An Efficient Distributed Graph Engine for Deep Learning on Graphs

Gangda Deng*
Ömer Faruk Akgül*
gangdade@usc.edu
akgul@usc.edu
University of Southern California
Los Angeles, California, USA

Hongkuan Zhou
University of Southern California
Los Angeles, California, USA
hongkuaz@usc.edu

Hanqing Zeng
Yinglong Xia
Jianbo Li
zengh@meta.com
yxia@meta.com
jianboli@meta.com
Meta
Menlo Park, California, USA

Viktor Prasanna
University of Southern California
Los Angeles, California, USA
prasanna@usc.edu

ABSTRACT

Traditional graph-processing algorithms have been widely used in Graph Neural Networks (GNNs). This combination has shown state-of-the-art performance in many real-world network mining tasks. Current approaches to graph processing in deep learning face two main problems. On the one hand, easy-to-use deep learning libraries lack support for widely used graph-processing algorithms and do not provide low-level APIs for building distributed graph-processing algorithms. On the other hand, existing graph-processing libraries are not user-friendly for deep learning researchers. Their graph primitives are not designed for batch processing, which is essential for deep learning use cases. In this paper, we present an efficient and easy-to-use graph engine that incorporates distributed graph processing into deep learning ecosystems. We develop a distributed graph storage system with an efficient batching technique to minimize communication overhead incurred by Remote Procedure Calls (RPC) between computing nodes. We propose an optimized method for distributed computation of Single Source Personalized PageRank (SSPPR) using the Forward Push algorithm based on lock-free parallel maps. Experimental evaluations demonstrate significant improvement, with up to three orders of magnitude in SSPPR throughput, of our graph engine compared with the tensor-based implementation. Both methods offer necessary usability with tensor operations, which are widely used for graph processing in current deep graph libraries.

*Both authors contributed equally to this research.

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624169

CCS CONCEPTS

• **Theory of computation** → **Graph algorithms analysis**; • **Computing methodologies** → *Neural networks*; **Distributed algorithms**.

KEYWORDS

Personalized PageRank, Forward Push, Graph Neural Networks

ACM Reference Format:
Gangda Deng, Ömer Faruk Akgül, Hongkuan Zhou, Hanqing Zeng, Yinglong Xia, Jianbo Li, and Viktor Prasanna. 2023. An Efficient Distributed Graph Engine for Deep Learning on Graphs. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624062.3624169

1 INTRODUCTION

Integrating traditional graph-processing algorithms with Graph Neural Networks (GNNs) has improved the state-of-the-art performance of various real-world network mining tasks. Traditional graph-processing algorithms are widely used to improve GNNs' scalability and accuracy. Many existing methods train GNNs on graph data that has been processed by algorithms such as shortest-path [21], random walk [29, 32], and Personalized PageRank (PPR) [2, 31]. However, most existing works assume that the full graph topology can be stored in a single machine, and are not optimized for distributed settings. Many interesting real-world graph mining problems involve graphs with billions of nodes and edges, requiring distributed storage and computation across multiple machines. On the one hand, existing distributed graph processing frameworks, such as GraphX [9] in Spark, are powerful but not specifically optimized for the target algorithms like PPR. It's also challenging for ML researchers to integrate those graph-processing tasks with deep learning frameworks. On the other hand, state-of-the-art deep learning frameworks for graphs, such as PyG [5] and DGL [24], lack support for widely used graph-processing algorithms and do not

provide low-level APIs for distributed graph processing. As a result, designing an efficient and easy-to-use graph engine that incorporates distributed graph processing into deep learning ecosystems is in high demand.

Mini-batch Stochastic Gradient Descent (SGD) is widely used for distributed GNN training. For each mini-batch, existing methods construct a subgraph for the batch nodes, which may include a subset of neighboring nodes as the supporting set. Various traditional graph-processing algorithms, such as BFS [10], Random Walk [29, 32], and Personalized PageRank [33], are used for mini-batch subgraph construction. Each graph-processing task starts with the root node(s) of a batch and usually traverses a large portion of the entire graph. To handle large-scale graphs, we first partition the input graph using a min-cut graph partitioning algorithm and store graph shards in distributed machines. To take advantage of the asynchronous nature of graph-processing tasks, we use PyTorch RPC[1], a tensor-based point-to-point communication library in PyTorch, as our communication protocol. Computation related to mini-batch subgraph construction is computed locally, with only necessary information fetched from remote graph storage.

Since data communicated between machines is wrapped in tensors, the key question is if we can achieve relatively good performance by solely depending on existing tensor libraries for distributed graph processing. If not, what kind of graph-processing algorithms require the support of additional C++ operators? With graph structure information represents in tensors, tensor-based operations are widely used in state-of-the-art deep learning libraries for graphs (e.g., PyG and DGL). The tensor is a dense data structure with a fixed shape. Naive tensor operators that can be embarrassingly parallelized are well-supported by tensor libraries. Other commonly used operators like *sort*, *scatter*, and *gather* also have efficient and easy-to-use APIs. However, current tensor libraries are inefficient when building algorithms with dynamic frontier node sets. Such a set has a varying shape and favors a hashmap-like data structure. Therefore, a C++ implementation is necessary for algorithms like BFS and Forward Push [1] to avoid the unacceptable overhead of the Python interpreter when using tensor operations. The Forward Push algorithm starts at a given source node and iteratively processes subsets of visited vertices, exploring their out-neighbors until a termination condition is met. Compared to the fixed-length unbiased Random Walk, Forward Push uses frontier node sets with dynamic shapes, and there is much room for improvement in a tensor-based implementation. On a moderate-sized graph dataset (e.g., Ogbn-products), our proposed graph engine can achieve a 1.7× and 83× speedup compared to the PyTorch tensor-based implementations of distributed Random Walk and Forward Push, respectively.

To the best of our knowledge, there is currently no efficient implementation of Forward Push on a distributed graph that has plug-and-play compatibility for deep learning applications. In this work, we present a distributed solution that performs the Forward Push approximation to compute single-source PPR (SSPPR) queries efficiently and in a user-friendly way. To further extend our methods for general use, we devise a distributed graph storage using PyTorch RPC to support graph traversal across machines. However, we encounter a new challenge: how can we provide low-level APIs while eliminating the inefficiencies? PyTorch RPC is easy to maintain and integrate, but has poor performance when frequently transferring small tensors with non-equal lengths. Additionally, it is challenging to present an efficient approach for computing PPR values with required neighbor information from local and remote graph storage. To tackle these issues, we make the following contributions:

- We implement an efficient and easy-to-use graph engine for deep learning on graphs. Our engine allows for easy integration of existing single-machine graph primitives, enabling the simple implementation of distributed graph processing algorithms.
- We design a distributed min-cut clustering graph storage for general graph-processing algorithms. We further present efficient batching techniques to reduce the number of RPC requests and compress each response, greatly reducing the remote fetch overhead.
- We develop an efficient method for computing SSPPR by parallelizing the Forward Push algorithm using the lock-free parallel map.
- Extensive experimental results show that our map-based solution is significantly faster than the tensor-based implementation for distributed SSPPR computing, with up to three orders of magnitude improvement in throughput.

2 BACKGROUND

PPR has been widely used in GNNs for large-scale graphs due to its ability to overcome the limitations of traditional graph convolutional layers and to address the problem of over-smoothing and neighbor explosion [30]. SSPPR is a variant of PPR that computes PPR scores for a single source node, which is particularly useful for subgraph sampling and node embedding [26].

Many GNN models incorporate PPR to improve aggregation quality. GDC [8] applies PPR diffusion directly to the input graph. PPNP [7] and PPRGo [2] devise an improved propagation scheme based on PPR. GBP [4] further generalizes PPR to a bidirectional propagation algorithm for feature propagation. In another line of research, ShaDow [33] samples the mini-batch subgraphs based on the top-K PPR values for arbitrary deep GNNs. Above works have shown promising results in improving the throughput and accuracy of PPR-based methods, thereby making them more practical for real-world GNN applications.

2.1 Preliminary

2.1.1 Single Source Personalized PageRank. Personalized PageRank is an algorithm for ranking nodes based on their significance for a given set of query nodes in a graph [13]. Given a source node s and a teleport factor α, PPR computes the probability that a random walk with restart from s terminates at each node v in the graph. This probability is referred to as the personalized PageRank $\pi(s, v)$ of v with respect to s. Single-source Personalized PageRank is a variant of PPR that finds the top-k nodes with the highest PPR values for a given source node s in a graph G. For simplicity, we focus on the

[1] https://pytorch.org/docs/stable/rpc.html

Algorithm 1: Forward Push Algorithm

Input : Edge-weighted graph $G(V, E, W)$, teleport probability α, maximum residual ε, target node t.

Output: An ε-approximate weighted PPR vector $\pi^{(\varepsilon)}$.

Function push$(v, \alpha, \pi, r, W, d_w)$**:**
- $\pi^{(v)} \leftarrow \pi^{(v)} + \alpha \cdot r(v)$
- $m \leftarrow (1-\alpha) \cdot r(v)$
- $r(v) \leftarrow 0$
- **for** $u \in N_{out}(v)$ **do**
  - $r(u) \leftarrow r(u) + \frac{W(v,u)}{d_w(v)} \cdot m$
- **end for**
- **return**

$\pi^{(\varepsilon)}(v) \leftarrow 0$ for all vertices $v \in V$

$r(t) \leftarrow 1,\ r(v) \leftarrow 0$ for all vertices $v \in V - \{t\}$

$d_w(v) \leftarrow \sum_{u \in N_{out}(v)} W(v,u)$ for all vertices $v \in V$

while $\exists v\ s.t\ r(v) > \varepsilon \cdot d_w(v)$ **do**
- apply push$(v, \alpha, \pi, r, W, d_w)$ at vertex v, updating $\pi(v)$, $r(v)$, and $r(u)$, where $u \in N_{out}(v)$

end while

return $\pi^{(\varepsilon)}$

distributed computation of approximate SSPPR processing without top-k selection, referred to as approximate whole-graph SSPPR [25].

2.1.2 Throughput of distributed SSPPR. To be consistent with the concept of batched SGD in GNN training, we denote the throughput as the total number of processed PPR queries (source nodes) per second across all machines. We assume the root nodes of a batch are evenly distributed across all machines. Each root node is considered the source node of an SSPPR query. We then collect the overall runtime of processing a large number of queries (e.g., 128) per machine in parallel, including synchronization time, and calculate the throughput.

2.1.3 Forward Push Algorithm. The Forward Push algorithm [1] is a graph traversal algorithm used for computing personalized PageRank. It propagates scores from the source node to its neighbors using a priority queue and is commonly used due to its efficiency and scalability [23]. In the following sections, we refer to the set of nodes whose residual value $r(v)$ satisfies the condition $r(v) > \varepsilon \cdot d_w(v)$ as *activated nodes*. Algorithmic details regarding the forward push are shown in Algorithm 1.

2.2 Related Work

2.2.1 PPR Computation Methods. There are three main categories of methods for computing PPR vectors: matrix-based methods, local-update based methods, and Monte-Carlo based methods. Matrix-based methods, such as power iteration [16] and Lanczos method [19], require computing the entire transition matrix, which can be expensive for large graphs. Local-update based methods, such as Push and Pull [20] and Reverse Push [1], update the PPR vectors of neighboring nodes iteratively. However, these methods require many iterations to converge, especially for nodes that are far from the source node. Monte-Carlo based methods, such as Random Walk with Restart [23] and Heat Kernel[6], estimate PPR vectors by simulating random walks on the graph. However, they suffer from high variance and require many iterations to achieve accurate results. For large graphs, these methods can be computationally expensive and memory-intensive. Forward Push method [3] is a recent advancement that overcomes these limitations by partitioning the graph and computing PPR vectors using forward push on each partition. However, it requires a global synchronization step and may not scale well for highly skewed graphs.

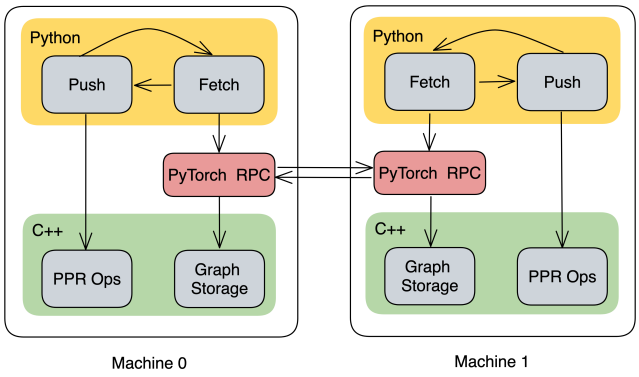


Figure 1: Overview

2.2.2 Parallelizing Personalized PageRank. One of the earliest works for computing personalized PageRank in a distributed setting is the graph partitioning approach proposed by Kamvar et al. [15], which divides the graph into disjoint subgraphs and computes the PageRank vector of each subgraph independently. Another popular approach is the parallelization of the power iteration algorithm using MapReduce, as proposed by Jeh and Widom [14]. Recently, distributed methods that use GraphX [28], a graph processing framework built on top of Apache Spark, have gained popularity. These methods typically store the graph as a distributed graph partition, where each partition is stored in the memory of a different machine.

3 APPROACH

3.1 Overview

In this work, we propose a distributed graph engine, optimized for SSPPR queries. To achieve efficient access to remote graph shards, we construct a distributed Graph Storage using the distributed RPC module in PyTorch with point-to-point asynchronous communication. For SSPPR computation, we choose the basic Forward Push [1] algorithm to approximate the exact SSPPR results. To avoid the inefficiency of current tensor libraries, we implement C++ Forward Push operators (denoted as PPR Ops) for storing and updating the intermediate results.

Figure 1 illustrates the high-level architectural design of the proposed distributed PPR engine. The use of native tensor communication libraries greatly enhances the compatibility of our engine with deep graph learning methods. All intermediate variables are wrapped in tensors, allowing our low-level Python APIs to work seamlessly with other tensor operations or graph primitives supported by deep graph libraries. Regarding the Forward Push process

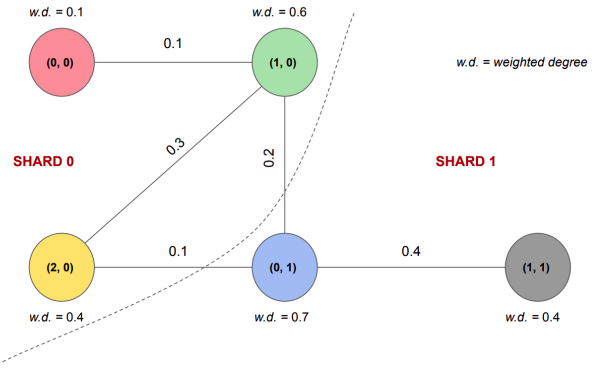


Figure 2: The nodes are denoted in (x, y) format, where x represents the local ID and y represents the shard ID. Additionally, the numbers on the edges represent the corresponding edge weights. Notably, a dotted line is utilized to indicate the separation between shards. Specifically, the node $(0, 1)$ serves as a halo node for Shard 0, while the nodes $(2, 0)$ and $(1, 0)$ function as halo nodes for Shard 1.

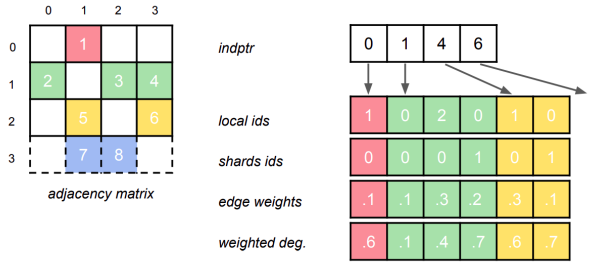


Figure 3: Shards only store the data about core nodes. Therefore, the third row in the adjacency matrix (with dotted line) is not included in the storage of Shard 0.

in Algorithm 1, the algorithm iteratively fetches the neighborhood of activated nodes (i.e., nodes with residual values greater than the threshold) and pushes the values of activated nodes to their neighbors. Following the owner compute rule, which dispatches computation to the data owner to reduce network communication, we assign each SSPPR query to the machine that hosts the corresponding graph partition of the source node. During the fetch step, computing processes access local and remote graph storage to obtain neighborhood information. In the push step, the visited nodes are stored and updated locally. By utilizing a min-cut graph partitioning algorithm, most of the nodes visited by the Forward Push algorithm are locally available via shared memory, reducing network traffic significantly.

In our implementation, we create a Remote Reference for each Graph Storage object and pass these references to every computing process. The Remote Reference serves as a distributed shared pointer to a local or remote Object, while the Graph Storage object provides operations on the local shard data. In the production scenario, each machine processes a batch of SSPPR queries in parallel. Therefore, a number of computing processes would register in the RPC group on each machine. Each process on each machine owns a unique Graph Storage object that points to the same local graph shard data in shared memory. Additionally, we provide a batch version for fetch and push operations, which significantly reduces the overall runtime.

Figure 4 shows Python code snippets for the iteration loop of distributed graph algorithms based on our proposed graph engine. Here we leverage two operations of Distributed Graph Storage: *get_neighbor_infos* and *sample_one_neighbor*. Both functions take the destination shard ID and a list of local vertex IDs as input, and output graph information related to the source vertex ID list from the remote/local machine. As discussed in Section 1, Random Walk can be efficiently performed using only tensor operations. However, for Personalized PageRank, additional C++ operators (*pop* and *push*) are required for efficient computation. Following this design, our proposed PPR engine can be easily extended to other graph processing algorithms, enabling efficient distributed computing for localized C++ graph operators.

3.2 Distributed Graph Storage

3.2.1 Graph Partitioning. For large-scale graph computations with billions of nodes and edges, calculating PPR in a distributed setting requires partitioning the graph into multiple shards. This prevents exceeding the memory capacity of a single machine, with each shard assigned to a different machine. Graph partitioning is a preprocessing step before distributed PPR computing. Once the input graph is partitioned, it can be used to compute many SSPPR queries, which amortizes the overhead.

To ensure efficient and effective partitioning, we employ the widely-used METIS [17] partitioning algorithm. METIS works by minimizing the number of edges between different partitions while ensuring a balanced graph partitioning. After partitioning the graph into several non-overlapping vertex sets, we assign all the 1-hop neighbors of each vertex set to the corresponding partition. We refer to the vertex set assigned by METIS to a partition as *core nodes*, and the vertex set of 1-hop neighbors that do not intersect with core nodes as *halo nodes*. Halo nodes can be viewed as nodes situated at the periphery of the partition, where each 1-hop halo node has at least one connection to the core nodes. The higher the hop value for halo nodes, the lower the communication requirements and the higher the amount of stored data. Our caching scheme for 1-hop halo nodes ensures that each partition can fulfill any requests related to its core nodes, while only adding a moderate amount of memory overhead.

3.2.2 Graph Shard Data Structure. After partitioning the graph, we convert each partition to the Compressed Sparse Row (CSR) format. This storage format is frequently used for large-scale graphs due to its memory and cache efficiency. We represent rows with core nodes and columns with the union of core nodes and halo nodes. For each core node, we store neighbor information including endpoint index (indptr), local vertex IDs of neighboring nodes, shard IDs of neighboring nodes, edge weights, and weighted degrees of neighboring nodes. We refer to this converted data structure as the *Graph Shard*. Using this format, we efficiently store the graph

Single Source Personalized PageRank (w. overlap)

```
g = DistGraphStorage(rrefs, SHARD_ID)
m = SSPPR(source_node_id, SHARD_ID, ALPHA, EPSILON)

while True:
  node_ids, shard_ids = m.pop()
  if len(node_ids) == 0: break

  mask_dict = {j: shard_ids == j for
               j in range(NUM_SHARDS)}

  futs = {}
  for j, mask in mask_dict.items():
    if j == SHARD_ID: continue
    futs[j] = g.get_neighbor_infos(j,
                                   node_ids[mask])
  mask = mask_dict[SHARD_ID]
  infos = g.get_neighbor_infos(SHARD_ID,
                               node_ids[mask])
  m.push(infos, node_ids[mask], shard_ids[mask])
  for j, mask in mask_dict.items():
    infos = futs[j].wait()
    m.push(infos, node_ids[mask], shard_ids[mask])
```

Random Walk

```
g = DistGraphStorage(rrefs, SHARD_ID)

node_ids = # root nodes local IDs
shard_ids = torch.empty(NUM_ROOTS)
walks_summary = torch.empty(NUM_ROOTS, WALK_LEN)

mask_dict = {SHARD_ID: torch.arange(NUM_ROOTS)}

for i in range(WALK_LEN):
  futs = {}
  for j, mask in mask_dict.items():
    futs[j] = g.sample_one_neighbor(j,
                                    node_ids[mask])

  for j, index in mask_dict.items():
    local_nids, global_nids, sids = futs[j].wait()
    node_ids[mask] = local_nids
    shard_ids[mask] = sids
    walks_summary[mask, i] = global_nid

  for j in range(NUM_SHARDS):
     mask_dict[j] = shard_ids == j
```

Figure 4: The Python implementation of distributed Single Source Personalized PageRank and distributed Random Walk, using the abstraction of the proposed graph engine and PyTorch. The variable *rrefs* denotes a list of PyTorch Remote Reference objects. The function names in red indicate the interface supported by Distributed Graph Storage, and PPR Ops are highlighted in blue.

data using contiguous shared memory arrays containing neighbor information for the core nodes. A Graph Shard consists of the following arrays:

- indptr (index pointer): This array stores the endpoint indices for each node's neighboring data, corresponding to non-zero values in the adjacency matrix.
- local IDs: This array stores the local ids of the neighboring nodes for each core node, facilitating efficient access.
- shard IDs: This array stores the shard ids of the neighboring nodes for each core node, indicating the shard to which the neighboring nodes belong.
- edge weights: This array stores the edge weights between the core node and its neighboring nodes, allowing for efficient retrieval of edge weight information.
- weighted degrees: This array stores the weighted degree values for the neighboring nodes of each core node, representing the sum of the edge weights for the outgoing edges incident to that node.

The Graph Shard data structure offers several benefits. First, a local ID and a shard ID can be used to represent an arbitrary node without converting them to a global ID. When traversing a graph, shard IDs can be used to dispatch tasks to target machines, and local IDs can be directly used as indices for core nodes. Second, the weighted degrees are useful for threshold checking in Forward Push. If the residual value of a node is greater than the threshold and the node is not already in the activated set, it can be added to the activated set for the next iteration. Storing weighted degrees in each shard eliminates the need to aggregate edge weights on the fly, which can be slow and resource-intensive, especially when dealing with cross-machine operations.

Overall, the utilization of the CSR format in Graph Shards enables efficient storage and retrieval of graph data, particularly for large-scale graphs with sparse connectivity patterns. Figure 2 visualizes an example of a 2-partitioned graph represented in our Graph Shard format.

Our graph engine provides a comprehensive set of APIs/primitives for general graph operations, allowing users to efficiently access and manipulate graph data. The vertex property (VertexProp) object is a critical logical component of the proposed approach, as it serves as a container for essential neighbor information pertaining to the core node within a shard. This information is essential for efficient graph traversals, as it allows the engine to quickly identify and access the relevant vertices. Specifically, these objects encompass pointers to local ids, shard ids, edge weights, and weighted degrees arrays of the corresponding graph shard, along with the start and end indices that delineate the neighborhood of each node. The arrangement of these vertex property components in Shard 0 is shown in Figure 3. Moreover, our engine includes several methods for retrieving critical statistics about the graph. This API can be easily extended to incorporate additional functions as needed.

3.2.3 RPC requests Optimization. Another noteworthy contribution of our research lies in the establishment of efficient communication mechanisms between shards to enable distributed computation of PPR values. The vanilla Forward Push algorithm (Algorithm 1) sequentially processes each activated vertex and updates along its out-neighbors. If we directly implement this sequential version based on our distributed Graph Storage, many RPC requests will be issued across machines, and each data package size will be proportional to the source node's out-degree. Most real-life graphs follow the power law, implying that the majority of nodes in the graph

have a very small set of neighbors. However, the TensorPipe backend for PyTorch RPC is designed for transferring large tensors with relatively low frequency. It can chunk and multiplex large tensors over multiple sockets to achieve very high bandwidths. Therefore, issuing a large number of RPC requests with small data packages is inefficient.

We propose three optimization techniques for tensor-based RPC communication to address the aforementioned problems. (1) We batch all the RPC requests according to the destination shard IDs in each iteration. (2) Before transferring the neighbor information for a batch of source nodes, we compress it into the CSR format and wrap each array in a tensor. (3) We overlap local operations with remote calls.

For better RPC request batching, we adopt the parallel Forward Push algorithm [22] as our single machine base algorithm. In each iteration, the parallel version processes the activated vertices and updates along their out-neighbors, which are processed in parallel. Although the parallel version requires slightly more "pushes" than the sequential version, the parallel Forward Push is naturally suitable for request batching since there are no dependencies within a set of activated vertices. As discussed in the previous section, each node is represented by a local ID and a shard ID. We can use tensor operations to efficiently construct a local ID mask for each shard. Then, each process requests a batch of neighbor information from each Graph Storage based on the filtered local ID array. This ensures that each process issues at most one request to another machine in each iteration, thereby minimizing communication overhead.

Given that we store the graph data in a vertex-centric way, the response is wrapped in a list-of-list format, returning a vector of neighbor information (wrapped in tensors) for a batch of source nodes. Note that nodes with extremely large degrees, also known as super-nodes, can generally impact the vertex-centric representation. However, in the context of GNNs, super-nodes are not an issue, since the degree of each node is usually limited during preprocessing. For local requests, the neighbor information is also wrapped in the same way and then returned to the Python layer without data copy. However, tensor wrapping dominates the local fetch time, and transferring a list of small tensors with non-equal lengths through Torch RPC is inefficient. To optimize this, we directly pass a vector of shared pointers of VertexProp across the C++ and Python layers for local fetching, without taking ownership of the original data. To perform a remote fetch, we first convert the requested neighbor information to CSR format and then wrap each array in a tensor. Since the CSR format response has the same structure as the Graph Shard data, we can easily use the VertexProp API to extract the response.

We further overlap local operations with remote calls, effectively leveraging parallelism and reducing overall runtime. Note that we use pybind11 to wrap the C++ Graph Storage and PPR operators into Python objects. Overlapping RPC target functions with local Python functions can cause a Global Interpreter Lock (GIL) contention issue, drastically increasing remote fetch time. To address this problem, we explicitly release the GIL in our C++ functions and separate the Graph Storage server process from the PPR computing process.

3.3 Local PPR Operators

In the following section, we present the local PPR operators that are optimized specifically for calculating approximated SSPPR queries, following the parallel Forward Push algorithm [22] outlined above. According to the algorithm, when the residual value of a node exceeds the computed threshold, the PPR value for that node and its neighboring nodes is updated. Here we discuss two main operators exposed to the Python layer. (1) The *pop* operator first returns the local ID tensor and the shard ID tensor from the current activated vertex set and then clears the set. (2) The *push* operator updates the PPR and residual values based on a given batch of source nodes and their neighbor information. For each SSPPR query computation, the main loop is written in Python. In each iteration, the computing process first retrieves the activated nodes from the *pop* operator, then fetches neighbor information from local and remote Graph Storage, and finally invokes the *push* operator to update the PPR and residual values of the vertices touched in this iteration.

To further optimize performance, we leverage the inherent parallelism of an open-source parallel hash map[2] in our implementation of operators. Parallel-hashmap stores key-value pairs in a hash table that is partitioned into multiple segments/submaps. The key is a <local:id, shard:id> pair, and the value is either the PPR value or the residual value. Each segment is protected by a lock to allow for concurrent access in a multi-threaded environment. To ensure thread safety in a parallel environment, we eliminate the need for locks by assigning computationally expensive map update operations to each thread based on the index of the submap. Here, we applied a simple strategy to determine whether to use multi-threading for *push* operation. If the batch size of source nodes surpasses a certain threshold, we distribute the workload among multiple threads using the OpenMP parallel directive for improved performance. Otherwise, only a single thread is used.

4 EXPERIMENTS

In this section, we experimentally evaluate our proposed method on several large-scale real-world graphs. We begin by comparing our method with pure tensor-based implementations and present the scalability analysis. Then, we present the overall runtime breakdown and examine each design's effectiveness. Finally, we demonstrate a straightforward example of integrating our proposed PPR engine with distributed GNN training.

4.1 Experimental Settings

Setup. We implement our graph engine using Python 3.9.13 and PyTorch 1.13.1. The Graph Storage and PPR Ops are implemented in C++14 and bonded to Python using Pybind11 [12]. We simulate the distributed setting using a machine with dual AMD EPYC 7763 CPU (128 cores 256 threads with Simultaneous Multi-Threading turned on) and 1TB ECC-DDR4 memory. On distributed clusters with fast interconnections (i.e., 100Gbps Ethernet or InfiniBand), we expect the time cost of remote communication to be similar to local communication through shared memory, and memory operations to be faster due to more independent memory channels. Note that we spawn $K \times (P + 1)$ processes to simulate a K-machines scenario, where P is the number of SSPPR computing processes.

[2] https://github.com/greg7mdp/parallel-hashmap

Table 1: Datasets

Name	$\|V\|$	$\|E\|$	d_{avg}	d_{max}
Ogbn-products	2.5M	120M	50.5	17,481
Twitter	41.7M	2.4B	57.7	2,997,487
Friendster	65.6M	3.6B	57.8	5,214
Ogbn-papers100M	111M	3.2B	29.1	251,471

Each simulated machine is also assigned an additional process as the Graph Storage server. Processes within a simulated machine can directly access the local graph shard from the shared memory, while accesses across simulated machines are routed through RPC requests.

Following the definition of approximated whole-graph SSPPR from existing works [18, 25], where an SSPPR query of source node s returns an estimated PPR $\pi(s, v)$ for each node $v \in V$ with a value greater than zero, we collect the throughput (number of computed SSPPR queries) and runtime in an average of 10 repeated runs after 4 warm-up runs. We fix the value of the teleport parameter to $\alpha = 0.462$ and residue threshold to $\varepsilon = 10^{-6}$ for all experiments.

Datasets and query sets. To better evaluate our proposed method, we run experiments on 4 public large-scale datasets: Ogbn-products, Twitter, Friendster, and Ogbn-papers100M. Among them, Ogbn-products and Ogbn-papers100M are from the OGB [11] datasets, which are widely used in graph machine learning research. We remove node features in the original OGB datasets and only keep the graph topology. Twitter and Friendster are social networks obtained from SNAP collection[3] and are intensively used to evaluate PPR query efficiency. All these datasets are converted to undirected graphs with randomly generated edge weights. Table 1 summarizes the statistics of the datasets.

Graph Shard Preprocessing. To achieve efficient distributed PPR computing, we partition the entire graph of each dataset into several graph shards and further preprocess each shard according to Section 3. Let vertex IDs and edge weights have data types with the same size (i.e., int32 and float32), the memory consumption of each preprocessed graph shard will increase by around 1.5 times. This is due to the additional storage of $|E|$ elements of the weighted degrees for each node v where $(u, v) \in E$. For example, the Ogbn-papers100M dataset with edge weights has a size of around 28GB. After partitioning the graph into 4 parts using Metis and preprocessing each part, the largest graph shard has a size of around 14GB.

4.2 Comparison with tensor-based Implementation

In this section, we aim to examine the overall performance of our proposed PPR Engine against tensor-based PPR methods.

Different from existing graph-processing libraries, tensor-based methods use tensors as the foundational data structure to implement complex graph algorithms, which offer the best flexibility and utility for integration with deep learning methods. For each whole-graph SSPPR query, we create a 1-D tensor x with the length of $|V|$ to store the PPR query regarding the source node. The index of x corresponds to the vertex global ID, and each element stores the PPR value of the node corresponding to the index. Following this setup, we can efficiently update a subset of PPR values in parallel by just using the tensor operations. Here we include two tensor-based implementations as our baseline models. The first is a single-machine implementation of the Power Iteration algorithm, which relies on the Sparse Matrix-Matrix multiplication library in DGL[24]. We refer to this implementation as *DGL SpMM*. The second implementation, denoted as *PyTorch Tensor*, is a distributed version of the parallel Forward Push algorithm [22], built using only PyTorch operators. We refer to our proposed method as *PPR Engine*.

[3]http://snap.stanford.edu/data/index.html

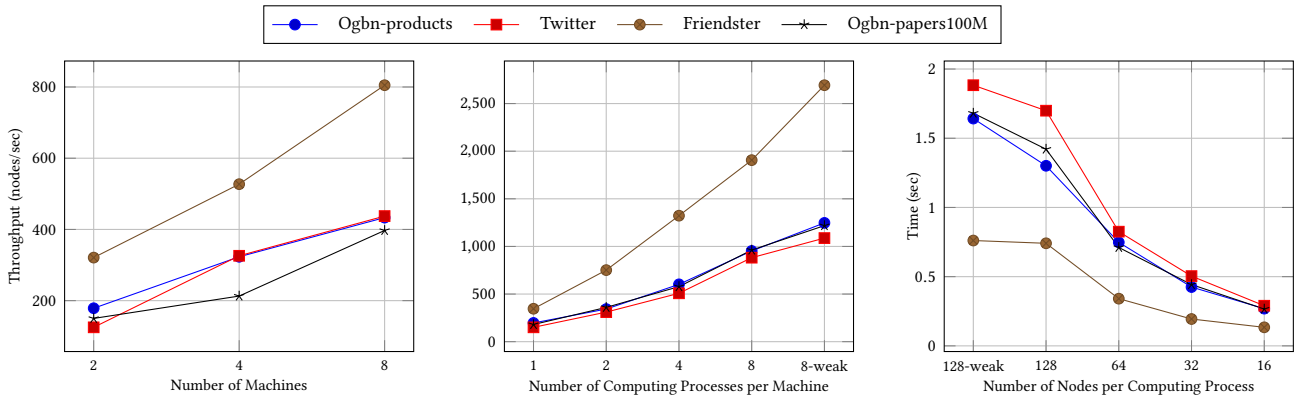
Table 2: Throughput (number of queries per second) of implementations under the 4-machine scenario. For Forward Push based implementations, the teleport probability $\alpha = 0.462$ and the residue threshold $\varepsilon = 10^{-6}$.

Data Struct.	Tensor		HashMap
Algo.	Power Iteration	Forward Push	
Impl. Name	DGL SpMM	PyTorch Tensor	PPR Engine
Ogbn-products	1.676	11.92	981.7
Twitter	0.364	2.617	905.2
Friendster	0.236	1.202	1304.1
Ogbn-papers	0.148	0.879	726.1

Table 2 shows the throughput for different implementations on each of the datasets. We simulate a 4-machine scenario and set the number of compute processes on each machine to 3. For the single-machine version of Power Iteration (*PyG SpMM*), we simply times the throughput of a single machine by a factor of 4 to obtain the results of the ideal case.

Unlike Forward Push, Power Iteration is a High-Precision SSPPR method [27] that aims to compute an estimation $\hat{\pi}(s, v)$ of $\pi(s, v)$ such that $|\hat{\pi}(s, v) - \pi(s, v)| \leq \epsilon$. Typically, ϵ is set to a very small value (e.g., 10^{-10} in our experiment), and the results obtained using Power Iteration are considered as ground truth. When comparing two tensor-based methods, we observed that the Forward Push implementation is up to 7.2× faster than the ideally distributed Power Iteration, while achieving 97%+ accuracy of the top-100 results with the residual threshold $\varepsilon = 10^{-6}$. This is because Forward Push can terminate most of the propagation steps, significantly reducing the number of push operations. For downstream GNN tasks, approximated SSPPR methods show comparable accuracy to exact methods [2], even with a residual threshold $\varepsilon = 10^{-4}$, but with significantly less overhead.

For Forward Push implementations, our PPR engine is 83 − 1085× faster than the tensor-based solution. The main difference between these two solutions is that the *PPR engine* uses C++ level Graph Storage and HashMap-based PPR operators. The main drawback of the tensor-based solution is that the overhead of SSPPR calculation increases in proportion to the total number of nodes. In contrast, the HashMap-based method's running time is more related to the local pattern of source nodes, rather than the input graph size.



(a) Scalability Analysis

(b) Inter-SSPPR Parallelization Analysis

Figure 5: Scalability analysis: varying the numbers of machines and processes.

4.3 Scalability Analysis

This section aims to examine the scalability of our proposed PPR engine by varying the number of machines and computing processes.

Figure 5 shows the throughput of processed SSPPR queries per second on each dataset. We fixed the problem size at 256 for SSPPR queries and set the number of graph partitions to be equivalent to the number of used machines. For each machine, we spawn only one process for SSPPR computing. When increasing the number of machines from 2 to 8, we observe a speedup of 2.5 – 3.5×. Note that increasing the number of machines also increases the number of partitions and reduces the size of each shard. In general, as the number of machines increases, a larger portion of graph traversal will take place outside of the local graph shard (e.g., from 3% to 13% for 2 to 8 partitions on Ogbn-products). This demonstrates the efficiency of our Remote Graph Storage for large-scale graph computation in the distributed setting. Additionally, an appropriate number of graph partitions can effectively reduce the number of edge cuts and thereby reduce communication overhead, which is crucial to the overall throughput. For instance, when increasing the number of machines from 2 to 4 on the Twitter dataset, the ratio of remote graph traversal decreased from 55% to 50%, resulting in a super-linear speedup of 2.6×.

Next, we examine the scalability of our method's inter-SSPPR parallelism. We fix the number of machines to 2 and vary the number of computing processes per machine from 1 to 8. We evaluate two types of scalability: (1) For strong scaling, we fix the total problem size (number of queries) to 128 and alter the number of processes. (2) For weak scaling, we fix the number of assigned queries to 128 for each process, which aligns with the GNN sampling setting. As shown in Figure 5, we use 8 processes to achieve a 4.8 – 5.5× speedup compared to a single process for the strong scaling setup, and a 6.4 – 7.8× speedup for the weak scaling setup. We see that our implementation scales well (almost linearly for weak scaling) with the number of processes. The relatively worse performance of the strong scaling setup can be attributed to the problem of workload imbalance, which becomes more severe when the number of queries per process is small.

Table 3: The ablation study of RPC optimizations on Friendster.

	Local Fetch (s)	Remote Fetch (s)	Push (s)	Total (s)	Speedup
Single	0.38	6.59	0.87	7.85	—
+Batch	0.16	0.80	0.15	1.11	7.1×
+Compress	0.03	0.13	0.15	0.30	26.2×
+Overlap	0.04	0.22	0.15	0.22	35.7×

4.4 Effects of optimizations

In this section, we evaluate the effectiveness of each proposed optimization technique.

Figure 6 shows the breakdown of running time for the *PPR Engine* and *PyTorch Tensor*, respectively. Both methods batch the RPC requests and avoid overlapping local operations with remote calls for a better breakdown of runtime. To facilitate comparison, we

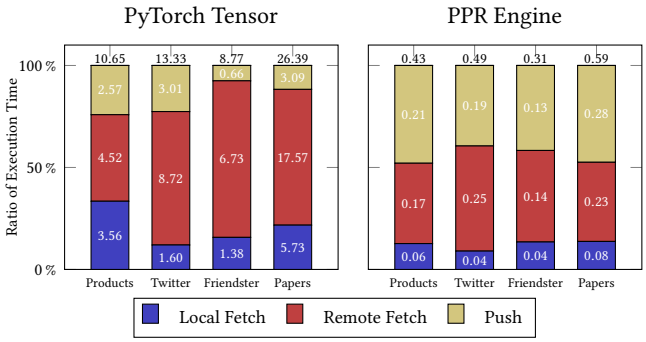


Figure 6: The runtime breakdown for *PyTorch Tensor* and *PPR Engine* on different datasets.

Distributed GNN Tranining with Personalized PageRank

```
g         = DistGraphStorage(rrefs, rank)

model     = GraphSAGE(NUM_INPUT, NUM_OUTPUT).to(rank)
model     = DistributedDataParallel(model,
                    device_ids=[rank])
optimizr = torch.optim.Adam(model.parameters())
loader    = DataLoader(torch.arange(NUM_NODES[rank]),
                    batch_size=BATCH_SIZE//WORLD_SIZE)

for epoch in range(NUM_EPOCHES):
    model.train()
    for batch_index in loader:
        ppr   = forward_push(g, batch_index)
        batch = convert_batch(g, ppr, batch_index)
        batch = batch.to(rank)
        optimizer.zero_grad()
        out   = model(batch.x, batch.edge_index)
        loss  = F.cross_entropy(out[batch.ego_idx],
                                batch.y)
        loss.backward()
        optimizer.step()

    torch.distributed.barrier()
```

Figure 7: The variable *rrefs* represents a list of PyTorch Remote References of Graph Storages. Functions for PPR computation are highlighted in red.

omit the time taken for activated node retrieval in both figures. For *PyTorch Tensor*, retrieving activated nodes requires scanning the entire PPR tensor, making the time cost dominant and proportional to $|V|$. On the other hand, for *PPR Engine*, retrieving activated nodes refers to the *pop* operation, and the time cost is negligible since the activated nodes equal the keys of a pre-stored node set. We see that the Remote Fetch time is similar to the Push time for our *PPR Engine*, and the Remote Fetch time is dominant for *PyTorch Tensor*. This indicates that the remote RPC requests are well optimized in our *PPR Engine*. Additionally, our proposed HashMap-based *push* operation is $5-16\times$ faster than the tensor-based push.

We further study the efficacy of the RPC request optimization techniques discussed in Section 3.2.3. For the baseline model, we do not apply any RPC optimization techniques and process only one vertex at a time. We denote the baseline model as *Single*. The RPC batching technique is employed on the baseline version, which is referred to as *Batch*. We then compress the neighbor information transferred in the *Batch* version and denote it as *Compress*. Finally, we overlap remote fetching with local operations based on *Compress* and name this version as *Overlap*. To ensure fairness, we use C++ Graph Storage and PPR Ops for all the competitors. As shown in Table 3, each proposed technique is able to effectively reduce the overall runtime. We observe that batching RPC requests can effectively speed up all operations. This is because batch parallelism is well utilized and the overhead of issuing RPC requests is significantly reduced. The neighbor information compression technique can further reduce local and remote fetching time by around 80%. Lastly, we can achieve an additional 1.3x speedup by overlapping remote fetching with local fetching and push operation.

4.5 Case Study: Training GNNs with Personalized PageRank

To illustrate the ease of integration for the proposed Graph Engine with distributed GNN training, we provide a simple yet intact example in this section. Figure 7 shows the code snippet that uses PPR to construct subgraphs for distributed mini-batch training of ShaDow-SAGE [33]. For simplicity, we use one GPU device per machine and store a different Graph Shard in the memory of each machine. We use PyTorch's *DistributedDataParallel* for distributed gradient synchronization and PyG for GNN model implementation. For each batch, we calculate a list of top-K SSPPR vectors on-the-fly using our proposed PPR engine. We then convert these vectors to PyG's data format and feed them to the GNN models located on GPUs. Note that a *convert_batch* function is all we need to integrate our PPR engine with existing GNN libraries by following ShaDow's design principle [33]. This function induces a subgraph from a set of vertices with top-K PPR values and slices corresponding features from a cross-machine feature store. These data can then be easily transformed into different formats suitable for various deep learning frameworks for graphs.

5 CONCLUSION

We presented a distributed graph engine for efficient SSPPR computation. Our proposed graph engine incorporates generalized graph storage, an optimized Forward Push algorithm for computing SSPPR, and efficient batching techniques to minimize PyTorch RPC communication overhead. We have shown through extensive experiments that our map-based solution achieves significant improvements in SSPPR computation speed, with up to three orders of magnitude throughput improvement over traditional tensor-based approaches. Our results demonstrate that our approach can scale well on large graphs and achieve high performance in distributed GNN training. We believe that our approach has broad applications in real-world graph mining tasks and will be a valuable resource for GNN researchers and practitioners.

ACKNOWLEDGMENTS

This work is supported by Meta Platforms Inc. under grant number INB2675366, National Science Foundation (NSF) under grant OAC-2209563, and DEVCOM Army Research Lab (ARL) under grant W911NF2220159.

REFERENCES

[1] 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.

[2] Aleksandar Bojchevski, Johannes Gasteiger, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. 2020. Scaling graph neural networks with approximate pagerank. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2464–2473.

[3] Jie Chen, Xing Xie, Shuo Zhou, Jiaxin Huang, Bin Qin, and Wenwu Zhu. 2018. Scalable Personalized PageRank Estimation via Graph Partitioning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 1334–1343.

[4] Ming Chen, Zhewei Wei, Bolin Ding, Yaliang Li, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. 2020. Scalable graph neural networks via bidirectional propagation. *Advances in neural information processing systems* 33 (2020), 14556–14566.

[5] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[6] Francois Fouss, Alain Pirotte, Jean-Michel Renders, and Marco Saerens. 2007. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. In *Proceedings of the Sixth International Conference on Data Mining*. IEEE, 233–242.

[7] Johannes Gasteiger, Aleksandar Bojchevski, and Stephan Günnemann. 2018. Predict then propagate: Graph neural networks meet personalized pagerank. *arXiv preprint arXiv:1810.05997* (2018).

[8] Johannes Gasteiger, Stefan Weißenberger, and Stephan Günnemann. 2019. Diffusion improves graph learning. *Advances in neural information processing systems* 32 (2019).

[9] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 599–613.

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

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

[12] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2017. pybind11 – Seamless operability between C++11 and Python. https://github.com/pybind/pybind11.

[13] Glen Jeh and Jennifer Widom. 2003. Scaling Personalized Web Search. In *Proceedings of the 12th International Conference on World Wide Web* (Budapest, Hungary) *(WWW '03)*. Association for Computing Machinery, New York, NY, USA, 271–279. https://doi.org/10.1145/775152.775191

[14] Glen Jeh and Jennifer Widom. 2003. Scaling personalized web search. In *Proceedings of the 12th international conference on World Wide Web*. 271–279.

[15] Sepandar D Kamvar, Taher H Haveliwala, and Gene H Golub. 2003. Exploiting the block structure of the web for computing pagerank. In *Proceedings of the 12th international conference on World Wide Web*. ACM, 405–412.

[16] Sepandar D Kamvar, Matthew T Schlosser, and Hector Garcia-Molina. 2003. Eigentrust: A trust-based reputation system for peer-to-peer networks. In *International Conference on Peer-to-Peer Computing*. IEEE.

[17] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

[18] Peter A Lofgren, Siddhartha Banerjee, Ashish Goel, and Comandur Seshadhri. 2014. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1436–1445.

[19] Michael W Mahoney. 2011. Randomized algorithms for matrices and data. In *Proceedings of the IEEE*, Vol. 99. IEEE, 1588–1601.

[20] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[21] Hongbin Pei, Bingzhe Wei, Kevin Chen-Chuan Chang, Yu Lei, and Bo Yang. 2020. Geom-gcn: Geometric graph convolutional networks. *arXiv preprint arXiv:2002.05287* (2020).

[22] Julian Shun, Farbod Roosta-Khorasani, Kimon Fountoulakis, and Michael W Mahoney. 2016. Parallel local graph clustering. *arXiv preprint arXiv:1604.07515* (2016).

[23] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. 2006. Fast random walk with restart and its applications. In *Sixth international conference on data mining (ICDM'06)*. IEEE, 613–622.

[24] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*.

[25] Sibo Wang, Renchi Yang, Xiaokui Xiao, Zhewei Wei, and Yin Yang. 2017. FORA: simple and effective approximate single-source personalized pagerank. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 505–514.

[26] Felix Wu, Tianyi Zhang, Alejandro L Souza Jr, Chris Fifty, Tao Yu, and Kilian Q Weinberger. 2019. Simplifying graph convolutional networks. In *Proceedings of the 36th International Conference on Machine Learning*. 6861–6871.

[27] 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*. 1996–2008.

[28] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. *First International Workshop on Graph Data Management Experiences and Systems* 1, 1 (2013), 2.

[29] 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*. 974–983.

[30] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L Hamilton, and Jure Leskovec. 2018. Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems* 31 (2018), 4800–4810.

[31] Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor Prasanna, Long Jin, and Ren Chen. 2021. Decoupling the depth and scope of graph neural networks. *Advances in Neural Information Processing Systems* 34 (2021), 19665–19679.

[32] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2019. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931* (2019).

[33] Yuan Zhang and William W Cohen. 2020. Shadow: scalable personalized PageRank estimation for large social graphs. In *Proceedings of the 13th ACM International Conference on Web Search and Data Mining*. ACM, 279–287.