithm 2 there is $\beta^* = \gamma\beta_s$. Then β' satisfies:

$$\beta' \leq \frac{\lambda^2/\log(2/\phi)}{2\lambda/3 + 2} - \sum_{i=1}^{n_B} \beta^* \theta_i \leq \frac{\lambda^2/\log(2/\phi) - 2\sum_{i=1}^{n_B} \beta^* \theta_i}{2\lambda/3 + 2}. \tag{11}$$

Therefore, β^* for base vectors and β' for remaining vectors satisfy Eq. (10). According to Lemma 3.4 this lemma follows. □

3.4 Complexity Analysis

We then develop theoretical analysis on the time and memory complexity of SCARA. We have the following lemma:

Lemma 3.6. *The time complexity of Feature-Push is $O(\sqrt{\frac{m\|\boldsymbol{x}\|_1}{\beta}})$.*

Proof. We analyze the two parts of Algorithm 1 separately. The forward push with early termination runs in $O(\|\boldsymbol{x}\|_1/r_{max})$ according to [3]. For the random walks in Feature-Push, we employ the complexity derived by [31] as $O(m \cdot r_{max}/\beta)$. Hence the overall running time of one query in Algorithm 1 is bounded by $O\left(\frac{\|\boldsymbol{x}\|_1}{r_{max}} + r_{max} \cdot \frac{m}{\beta}\right)$ .By applying Lagrange multipliers, the complexity is minimized by selecting $r_{max} = \sqrt{\frac{\beta\|\boldsymbol{x}\|_1}{m}}$. □

According to Lemma 3.6, the time complexity of computing $\hat{\pi}(\boldsymbol{x}, \beta)$ with Algorithm 1 can be bounded by $O(\sqrt{m\|\boldsymbol{x}\|_1/\beta})$. To get PPR value with absolute error guarantee of λ, Algorithm 1 requires a push coefficient $\beta_s = \frac{\lambda^2/\log(2/\phi)}{2\lambda/3+2}$. Then without Feature-Reuse, the time complexity for computing PPR value for each normalized feature vector is bounded by $O(\sqrt{m/\beta_s})$.

When Feature-Reuse applies, let $\theta_{sum} = \sum_{i=1}^{n_B} \theta_i$ denote the proportion of a feature $\boldsymbol{x}_f$ computed by base vectors, and the L1 length of the rest $\boldsymbol{x}'$ is $1-\theta_{sum}$. In Algorithm 2, we compute the remaining part with push coefficient of $(1-\gamma\theta_{sum})\beta_s$, where $0 < \gamma \leq 1$. Recalling that the L1 length of the feature vector is reduced by θ_{sum} with Feature-Reuse, we derive the time complexity of Feature-Reuse on $\boldsymbol{x}$ is $O\left(\sqrt{\frac{m(1-\theta_{sum})}{\beta_s(1-\gamma\theta_{sum})}}\right)$, which is $\sqrt{\frac{1-\theta_{sum}}{1-\gamma\theta_{sum}}}$ times smaller than those without Feature-Reuse.

For example, if we compute $\theta_{sum} = 1/2$ for a vector $\boldsymbol{x}_f$ with the base vectors, and set $\gamma = 1/4$, then the complexity of computing the PPR for $\boldsymbol{x}_f$ is $O(\sqrt{4m/7\beta_s})$, which is substantially better than the consumption without Feature-Reuse $O(\sqrt{m/\beta_s})$. The overhead of each base vector is $O(\sqrt{4m/\beta_s})$, which is only twice slower than the original complexity. As we select only a few base vectors, the additional overhead produced by computing base vectors is neglectable compared with the acceleration gained.

When Feature-Reuse applies, the complexity of computing a feature vector is not worse than the complexity without Feature-Reuse, and is equivalent to the latter only when $\theta_{sum} = 0$ (i.e. the feature vector is completely orthogonal with the base vectors). Therefore in the worst case, the complexity of SCARA on feature matrix $\boldsymbol{X}$ is equivalent to repeating F queries of Algorithm 1. By setting $\phi = 1/n$, we can derive the time overhead of SCARA precomputation. For the complexity of memory, the usage of a single-query Feature-Push can be denoted as $O(n)$. Hence the precomputation complexity of SCARA is given by the following theorem:

Theorem 3.7. *Time complexity of SCARA precomputation stage is bounded by $O\left(F\sqrt{m\log n}/\lambda\right)$. Memory complexity is $O(nF)$.*

4 EXPERIMENTAL EVALUATION

4.1 Experiment Setting

Datasets. We adopt benchmark datasets of different graph properties, feature dimensions, and data splitting for large-scale node classification tasks. We present the dataset statistics in Table 2. Among the datasets, PPI, Yelp, and Amazon are for *inductive* learning, where the training and testing graphs are different and require separate graph precomputation and propagation. The given original node splittings are in Table 2. The learning tasks on the other datasets are *transductive* and are performed on the same graph structure. For a dataset with N_c target classes, we refer to convention in [6, 19] to randomly select two sets of $20N_c$ and $200N_c$ nodes for training and validation, respectively, and the rest labeled nodes in the graph as the testing set.

Table 2: Dataset statistics and parameters. "Split" is the percentage of nodes in training/validation/testing set. "(i)" and "(t)" stand for inductive and transductive tasks. "(m)" and "(s)" stand for multiple and single target classifications.

Dataset	Nodes n	Edges m	Features F	Classes N_c	Split	Probability α	Convolution r	Common
PPI [16]	56,944	818,716	50	121 (m)	0.79/0.11/0.10 (i)	0.3	0.0	
Yelp [37]	716,847	6,977,410	300	100 (m)	0.75/0.10/0.15 (i)	0.9	0.3	$\lambda = 1 \times 10^{-4}$
Reddit [16]	232,965	114,615,892	602	41 (s)	0.01/0.04/0.96 (t)	0.5	0.5	$n_B = 0.02n_U$
Amazon [11]	2,400,608	123,718,024	100	47 (s)	0.70/0.15/0.15 (i)	0.2	0.2	$\gamma = 0.2$
MAG [35]	27,394,820	366,143,207	200	100 (m)	0.01/0.01/0.99 (t)	0.5	0.5	$\delta_0 = 1/16$
Papers100M [17]	111,059,956	1,615,685,872	128	172 (s)	0.78/0.08/0.14 (t)	0.2	0.5	

Table 3: Average results of SCARA and baselines on large-scale datasets for transductive and inductive learning. "Learn" and "Infer" columns are the learning (sum of precomputation and training) and inference time (s), respectively. "Mem." is the peak RAM memory (GB). "F1" is the micro F1-score (%) on testing sets. "OOM" stands for out of memory error, "> 12h" means the model requires more than 12h clock time to produce proper results. The respective models of first and second best performance in "Learn", "Infer", "Mem.", and "F1" columns are marked in bold and underlined font.

Transductive	Reddit						MAG						Papers100M					
	Learn	(Pre.	+ Train)	Infer	Mem.	F1	Learn	(Pre.	+ Train)	Infer	Mem.	F1	Learn	(Pre.	+ Train)	Infer	Mem.	F1
GraphSAINT	51.5	–	51.5)	26.1	11.1	30.7 ±3.0	–	–	–	–	OOM	–	–	–	–	–	OOM	–
GAS	3563	–	3563)	**0.1**	14.6	38.0 ±0.2	–	–	–	–	OOM	–	–	–	–	–	OOM	–
PPRGo	163	157	+ 4.8)	74.1	8.0	31.0 ±1.7	–	> 12h	–	–	146	–	–	–	–	–	OOM	–
GBP	1891	(2127	+ 16.3)	6.2	8.4	39.2 ±0.3	4572	(4470	+ 102)	1433	177 *	34.8 ±0.1	–	–	–	–	OOM	–
SCARA (ours)	**12.0**	(1.8	+ 10.6)	4.8	**4.7**	**40.3 ±0.7**	**460**	(380	+ 80.0)	**1421**	**49.4**	**35.0 ±0.3**	**1471**	(83.5	+ 1388)	**2.8**	**63.7**	**35.5 ±0.8**

Inductive	PPI						Yelp						Amazon					
	Learn	(Pre.	+ Train)	Infer	Mem.	F1	Learn	(Pre.	+ Train)	Infer	Mem.	F1	Learn	(Pre.	+ Train)	Infer	Mem.	F1
GraphSAINT	2813	–	2813)	4.1	3.2	89.3 ±0.2	8589	–	8589)	193	54.0	**64.9 ±0.1**	2612	–	2612)	804	87.9	81.3 ±0.1
GAS	326	–	326)	**0.1**	6.6	99.3 ±0.1	3622	–	3622)	**0.1**	22.0	63.8 ±0.0	19218	–	19218)	**0.4**	41.7	71.7 ±0.5
PPRGo	4019	(70.0	+ 3949)	1.7	2.7	50.1 ±0.7	13073	(91.9	+12981)	30.1	16.9	56.5 ±2.6	3041	(2092	+ 949)	63.3	27.4	78.4 ±3.0
GBP	86.4	(18.5	+ 67.9)	0.3	**2.5**	99.3 ±0.0	198	(77.2	+ 121)	2.9	13.4	60.6 ±0.1	2193	(1019	+ 1174)	7.5	13.4	**86.8 ±0.1**
SCARA (ours)	**49.3**	(0.5	+ 48.9)	0.3	2.5	**99.3 ±0.0**	**154**	(3.6	+ 150)	3.1	**7.4**	61.4 ±0.4	**1281**	(7.0	+ 1274)	6.8	**7.3**	83.8 ±0.1

* GBP experiment of this entry is conducted on a different machine with a larger 192GB RAM.

Metrics. Predictions on datasets PPI, Yelp, and MAG are multi-label classification having multiple targets for each node. The other tasks are multi-class with only one target class per node. We uniformly utilize micro F1-score to assess the model prediction performance. The evaluation is conducted on a machine with Ubuntu 18 operating system, with 160GB RAM, an Intel Xeon CPU (2.1GHz), and an NVIDIA Tesla K80 GPU (11GB memory). The implementation is by PyTorch and C++.

Baseline Models. We select the state-of-the-art models of different scalable GNN methods analyzed in Section 2 as our baselines. GraphSAINT-RW [37] and GAS [13] are representative of different sampling-based algorithms. For post- and pre-propagation decoupling approaches, we respectively employ the most advanced PPRGo [6] and GBP [9]. For a fair comparison, we mostly retain the implementations and settings from original papers and source codes. We uniformly apply single-thread executions for all models.

Hyperparameters. Propagation parameters α, r, λ and FEATURE-REUSE parameters n_B, γ, δ_0 are presented in Table 2 per dataset. For neural network architecture we set layer depth $L = 4$, layer width $W = 2048$ and $W = 128$ for inductive and transductive tasks, respectively, to be aligned with optimal baseline results in [9]. In model optimization, we employ mini-batch training with respective batch size 2048 and 64 for inductive and transductive learning, for a maximum of 1000 epochs with early stopping.

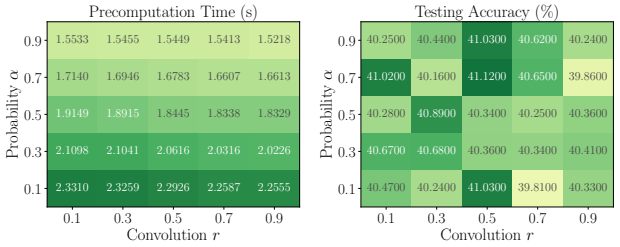
The effects of α and r values on accuracy and efficiency metrics are shown in Figure 1, which indicates that our selections of parameter values are efficient and do not influence GNN performance. The full exploration on hyperparameters can be found in [1].

4.2 Performance Comparison

We evaluate the performance of SCARA and baselines in terms of both effectiveness and efficiency. Table 3 shows the average results of repetitive experiments on 6 large datasets, including the assessments on accuracy, memory, and the running time for different phases. Among them the key metric is learning time, which is summed up by precomputation and training times and presents the efficiency through the information retrieving process to acquire an effective model. The training curves are given in Figure 2.

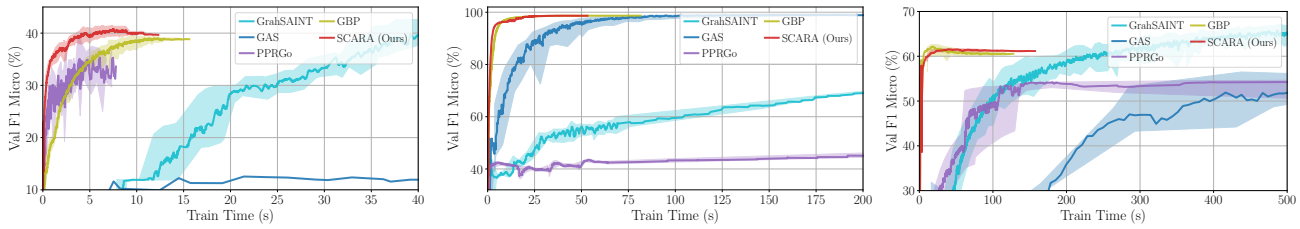
As an overview, the experimental results demonstrate the superiority of our model achieving scalability throughout the learning phase. On all datasets, SCARA reaches $5 - 100\times$ acceleration in precomputation time than the best decoupling method, as well as comparable or better training and inference speed, and significantly better memory overhead. When the graphs are scaled-up, the time and memory footprints of SCARA increase relatively slower than our GNN baselines, which is in line with our complexity analysis. For prediction performance, SCARA converges in all tasks and outputs comparable or better accuracy than other scalable competitors.

From the aspect of time efficiency, our SCARA model effectively speeds up the learning process in all tasks, mostly benefiting from the fast and scalable precomputation for graph propagation. The simple neural model forwarding implemented in mini-batch approach also contributes to the efficient computation of model training and inference. On the largest available dataset Papers100M, our method efficiently completes precomputation in 100 seconds, and finishes learning in an acceptable length of time, showing the scalability of processing billion-scale graphs. In comparison,



(a) Precomputation Time **(b) Testing Accuracy**

Figure 1: Effect of propagation parameters α and r on SCARA (a) efficiency and (b) accuracy for Reddit dataset.



(a) Reddit (b) PPI (c) Yelp

Figure 2: Validation F1 convergence curves of SCARA and baseline models on (a) Reddit, (b) PPI, and (c) Yelp datasets. Curves only represents the process of training phase. Shaded area is the result range of multiple runs.

sampling-based GraphSAINT and GAS achieve good performance on several datasets, but the $O(ILmF)$ term in training complexity results in great slowdown when graphs are scaled-up. GraphSAINT is costly for its full-batch prediction stage on the whole graph, which is usually only executable on CPUs. GAS is particularly fast for inference, but it comes with the price of trading off memory expense and training time to manipulate its cache. The propagation decoupling models PPRGo and GBP show better scalability, but take more time than SCARA to converge, due to the graph information yielded by precomputation algorithms. It can be seen that their node-based propagation computations are less efficient when the graph sizes become larger, which aligns with Table 1 complexity analysis. Remarkably, SCARA achieves about 100× and 40× faster for precomputation than these two competitors on Reddit and PPI.

Regarding memory overhead, our method also demonstrates its efficiency benefit from its scalable implementation. We discover that the major memory expense of SCARA only increases proportional to the graph attribute matrix, while PPRGo and GBP usually demand twice as large RAM, and GraphSAINT and GAS use even more for their samplers. On the billion-scale Papers100M graph, most baselines meet out of memory error in our machine.

For learning effectiveness, SCARA achieves similar or better F1-score compared with current GNN baselines. For most datasets, our model outperforms both the state-of-the-art pre-propagation approach GBP and the scalable post-propagation baseline PPRGo. It is worth noting that most baselines fail to or only partially converge before training terminates in certain tasks.

Figure 2 shows the validation F1-score versus training time on representative datasets and corresponding GNN models. It can be observed that when comparing the time consumption to convergence, the SCARA model is efficient in reaching the same precision faster than most methods. The performances of GraphSAINT, GAS, and PPRGo in the figure are relatively suboptimal because they require more training time beyond the display scopes to converge.

4.3 Effect of Feature-Reuse

To examine the Feature-Reuse technique utilized in our model, we conduct an ablation study to compare the precomputation performance of SCARA by applying the Feature-Reuse as in Algorithm 2 or without reusing features and only full-precision Feature-Push computation. We choose the Reddit dataset to generate trimmed feature matrices with different dimensions F to evaluate the feature-oriented optimization. The results of average times and testing accuracies over these feature matrices are given in Table 4.

By comparing the relative speed-up in Table 4, we state that Feature-Reuse substantially reduces the precomputation time for different node feature sizes. When the number of features increases, the algorithm benefits more acceleration from adopting the feature optimization scheme, and achieves up to 1.6× speed-up compared to Feature-Push propagation without reuse. Meanwhile, Feature-Reuse causes no significant difference on effectiveness as minor accuracy fluctuations are under the error bound of repetitive experiments. More detailed results can be found in [1].

Table 4: Effect of SCARA with and without Feature-Reuse on precomputation time (s) and testing accuracy (%) for Reddit dataset with different feature dimensions F.

	Feature	$F = 100$	$F = 200$	$F = 400$	$F = 602$
Pre. Time	w/o Reuse	0.46	0.93	1.83	2.84
	w/ Reuse	0.35	0.67	1.30	1.85
	Speed-up	133%	138%	141%	155%
Accuracy	w/o Reuse	27.7	31.9	37.0	40.5
	w/ Reuse	27.8	31.7	36.7	40.3
	Δ	+0.1	−0.2	−0.3	−0.2

5 CONCLUSION

In this paper, we propose SCARA, a scalable Graph Neural Network algorithm with feature-oriented optimizations. Our theoretical contribution includes showing the SCARA model has a sub-linear complexity that efficiently scale-up the graph propagation by Feature-Push and Feature-Reuse algorithms. We conduct extensive experiments on various datasets to demonstrate the scalability of SCARA in precomputation, training, and inference. Our model is efficient to process billion-scale graph data and achieve up to 100× faster than the current state-of-the-art scalable GNNs in precomputation, while maintaining comparable or better accuracy.

ACKNOWLEDGMENTS

This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 1 Seed (RS05/21) and in part by NTU startup grant. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore. Xiang Li is supported by Shanghai Pujiang Talent Program (Project No. 21PJ1402900) and Shanghai Science and Technology Committee General Program (Project No. 22ZR1419900). We also thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] 2022. SCARA Technical Report. https://sites.google.com/view/scara-techreport

[2] Rami Al-Rfou, Bryan Perozzi, and Dustin Zelle. 2019. DDGK: Learning Graph Representations for Deep Divergence Graph Kernels. In *The World Wide Web Conference* (San Francisco, CA, USA). 37–48.

[3] Reid Andersen, Fan Chung, and Kevin Lang. 2006. Local Graph Partitioning using PageRank Vectors. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. IEEE, 475–486. https://doi.org/10.1109/FOCS.2006.44

[4] James Atwood and Don Towsley. 2016. Diffusion-convolutional neural networks. *29th Advances in Neural Information Processing Systems* (2016), 2001–2009. arXiv:1511.02136

[5] Rianne van den Berg, Thomas N Kipf, and Max Welling. 2018. Graph convolutional matrix completion. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

[6] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. 2020. Scaling Graph Neural Networks with Approximate PageRank. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2020), 2464–2473.

[7] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations*.

[8] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic training of graph convolutional networks with variance reduction. *35th International Conference on Machine Learning* 3 (2018), 1503–1532. arXiv:1710.10568

[9] Ming Chen, Zhewei Wei, Bolin Ding, Yaliang Li, Ye Yuan, Xiaoyong Du, and Ji Rong Wen. 2020. Scalable graph neural networks via bidirectional propagation. *33rd Advances in Neural Information Processing Systems* (2020).

[10] Zhengdao Chen, Joan Bruna, and Lisha Li. 2019. Supervised community detection with line graph neural networks. *7th International Conference on Learning Representations* (2019).

[11] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 257–266.

[12] Mucong Ding, Kezhi Kong, Jingling Li, Chen Zhu, John P Dickerson, Furong Huang, and Tom Goldstein. 2021. VQ-GNN: A Universal Framework to Scale up Graph Neural Networks using Vector Quantization. *34th Advances in Neural Information Processing Systems* (2021).

[13] Matthias Fey, Jan E. Lenssen, Frank Weichert, and Jure Leskovec. 2021. GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings. In *38th International Conference on Machine Learning*. PMLR 139.

[14] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. 2005. Towards Scaling Fully Personalized PageRank: Algorithms, Lower Bounds, and Experiments. *Internet Mathematics* 2, 3 (jan 2005), 333–358. https://doi.org/10.1080/15427951.2005.10129104

[15] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning in Large Attributed Graphs. *30th Advances in Neural Information Processing Systems* (oct 2017). arXiv:1710.09471

[16] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[17] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, Jure Leskovec, Regina Barzilay, Peter Battaglia, Yoshua Bengio, Michael Bronstein, Stephan Günnemann, Will Hamilton, Tommi Jaakkola, Stefanie Jegelka, Maximilian Nickel, Chris Re, Le Song, Jian Tang, Max Welling, and Rich Zemel. 2020. Open Graph Benchmark : Datasets for Machine Learning on Graphs. *33rd Advances in Neural Information Processing Systems* (2020).

[18] Zengfeng Huang, Shengzhong Zhang, Chong Xi, Tang Liu, and Min Zhou. 2021. Scaling Up Graph Neural Networks Via Graph Coarsening. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, Vol. 1. 675–684.

[19] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*.

[20] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2019. Predict then propagate: Graph neural networks meet personalized PageRank. *7th International Conference on Learning Representations* (2019), 1–15.

[21] Xiang Li, Renyu Zhu, Yao Cheng, Caihua Shan, Siqiang Luo, Dongsheng Li, and Weining Qian. 2022. Finding Global Homophily in Graph Neural Networks When Meeting Heterophily. In *39th International Conference on Machine Learning*. arXiv:2205.07308

[22] Dandan Lin, Raymond Chi-Wing Wong, Min Xie, and Victor Junqiu Wei. 2020. Index-Free Approach with Theoretical Guarantee for Efficient Random Walk with Restart Query. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 913–924. https://doi.org/10.1109/ICDE48307.2020.00084

[23] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report.

[24] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. 2015. An Overview of Microsoft Academic Service (MAS) and Applications. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy). 243–246.

[25] Lichao Sun, Yingtong Dou, Carl Yang, Ji Wang, Philip S. Yu, Lifang He, and Bo Li. 2018. Adversarial Attack and Defense on Graph Data: A Survey. *arXiv e-prints* (2018).

[26] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. 2018. Attention-based graph neural network for semi-supervised learning. *arXiv e-prints* (2018). arXiv:1803.03735v1

[27] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. In *8th International Conference on Learning Representations*.

[28] Chun Wang, Shirui Pan, Guodong Long, Xingquan Zhu, and Jing Jiang. 2017. MGAE: Marginalized Graph Autoencoder for Graph Clustering. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (Singapore, Singapore) *(CIKM '17)*. Association for Computing Machinery, New York, NY, USA, 889–898. https://doi.org/10.1145/3132847.3132967

[29] Hanzhi Wang, Mingguo He, Zhewei Wei, Sibo Wang, Ye Yuan, Xiaoyong Du, and Ji Rong Wen. 2021. Approximate Graph Propagation. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Vol. 1. Association for Computing Machinery, 1686–1696.

[30] Sibo Wang, Renchi Yang, Runhui Wang, Xiaokui Xiao, Zhewei Wei, Wenqing Lin, Yin Yang, and Nan Tang. 2019. Efficient Algorithms for Approximate Single-Source Personalized PageRank Queries. *ACM Transactions on Database Systems* 44, 4 (dec 2019), 1–37. https://doi.org/10.1145/3360902 arXiv:1908.10583

[31] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: Simple and effective approximate single-source personalized PageRank. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* Part F1296 (2017), 505–514.

[32] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying Graph Convolutional Networks. In *Proceedings of the 36th International Conference on Machine Learning*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. 6861–6871.

[33] Hao Wu, Junhao Gan, Zhewei Wei, and Rui Zhang. 2021. Unifying the Global and Local Approaches: An Efficient Power Iteration with Forward Push. In *Proceedings of the 2021 International Conference on Management of Data*, Vol. 1. 1996–2008.

[34] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (1 2021), 4–24.

[35] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, Juncheng Liu, and Sourav S Bhowmick. 2021. Scaling Attributed Network Embedding to Massive Graphs. *Proceedings of the VLDB Endowment* 14, 1 (2021), 37–49.

[36] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom). 974–983.

[37] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. GraphSAINT: Graph Sampling Based Learning Method. In *International Conference on Learning Representations*.

[38] Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. 2020. Graph-bert: Only attention is needed for learning graph representations. *arXiv e-prints* (2020). arXiv:2008.08617

[39] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in Neural Information Processing Systems* 31 (2018), 5165–5175.

[40] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2020. Deep Learning on Graphs: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 14, 8 (2020), 1–1. https://doi.org/10.1109/TKDE.2020.2981333 arXiv:1812.04202

[41] Zulun Zhu, Jiaying Peng, Jintang Li, Liang Chen, Qi Yu, and Siqiang Luo. 2022. Spiking Graph Convolutional Networks. In *31th International Joint Conference on Artificial Intelligence*. arXiv:2205.02767

[42] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. 2018. Adversarial Attacks on Neural Networks for Graph Data. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom). 2847–2856.