SG-Serve: Efficient Model Serving for Subgraph-based Graph Representation Learning

QIHUI ZHOU, The Chinese University of Hong Kong, China
PEIQI YIN, The Chinese University of Hong Kong, China
XIAO YAN*, Institute for Math and AI, Wuhan University, China
CHANGJI LI, The Chinese University of Hong Kong, China
JAMES CHENG, The Chinese University of Hong Kong, China

Subgraph-based graph representation learning (SGRL) is an emerging class of GNN models that achieve much higher accuracy for various tasks than classical GNN models (e.g., GCN and GAT). However, we observe that serving SGRL models for online applications is challenging due to their *irregular request workloads*. Specifically, some heavy requests need significantly more computation than regular requests (e.g., 100x), leading to excessively long tail latency in existing systems. As such, we build SG-Serve, which tailors *subgraph extraction* and *model inference* (i.e., the two main stages of SGRL models) to handle the irregular request workloads. For subgraph extraction on the CPU, we propose a general API to implement the diverse extraction methods of SGRL models. Beside generality, the API also exposes parallelization opportunities and allows the heavy requests to utilize multiple threads for speedup. To schedule the CPU threads to conduct extraction for concurrent requests, we design a work-stealing policy, which enjoys parallelism while avoiding head-of-line blocking. For model inference on the GPU, we batch the requests according to their workload instead of request count (i.e., as in existing systems) and run two GPU processes to prevent the heavy requests from monopolizing the GPU. Our experiments show that compared with existing systems, SG-Serve can reduce the 99th percentile (P99) latency by over 13x and improve the request throughput by over 33x.

CCS Concepts: • **Computing methodologies** → *Parallel algorithms*; *Machine learning algorithms*; • **Information systems** → *Data management systems*.

Additional Key Words and Phrases: Graph learning, query processing

ACM Reference Format:
Qihui Zhou, Peiqi Yin, Xiao Yan, Changji Li, and James Cheng. 2026. SG-Serve: Efficient Model Serving for Subgraph-based Graph Representation Learning. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 83 (February 2026), 29 pages. https://doi.org/10.1145/3786697

1 Introduction

Graph neural networks (GNNs) achieve high accuracy for many graph tasks, such as node classification [46, 66] and link prediction [2, 7]. Recently, a new class of GNN models called *subgraph-based graph representation learning* (SGRL) is becoming popular. Classical GNNs (e.g., GCN [46], SAGE [38], and GAT [41]) compute an embedding for each node in the data graph and use these embeddings

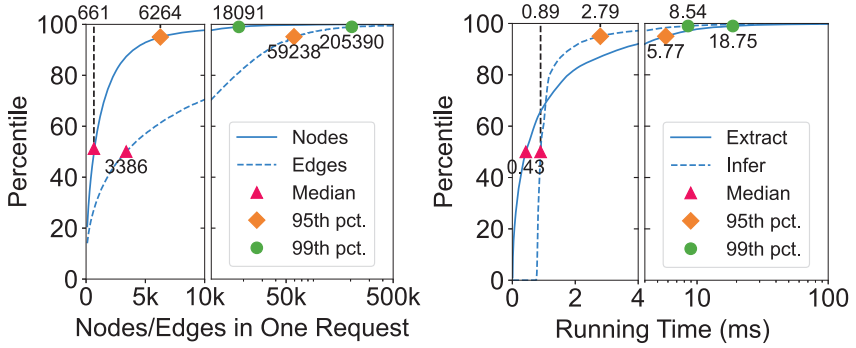
*Dr. Xiao Yan is the corresponding author.

Authors' Contact Information: Qihui Zhou, qhzhou@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China; Peiqi Yin, pqyin22@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China; Xiao Yan, yanxiaosunny@whu.edu.cn, Institute for Math and AI, Wuhan University, Wuhan, China; Changji Li, cjli@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China; James Cheng, jcheng@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, China.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/2-ART83
https://doi.org/10.1145/3786697

Table 1. Accuracy of classical GNN models (e.g., GCN, SAGE, and GAT) and SGRL models for some representative graph tasks (results obtained from the referenced papers).

Task	Dataset	Model			
		GCN	SAGE	GAT	SGRLs
Link Prediction [54]	Coro [67]	0.86	0.85	0.88	0.91
Relation Prediction [84]	Collab [39]	0.44	0.54	-	0.64
High-order Pattern [83]	MAG [39]	0.57	0.61	-	0.82
Subgraph Prediction [11]	PPI-BP [80]	0.39	-	0.31	0.59



(a) CDF of subgraph size (b) CDF of running time

Fig. 1. The extracted subgraph sizes and running time distributions of SGRL requests. We use the SEAL [92] model and the Papers100M graph from the Open Graph Benchmark [39] but the observations generalize. The running time is obtained by *executing each request individually*.

for downstream tasks. In contrast, SGRL models involve two sequential stages, namely, *subgraph extraction* and *model inference*: a subgraph is first extracted from the underlying data graph by jointly considering the given target nodes; and then output is computed by running a model (e.g., classical GNNs) on the extracted subgraph. It has been proven in theory that SGRLs are more expressive than classical GNNs due to the subgraph extraction process [10, 52, 84, 93]. Table 1 shows that SGRLs achieve substantially higher accuracy across various graph tasks, which underpin numerous online applications including personalized recommendation [50, 89] and fraud detection [69, 103]. Due to their good accuracy, many researches design SGRL models with different subgraph extraction methods and model architectures [13, 32, 48, 52, 60, 71, 90, 92, 102].

Despite the efforts in designing powerful SGRL models, efficient online serving for SGRL models has been less explored. In particular, to utilize a trained SGRL model in real-world applications, the inference procedure is deployed as a dedicated service, which receives user requests consisting of target node sets in real time, computes model outputs for the requests, and returns the results back to users. For example, when applying SGRL models to commodity recommendation [50, 89], each user request may contain two nodes representing a consumer and a product. The deployed SGRL model predicts the likelihood of a link between these two nodes to determine whether to issue a recommendation. Similarly, when applying SGRL models to fraud detection [69, 103], each request may include a node representing a suspicious bank account. The deployed SGRL model estimates the probability that this node is fraudulent and triggers blocking actions when necessary. To effectively support such online services, the primary goal of SGRL model serving is to achieve low latency, high throughput, and short tail latency (e.g., the 99th percentile, P99), which is critical for ensuring high quality of service (QoS) [12, 23, 27, 28, 45]. Some systems (e.g.,

DGI [85] and GCNP [96]) consider model serving for classical GNN models and compute the node embeddings offline for online use. This methodology does not work for SGRL because subgraph extraction processes a set of target nodes, and the large number of possible target node sets makes pre-computation infeasible.

Serving SGRL models is more challenging than serving regular machine learning (ML) models such as multilayer perceptrons (MLPs) and convolution neural networks (CNNs). To be specific, the computation process of a request is fixed for regular ML models, resulting in the same amount of workload for each request. In contrast, the requests of SGRL models exhibit highly irregular workloads with a long tail. As shown in Figure 1(a), considering the number of nodes and edges in the extracted subgraphs, the request at the 99th percentile is 27.4x and 60.7x of the median, respectively. The skewed request size translates into heavily tailed running time for both the subgraph extraction stage and the model inference stage as shown in Figure 1(b). Such an irregular request workload is attributed to the subgraph extraction process and thus inherent for SGRLs. In particular, node degrees usually follow a power-law distribution for real-world graphs [19, 36], which means that some nodes have many more neighbors than average, and a request will be heavy if it involves the high-degree nodes during subgraph extraction. Unfortunately, existing SGRL systems [1, 4, 5] overlook the irregular workload when dealing with requests, leading to the following three sub-optimal design choices.

Synchronized batch processing (SBP). Built upon open-source machine learning frameworks [30, 62, 72], existing SGRL systems adopt the SBP protocol for request processing, which is widely used to improve system throughput and GPU utilization when serving regular ML models [6, 24, 61]. Specifically, SBP accepts a batch of requests each time and returns the inference results for the entire batch after processing all requests in the batch. With SBP, heavy requests become batch stragglers due to their long running time and prolong the response time of other requests in the same batch.

Single-thread subgraph extraction. Existing systems conduct subgraph extraction on the CPU to host large data graphs and express the extraction procedure with one user-defined function (UDF) [1, 4, 5]. The UDF is invoked and executed by a single thread for the subgraph extraction of each request. Despite flexibility, single-thread extraction causes slow extraction processes for heavy requests and low CPU utilization due to limited parallelism.

Sequential request processing. Existing systems group the requests into batches with a predefined request count using the first-come-first-served (FCFS) strategy and process these batches sequentially. As heavy requests take substantially longer processing time than normal requests, they block subsequent requests from being processed for a long time, leading to high queuing delays.

In this paper, we propose SG-Serve, a general and efficient serving system for SGRL models. Instead of perceiving subgraph extraction and model inference as an entire process in SBP, SG-Serve connects the two stages using a *producer-consumer* mechanism, which allows the requests that finish extracting subgraphs to enter the model inference stage without waiting for the other requests. For subgraph extraction, we summarize the subgraph extraction methods of existing SGRL models into a general 5-step procedure and design a succinct API for each step. The API exposes parallelization opportunities by modeling the subgraph extraction for a request as a directed acyclic graph (DAG) of fine-grained tasks that can run in parallel. To schedule these tasks, we design a strategy based on work-stealing that not only allows the requests to use extra parallelism (i.e., run with more than one thread) but also prevents the heavy requests from saturating all the threads.

For model inference, instead of using request counts, we batch the requests according to their workload. Specifically, we set limits on the total number of nodes and edges in a batch, and

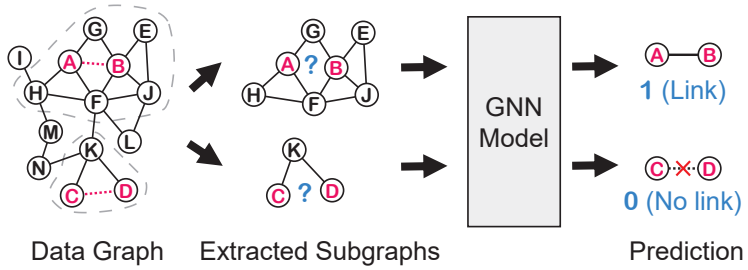


Fig. 2. An illustration of SGRL. The seed nodes are $\{A, B\}$ and $\{C, D\}$ for two requests, respectively, and subgraph extraction retrieves common 1-hop neighbors.

subgraphs are added to a batch in an FCFS order until reaching the limits. We also leverage the GPU multi-process service (MPS) to create two processes for inference and allow at most one process to handle heavy requests that exceed the batch limits. The *workload-aware batching* separates the processing of heavy requests from other requests, and the *dual-process dispatching* prevents heavy requests from blocking the GPU. To configure the batch size, we conduct dynamic tuning to minimize the request latency.

We conduct extensive experiments to evaluate SG-Serve with 3 SGRL models and 3 graphs with different characteristics. The results show that SG-Serve consistently outperforms existing systems by a large margin. In particular, under the same request throughput, SG-Serve can reduce the 50th and 99th percentile request latency by up to 19.3x and 13.1x, respectively. Under the same request latency, SG-Serve achieves a higher throughput by up to 33.3x. The experiment results also suggest that the designs of SG-Serve are effective. Specifically, with our API and execution model, the P99 latency of subgraph extraction is reduced by up to 5.17x compared to single-thread execution. With our batching and dispatching strategies, the P99 latency of inference is reduced by up to 6.03x compared with batching according to request counts.

2 Background and Motivation

Here, we introduce the basics of subgraph-based graph representation learning (SGRL) to provide background and analyze the problems of existing SGRL serving solutions to motivate our designs.

2.1 SGRL Models

SGRL models usually consider an attributed data graph $G(V, E, X)$, where V is the vertex set, E is the edge set, and X contains an embedding vector X_v for each vertex $v \in V$. For instance, in the domain of e-commerce, nodes correspond to customers and products, edges represent purchase behaviors and customer relationships, and embedding vectors capture specific information for each customer or product. A request $R = \{v_1, v_2, \cdots, v_m\}$ contains some target nodes in the data graph, often referred to as *seeds*. The goal of model serving is to utilize a pre-trained SGRL model to predict a label for the given request. For instance, in link prediction for product recommendations, a request contains a consumer node and a product node. The goal here is to predict whether there exists an edge between the two nodes, indicating whether a product should be recommended to a particular consumer. As shown in Figure 2, SGRL serving usually involves two stages.

Subgraph extraction. An enclosing subgraph g_R of request R is extracted from the data graph G around the seed nodes specified by R. Consider the two requests in Figure 2: the first request employs $\{A, B\}$ as seed nodes, whereas the second request utilizes $\{C, D\}$. For the first request, we find the common 1-hop neighbors of seed nodes $\{A, B\}$ as $\{E, F, G, H, J\}$, and then extract all edges among the seeds and the selected neighbors. SGRL models adopt different subgraph extraction

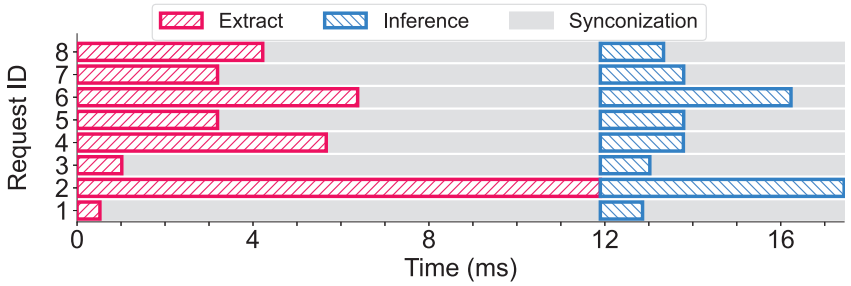


Fig. 3. The execution timeline for 8 requests in a batch.

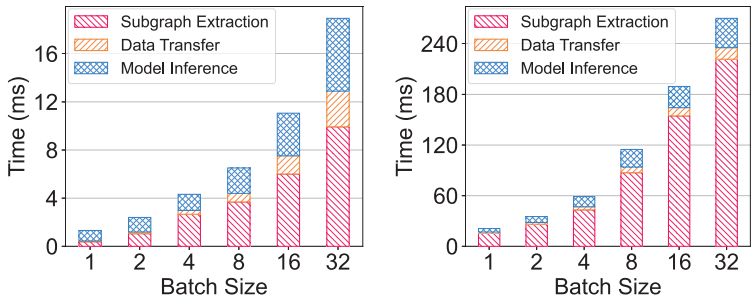
policies, e.g., k-hop neighbors, neighbors visited by random walks, and the neighbors of different seeds can be unioned or intersected. Labels are also assigned to each node in the extracted subgraph to indicate their relations to the seeds. Example labeling policies include shortest distances to the seeds and random walk landing probabilities. For each node v, the labels assigned by the seeds are concatenated with its embedding vector X_v in the data graph to form a feature vector f_v.

Model Inference. The extracted subgraph g_R, along with the feature vectors of its nodes, is fed to the trained SGRL model to compute output. For instance, in Figure 2, the model decides that there is likely to be an edge between A and B, whereas the likelihood is low for the presence of an edge between C and D. In this stage, SGRL models usually adopt the structure of classical GNN models, e.g., GIN [79], GCN [46], and SAGE [38].

SGRLs are more expressive than classical GNN models because subgraph extraction allows to consider the joint neighborhood of the seeds. Many SGRL models have been proposed for various graph tasks. For example, SEAL considers link prediction, extracts the k-hop neighbors of the seeds, and uses the GIN model. GraIL predicts the relation type between two nodes and follows the subgraph extraction policy of SEAL. SEAM takes k seeds and predicts high-order patterns, i.e., the existence of a motif (e.g. k-cliques and k-cores) among the seeds, to support applications like fraud detection. NBFNet [102] represents a node pair using all paths connecting them and then formulates link prediction as a path-based reasoning problem. ULTRA [32] constructs relational subgraphs where nodes correspond to relations and edges represent their co-occurrence, and applies GNN models over this subgraph to learn transferable relation representations for zero-shot link prediction.

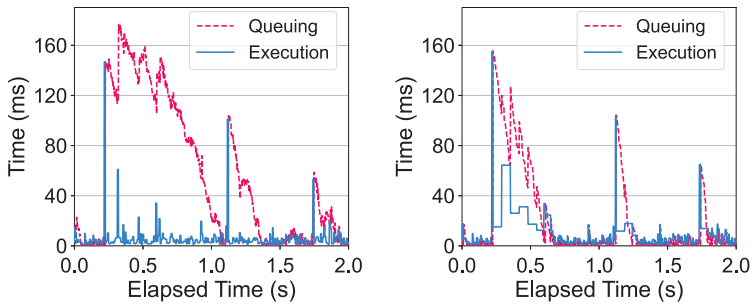
2.2 Problems of Existing Solutions

Existing SGRL serving solutions. Online SGRL model serving requires low latency and high throughput since it targets user-facing applications such as recommendation and fraud detection. However, existing solutions (e.g., DGL [72] and PyG [30]) perceive subgraph extraction and model inference as an entire process and adopt the *synchronous batch processing* (SBP) strategy to process the requests, without considering the irregular workloads of SGRL requests. In particular, the data graph is stored in CPU memory for capacity, and inference is conducted on GPU for efficiency. The system first collects a batch of requests, whose number does not exceed a pre-defined threshold (i.e., the batch size). Subgraph extraction is then conducted for all requests in the batch simultaneously, where one CPU thread is assigned to each request. After all requests in the batch finish extraction, their subgraphs are transferred collectively to the GPU for model inference. Processing of the next batch begins only after the previous batch finishes. Notice that modern ML frameworks such as PyTorch [62] provide flexible data loaders to prepare training data for ML models. Developers can implement customized preprocessing logic (e.g., subgraph extraction) and utilize multiple CPU



(a) P50 latency (b) P99 latency

Fig. 4. Request processing time w.r.t. batch size.



(a) Batch size = 4 (b) Batch size = 32

Fig. 5. The execution and queuing time of the requests over a sampled period.

threads to accelerate data preparation. However, these data loaders process and return data in a synchronous batch manner, and thus their behavior closely resembles the SBP strategy.

To analyze existing SGRL serving solutions in depth, we profile PyG using the SEAL model and Papers100M graph, discovering three main deficiencies.

Long batch synchronization time. Figure 3 shows the execution timeline for 8 requests within an example batch. Notably, the execution time of *request 2* markedly exceeds that of the other requests during both the subgraph extraction and model inference stages, which leads to extended synchronization times for other requests, thus prolonging their end-to-end execution durations. This is due to the SBP adopted in existing systems, which mandates requests in a batch to start and finish both stages synchronously.

Long subgraph extraction time. In Figure 4, we report the P50 and P99 batch execution latency with different batch sizes. We also decompose the latency into 3 parts, namely, *CPU subgraph extraction*, *CPU-GPU data transfer*, and *GPU model inference*. The results show that the extraction time dominates request processing time, especially for P99 latency and larger batch sizes. The reasons are two-fold. Firstly, existing systems use a single thread to execute the subgraph extraction procedure for each request, which leads to long subgraph extraction time for heavy requests. Secondly, SBP exacerbates the situation due to the batch straggler problem. The two effects are more significant for larger batch size (i.e., requests are more likely to be straggled) and queries in the latency tail.

Long queuing time. We measure the request execution and queuing time under an input load of 500 requests per second (RPS) with the a batch size of 4 and 32, respectively. Figure 5 shows the temporal variations in these two metrics over a duration of 2 seconds. We can observe that following the execution of each heavy request, characterized by a long execution time, there is

a spike in the queuing time of subsequent requests. This is attributed to the fact that requests are processed in an FCFS manner in existing systems. Specifically, heavy requests undergo a long processing time for both subgraph extraction and model inference, resulting in significant delays in the processing of subsequent requests.

3 SG-Serve System Overview

The architecture of SG-Serve is depicted in Figure 6. In particular, SG-Serve comprises two main components, i.e., subgraph extraction engine and model inference engine, which correspond to the two stages of SGRL serving. The two components are connected by a shared queue and communicate via a producer-consumer paradigm. The subgraph extraction engine extracts subgraphs for incoming requests and puts the extracted subgraphs into the shared queue. The model inference engine retrieves subgraphs from the queue, organizes several subgraphs into a batch, and dispatches the batch to the GPU for model inference. In contrast, existing systems execute the two stages in synchronous batches (i.e., the SBP strategy). As a result, the requests that finish extraction early need to wait for the other requests in the batch before entering the model inference stage, leading to the batch straggler problem. Our producer-consumer design eliminates such stragglers as a request can run inference once it finishes extraction.

Request processing workflow. On receiving a request, the subgraph extraction engine creates a directed acyclic graph (DAG) (Section 4.1), which models the tasks for subgraph extraction and enables parallel execution with multiple threads. The DAG is then submitted to a pool of CPU threads for execution, and the threads are scheduled by our work-stealing scheme to process multiple concurrent requests while balancing parallelism and HoL blocking (Section 4.3). Subgraph extraction finishes when all tasks in the DAG are processed, and the extracted subgraph is appended to the end of the subgraph queue for model inference.

The model inference engine creates two separate GPU processes to conduct model inference, and each process takes 50% of the GPU resource (Section 5). This design avoids HoL blocking by preventing a heavy request from monopolizing the entire GPU for a long time. A dispatcher process monitors the status of the GPU processes. When a GPU process becomes idle, the dispatcher selects a batch of subgraphs from the subgraph queue using our workload-aware batching scheme (Section 5) and notifies the GPU process. Then, the GPU process loads the batch to the GPU for model inference. Finally, the model outputs for a batch of subgraphs are sent back to the host CPU as inference results.

Data layout. SG-Serve stores both the graph topology and node embeddings in host memory due to its large capacity. To reduce the overhead of host-GPU data transfer, we cache the embeddings of hot nodes on GPU such that these embeddings no longer need to be loaded along with the extracted subgraphs. Currently, we treat the nodes with high degrees as hot but SG-Serve can also support identifying hot nodes using alternative criteria (e.g., page rank scores).

Algorithm deployment. SG-Serve provides expressive API (Section 4.2) that allows users to customize their subgraph extraction algorithms, which are run by the subgraph extraction engine as function calls. For model inference, SG-Serve imports trained SGRL models stored in files that follow the open neural network exchange (ONNX) standard [3], which is supported by many machine learning frameworks [8, 20, 61, 62]. The inference engine loads the model parameters upon system initialization.

4 Parallel Subgraph Extraction

In this part, we first abstract subgraph extraction into a directed acyclic graph (DAG) of tasks to enable intra-request parallelism in Section 4.1, then present our API for subgraph extraction in

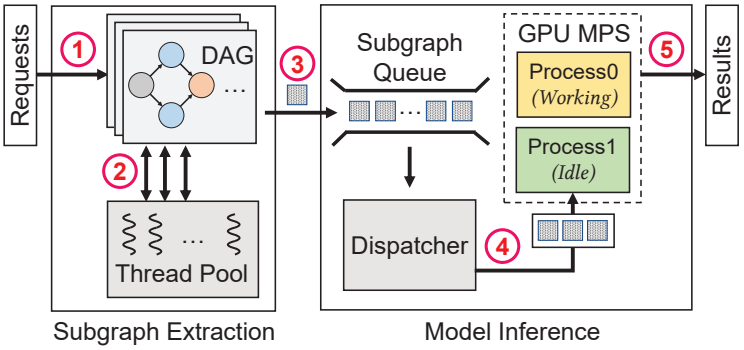


Fig. 6. The system architecture and workflow of SG-Serve.

Section 4.2, and finally discuss how to schedule the concurrent execution of multiple DAGs from different extraction requests in Section 4.3.

4.1 Abstraction of Subgraph Extraction

As introduced in Section 2.1, the subgraph extraction methods of existing SGRL models adopt diverse processing logic (e.g., k-hop and PPR). Our abstraction should be general so that users can express existing extraction methods and possible extensions in the future. Moreover, the abstraction should also expose parallelization opportunities by modeling (some part of) subgraph extraction as independent tasks. This is different from existing systems, where the subgraph extraction process is treated as an indivisible user-defined function (UDF), and thus each request can only be handled by a single CPU thread without parallelization.

We survey the subgraph extraction methods of 16 popular SGRL models [9, 11, 13, 18, 29, 48, 52–54, 60, 70, 71, 83, 84, 90, 92] and observe that all of them can be decomposed into the following five sequential steps:

❶ Expand selects nodes from the data graph based on the seed nodes, such as the k-hop neighbors. The selection process can be conducted either individually for each seed node or collectively for all the seed nodes.

❷ Merge consolidates the node sets selected for individual seeds to obtain the nodes in the extracted subgraph. This is typically achieved by set intersection or union. Merge can be skipped for collective expansion.

❸ Induce obtains the edges of the extracted subgraph by considering all the merged nodes and retrieving the edges between them in the data graph.

❹ Encode generates structural labels for nodes in the extracted subgraph to reflect their relations to individual seeds (e.g., the shortest distance to the seeds). It typically runs a graph algorithm on the extracted subgraph. Some algorithms require exclusive encoding, which removes the other seeds from the subgraph when computing encoding for a seed.

❺ Combine aggregates the labels assigned by individual seeds to each node in the subgraph. Common aggregating methods include summation, minimum, and maximum.

We abstract the five steps as operators and model the process of subgraph extraction as a DAG of these operators. Figure 7 shows an example DAG along with the input and output of each operator. Specifically, the DAG has two *Expand* operators, each performing k-hop neighbor expansions individually for a seed. The node sets produced by the two *Expand* operators are unioned by the subsequent *Merge* operator. Then, the *Induce* operator collects the edges between the merged nodes from the data graph to form the extracted subgraph. Afterward, two *Encode* operators generate

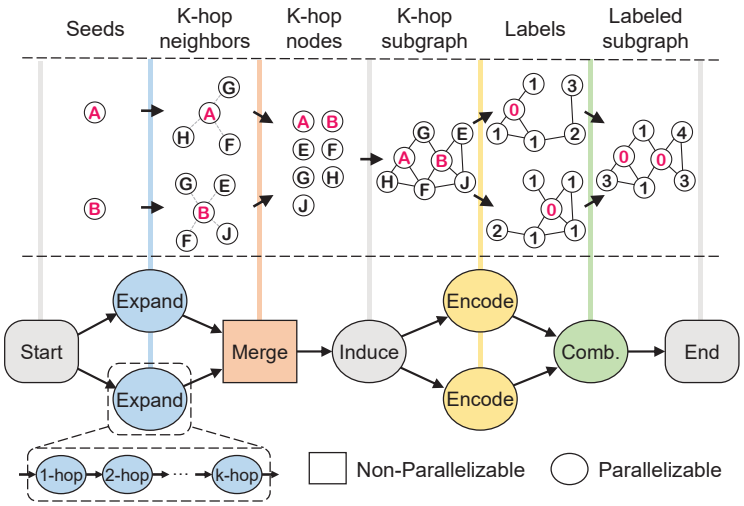


Fig. 7. An example DAG for subgraph extraction.

labels for nodes in the subgraph, each solving the distances of the nodes to a seed node. Finally, the *Combine* operator aggregates the labels of each node into a final label using summation. Besides the five operators, the DAG also uses a *Start* operator for signal initialization and an *End* operator to collect output.

With the DAG abstraction, operators that have no dependencies can be executed in parallel, e.g., the two *Expand* operators and *Encoder* operators in Figure 7. Besides such inter-operator parallelism, we also exploit fine-grained intra-operator parallelism. In particular, we observe that the computation of *Expand*, *Induce*, *Encode*, and *Combine* operators in existing algorithms follows a *node-parallel paradigm* [36, 57], which ingests a set of nodes and processes each node independently. This is because they work on graph, and graph operations essentially handle nodes and edges. For instance, each step of *Expand* takes some nodes and obtains their 1-hop neighbors, and getting the neighbors of each source node is independent[1]. *Combine* is also independent for each node while *Induce* is independent for different node pairs. *Encode* runs a graph algorithm, and graph algorithms are essentially node-parallel. For these operators, we use a *segment* of input nodes as the minimum task execution and scheduling unit, and each segment contains 128 nodes by default. This allows to break the workload of an operator into segments and execute the segments in parallel. Note that we do not adopt intra-operator parallelism for the *Merge* operator.

4.2 Application Programming Interface

Based on the operators summarized in Section 4.1, we propose the API in Figure 8 for subgraph extraction. The key design principle is to expose the node-parallel pattern of the operators by writing their functions as operations on individual nodes.

- The *Expand* operator involves four interfaces in total. Firstly, the expand_individual variable allows users to choose between individual or collective expansion, the expand_hop variable sets the hop count for expansion, and the expand_unique variable deduplicates the results produced by different nodes at each hop when set as true. The expandStep function runs 1-hop of expansion, and thus takes a node and the *context* as inputs, updates the expanded node set, and returns the new nodes for expansion. The *context* encompasses references to the data graph, current subgraph, and seeds.

[1]We insert a *Merge* operator to deduplicate the results produced by different source nodes.

```
// Expand
bool expand_individual;
int expand_hop;
bool expand_unique;
NodeList expandStep(Node n, Context ctx);
// Merge
NodeList merge(Vec<NodeList> nodes);
// Encode
bool encode_exclusive;
int encode_hop;
bool encode_unique;
Label encodeInit(Node n, Context ctx);
Vec<Node> encodeStep(Node n, Context ctx);
// Combine
Label combine(LabelList labels);
```

Fig. 8. API for subgraph extraction.

```
// Expand
bool expand_individual = false;
int expand_hop = k;
bool expand_unique = true;
NodeList expandStep(Node n, Context ctx) {
  NodeList frontiers = ctx.graph.neighbors(n);
  ctx.subgraph.add(frontiers);
  return frontiers;
}
// Merge
NodeList merge(Vec<NodeList> ns) {return ns[0];}
// Encode
bool encode_exclusive = true;
int encode_hop = -1;
bool encode_unique = true;
Label encodeInit(Node n, Context ctx) {
  return ctx.seeds.has(n) ? 0 : MAX_INT;
}
NodeList encodeStep(Node n, Context ctx) {
  NodeList frontiers = {};
  for (Node u : ctx.subgraph.neighbors(n)) {
    if (atomicMinUpdate(u.label, n.label+1))
      frontiers.add(u);
  }
  return frontiers;
}
// Combine
Label combine(LabelList ls) {return MIN(ls);}
```

Fig. 9. Express SEAL subgraph extraction with our API.

- The *Merge* operator features a merge function that accepts multiple node sets derived from the seeds and outputs a single consolidated node set. We provide built-in implementations for set intersection and union as common merge operations.

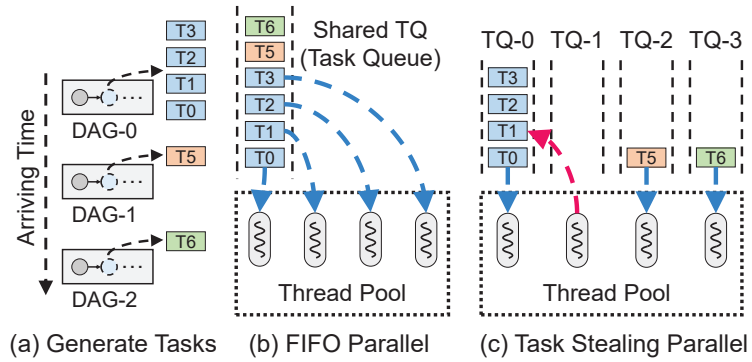


Fig. 10. Task scheduling options for subgraph extraction.

- The *Encode* operator uses the encode_exclusive variable to decide whether it is run under the exclusive mode. The encode_hop variable specifies the hop count, the encode_unique variable deduplicates the results at each hop when set as true, and the encodeInit function is utilized to set initial values for the node labels. The encodeStep function takes an input node, updates the labels of its neighbors in the subgraph, and yields the input nodes for the next step.
- The *Combine* operator's combine function takes as input the node labels generated by each seed and produces the aggregated labels. We provide standard combine operator implementations such as summation, maximum, minimum, and concatenation.

We do not require users to specify the *Induce* operator because all existing SGRL models adopt the same induce logic (i.e., checking edge existence), which is executed automatically by SG-Serve. If later SGRL models develop other induce patterns, we can provide an interface for *Induce*.

Our API is general in expressing various subgraph extraction algorithms. We use SEAL as an example in Figure 9. In particular, SEAL extracts k-hop neighbors of the seeds and labels a node using the shortest distance between the node and the seeds. In Figure 9, the expandStep function (lines 5-9) adds all neighbors of a node into the subgraph's node set and returns these neighbors as frontiers for the next expansion step. The encodeStep function (lines 19-26) updates the distance labels for the neighbors of the input node. The variable encode_hop is set to be -1, which means that the encoding process terminates when no label updates can be made. Finally, the combine function (line 28) selects the minimal distances between a node and the seeds to determine its final label.

Our API is different from those used in graph processing systems [57, 101] and subgraph matching systems [16, 44]. Graph processing systems usually adopt a node-centric API, where each node sends and receives messages and updates its state iteratively. In subgraph matching systems, users specify query graphs with different topologies using graph query languages such as Gremlin [65] and Cypher [31]. In contrast, our API allows users to implement customized methods for extracting enclosed subgraphs centered around a set of seed nodes and generating structural labels that capture each node's relation to the seeds, which cannot be implemented using the APIs of graph processing and subgraph matching systems. Furthermore, our API models the dependency between the functions of an extraction algorithm for parallel execution.

4.3 Task Scheduling

The CPU threads process the DAGs of multiple requests concurrently, and we need to schedule the tasks from different DAGs. In particular, for a DAG, an operator can be executed when its

Algorithm 1: Workload-aware batching algorithm

N_b: the maximum number of nodes in a batch.
E_b: the maximum number of edges in a batch.
while True **do**
  $\mathcal{W} \leftarrow$ wait for a free worker
  $\mathcal{B} \leftarrow \{\}$
  for *subgraph* $\mathcal{G}$ *in subgraph queue* $\mathcal{Q}$ **do**
    if $\mathcal{B}.n + \mathcal{G}.n \leq N_b$ **and** $\mathcal{B}.e + \mathcal{G}.e \leq E_b$ **then**
      Pop $\mathcal{G}$ from $\mathcal{Q}$ // light subgraph
      $\mathcal{B} \leftarrow \mathcal{B} \cup \{\mathcal{G}\}$
    else if *no heavy running* **and** $\mathcal{B} = \{\}$ **then**
      Pop $\mathcal{G}$ from $\mathcal{Q}$ // heavy subgraph
      $\mathcal{B} \leftarrow \{\mathcal{G}\}$
      break
  Submit $\mathcal{B}$ to $\mathcal{W}$

dependencies have been resolved (i.e., the parent operators in the DAG complete). If an operator follows the node-parallel pattern, we will generate fine-grained tasks for it in the granularity of node segment as introduced in Section 4.1. Thus, each request has multiple ready tasks to run.

As shown in Figure 10(b), a natural solution is to put the ready tasks of all requests into a common queue and assign the threads to fetch tasks from the queue in a first-in-first-out (FIFO) order. However, this solution suffers from HoL blocking due to the irregular extraction workloads of the requests. Specifically, a request that needs to extract a large subgraph will generate many tasks, e.g., DAG-0 in Figure 10(b). These tasks can take up all the CPU threads and starve the tasks of the other requests, e.g., DAG-1 and DAG-2 in Figure 10(b). Compared with using a single thread to process each request in existing systems, this solution reduces the execution time of heavy requests by exploiting parallelism but increases the queuing time of the other requests.

To meet the two seemingly contradicting goals, we adopt a simple but effective task-stealing parallel strategy, which is shown in Figure 10(c). The principle is that only spare threads should be leveraged for parallelization. In particular, each newly launched DAG is assigned to a primary thread, and each thread can serve as the primary thread of only one DAG. The ready tasks of a DAG are put into the local task queue of its primary thread instead of a global task queue. By default, a thread executes the tasks from its local queue. Only when a thread finishes its DAG and cannot obtain a new DAG (i.e., becomes idle), it steals tasks from the other threads using a round-robin scheme. The task-stealing strategy ensures that no individual request monopolizes all threads by associating each DAG with a primary thread. It also improves thread utilization and reduces the subgraph extraction time by enabling the free threads to take tasks from other threads.

We note that although work stealing is a common technique for processing concurrent requests [17, 101], the fine-grained task decomposition provided by our API lays the foundation of applying it to the subgraph extraction process of SGRL models. In contrast, existing solutions treat subgraph extraction as a standalone function and thus one request must be handled by only one thread. Our API makes work stealing simple as it decomposes subgraph extraction into fine-grained tasks and thus naturally allows multiple threads to execute the tasks of the same request in parallel.

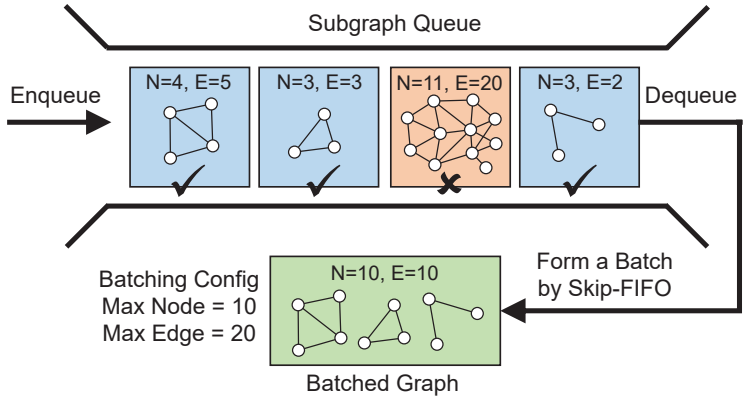


Fig. 11. Running example of workload-aware batching.

5 Workload-aware Model Inference

In this part, we introduce how to organize the subgraphs into batches for model inference, and the key consideration is the irregular workload of the requests.

Model serving systems usually batch a pre-defined number of requests for GPU computation to improve resource utilization and throughput [24, 34, 68]. Existing serving systems for SGRL models also adopt this strategy but encounter two problems due to the irregular inference workload of the requests. First, if a heavy request is batched with light requests, the heavy request will prolong the inference time of the light requests, which causes the batch straggler problem. Second, if a batch contains a heavy request, it may monopolize the GPU for a long time, which keeps the incoming light request queuing and causes the HoL blocking problem. Our batching and dispatching strategies avoid the two problems. The key observation is that the inference workload for a SGRL request is determined by the number of nodes and edges in its subgraph. This is distinct from existing systems that assume all requests to have uniform workload.

Workload-aware batching. Algorithm 1 shows the batching strategy of SG-Serve. The key observation is that the inference workload for a batch is determined by the number of nodes and edges in the subgraphs, as opposed to request count in other ML models. As such, Algorithm 1 collects a batch of subgraphs satisfying the maximum number of nodes and edges. In particular, when a GPU worker becomes idle, the dispatcher checks the subgraph in the subgraph queue in FIFO order. A subgraph is appended to the current batch if adding it does not violate the node and edge limits. To fully utilize the capacity of a batch, Algorithm 1 does not stop when adding a request exceeds the batch limits, instead, it skips the request and continues to check other requests until reaching the end of the queue. Note that we disable the node and edge limits when adding the first subgraph to the batch, ensuring that heavy requests can be handled. Our strategy does not starve the skipped (possibly heavy) requests as they are now at the head of the queue, and can be processed soon by the subsequent batches.

Figure 11 provides a running example of Algorithm 1. The node and edge limits are 10 and 20, respectively, and there are four subgraphs in the queue. The dispatcher checks the subgraph from the head and adds the first subgraph to the batch. The second subgraph is skipped because adding it violates the node and edge limits. The dispatcher continues to check the third and final subgraphs and adds them to the batch. The second subgraph does not starve as it will be processed by the next batch. The example shows that our workload-aware batching strategy has two advantages. First, heavy requests are processed as standalone batches, and thus they do not prolong the inference

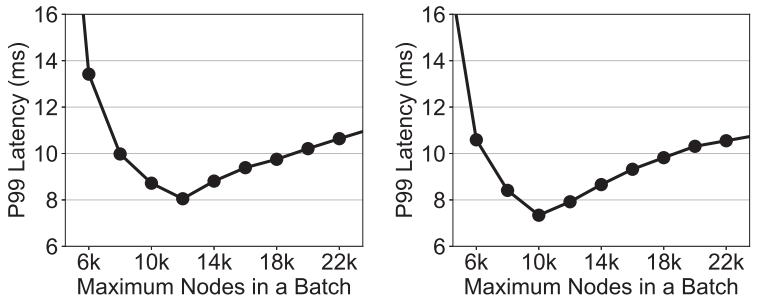


Fig. 12. Model inference latency (P99) v.s. the maximum number of nodes in a batch under 2k RPS.

time of other requests. Second, light requests that arrive after a heavy request are not blocked by the heavy request and can be added to the current batch.

Dual-process dispatching. Although workload-aware batching organizes a heavy request as a standalone batch, it may still monopolize GPU for a long time and make the other requests queuing. To tackle this problem, SG-Serve leverages Nvidia's multi-process service (MPS) to partition the GPU into two distinct regions and runs a worker process for model inference on each region. MPS guarantees that the kernels launched by each process utilize at most half of the total GPU streaming multiprocessors, and the two processes can run in parallel. As shown in Algorithm 1, the dispatcher ensures that the two processes do not handle heavy requests simultaneously. In particular, we consider a request to be heavy if it is larger than the node and edge limits for a batch. If one process is running a heavy request, the dispatcher skips the heavy requests when collecting a batch of subgraphs for the other process. This design ensures that there is at least one process open for the light request and thus avoids HoL blocking for inference. Since half of the GPU resources are reserved for light requests, only half of the GPU may be utilized when all requests are heavy. However, as shown in Figure 1, heavy requests come far less frequently than light requests, making it unlikely for many heavy requests to arrive simultaneously. Moreover, once light requests appear, the previously idle GPU resources are promptly utilized. Therefore, the overall GPU utilization is not significantly affected by our resource reservation.

Profiling-based batch configuration. Just as existing systems need to set the number of requests in a batch (i.e., maximum batch size), SG-Serve needs to properly configure the node and edge limits for a batch to achieve good performance. We first simplify this problem by deciding the edge limit E_b according to the node limit N_b. In particular, we keep track of the average ratio of the number of edges over the number of nodes in the extracted subgraphs, which is denoted as λ, and use $E_b = \lambda \times N_b$. To explore the influence of N_b, we profile the model inference latency of SEAL on two datasets in Figure 12. The results show that there exists an optimal N_b that minimizes latency, and latency increases when N_b deviates from the optimal value. This is because the optimal N_b strikes a balance between queuing time and computation time. On the one hand, with a small N_b, the GPU computation time of a batch is short but the inference throughput is low, which leads to a long request queueing time. On the other hand, with a large N_b, the GPU resource utilization and inference throughput are high, which reduces queuing time, but the GPU computation time is long. The penalty of using a large N_b is smaller because computation time increases slower than the queuing time.

Building a cost model to set the optimal N_b is challenging because it depends on the SGRL model, underlying data graph, input request throughput, and hardware platform. Thus, we use a profiling-based method instead. In particular, we start with a large enough N_b (e.g., 20k) and reduce

N_b by a step size (e.g., 1k) each time to measure the inference latency for a sufficient number of requests. We continue to reduce N_b until the inference latency is observed to increase. This process is quick as each batch of requests takes only a short time (e.g., tens of ms).

One may wonder whether it is beneficial to categorize the requests into more than 2 kinds and run more GPU processes. We observe that such configurations yield little performance gain over using 2 request kinds and GPU processes. This is because separating the heavy requests from the normal ones already solves the straggler and head-of-line block problems to a large extent.

The scheduling optimizations, i.e., workload-aware batching and dual-process dispatching, have not been implemented by existing ML serving systems such as Triton [6] and vLLM [47]. Specifically, Triton is a general-purpose ML serving framework designed to support diverse models, execution engines, and hardware backends. It handles request reception, model execution, and response delivery in a *model-agnostic* manner for good generality. In contrast, our scheduling optimizations consider the unique properties of SGRL inference requests, i.e., different requests have different workloads, and the workload of a request is decided by the number of nodes and edges in its subgraph. As a system for serving large language models (LLMs), vLLM introduces scheduling optimizations tailored to the auto-regressive computation pattern of LLMs, such as paged attention for dynamic GPU memory allocation and continuous batching for requests with varying output lengths. However, SGRL requests do not conduct auto-regressive computation and do not suffer from the dynamic sequence length problem, and thus our scheduling optimizations are entirely different from vLLM.

6 Implementation

We implement SG-Serve atop Pytorch [62] with about 6k lines of C++ code. SG-Serve is deployed as a runtime backend for Triton [6], a state-of-the-art machine learning serving system from Nvidia. We incorporate several implementation optimizations to improve system efficiency. Since our system is implemented as a backend of triton, we focus on one GPU setting and rely on the instance group configuration of triton to deploy our systems on multiple GPUs.

Fast inter-process communication. During model inference, the dispatcher process needs to send subgraph batches to the GPU worker processes, requiring frequent inter-process communication. To avoid the overhead of data serialization and deserialization, we leverage shared-memory buffers for fast data transfer between the dispatcher and GPU workers We further pin these buffers so that the subgraph batches can be sent to GPU in a zero-copy manner.

Shared model parameters and embedding cache. With MPS, our two GPU worker processes operate in separate address spaces, which requires replicating the model parameters and the cached node embeddings. To avoid data replication and increase cache size, we keep the model parameters and node embeddings in the shared memory of GPU and let the GPU processes access these data via inter-process call (IPC).

Pipeline communication and computation. Conducting model inference for a batch involves two sequential phases, i.e., the *communication* phase that transfers data from the host to the GPU and the *computation* phase that conducts model forwarding on the GPU. To avoid GPU idle time during data transfering, we employ separate CUDA streams for communication and computation kernels in each GPU worker, which allows their executions to overlap and thus maximizing GPU utilization.

7 Experimental Evaluation

In this part, we conduct extensive experiments to evaluate our SG-Serve and compare it with existing systems for SGRL serving. The main findings include:

Table 2. Statistics of the graph datasets in the experiments.

Name	Abbr.	Nodes	Edges	Dim.	Size
Orkut	OT	3.1 M	120 M	100	1.3G
Livejournal	LJ	4.8 M	68 M	100	2.5 G
Friendster	FS	65 M	1.8 B	128	48.2 G
Papers100M	PP	111 M	1.7 B	128	69.1 G

- SG-Serve significantly outperforms existing solutions, achieving much shorter request latencies at the same request load and serving much higher request loads at the same request latency.
- The optimized subgraph extraction engine and model inference engine are both efficient and contribute to overall performance.
- The key design choices of SG-Serve (e.g., for thread scheduling on CPU and request batching on GPU) outperform their alternatives and are effective in improving system performance.

7.1 Experiment Settings

Datasets. We conduct experiments on three popular and public graphs, i.e., Orkut, Livejournal and Friendster from the Stanford Network Analysis Platform [80], and Papers100M from the Open Graph Benchmark [39]. Their statistics are reported in Table 2, where *Dim.* refers to the dimension of node features and *Size* includes both graph topology and node features. Since there are no open-source request traces for SGRL model serving, we generate traces by randomly sampling nodes in the graphs to construct requests and setting the arrival time of the requests by a Poisson process [34, 87]. There is an arrival rate parameter for Poisson process that determines the expected number of incoming requests per second and is used to control the input request throughput. Note that such a synthetic request generation method is widely used when evaluating model serving systems [51, 76].

Algorithms. We use three popular SGRL models with diverse computation patterns for the experiments, i.e., SEAL [92], Shadow [90], and GraIL [71]. In particular, SEAL extracts the k-hop neighbors collectively for the seeds, labels the nodes using their shortest distances to seeds, and applies the GIN model [79] for inference. Shadow employs the approximate page rank algorithm to extract a node set for each seed individually and then unions these node sets. The node labels are generated by concatenating their shortest distances to the seeds, and the GCN model [46] is applied for inference. GraIL individually extracts k-hop neighbors for each seed and interests these neighboring nodes to construct the subgraph. GraIL utilizes the same labeling method as Shadow and employs the GCN model for inference. We deploy these models on SG-Serve using our API following the implementations and configurations in their open-sourced codes [1, 4, 5].

Baseline systems. Existing SGRL systems are implemented on open-source graph learning frameworks such as PyG [30] and DGL [72], and they follow the synchronous batch processing (SBP) design introduced in Section 2.2 to process requests. They run on Python and thus have higher runtime overhead than our SG-Serve, which is based on C++. For a fair comparison, we re-implement their workflow on the codebase of SG-Serve and denote it as SBP.

To evaluate the designs proposed in SG-Serve, we progressively incorporate them and create two variants of SG-Serve for ablation studies. Initially, we enhance the baseline SBP with the producer-consumer architecture, which is referred to as *PC*. Next, we incorporate our parallelized subgraph extraction into PC, which is called *PC+E*. Finally, we add the irregularity-aware model inference to *PC+E*, which results in the complete SG-Serve system. Following existing works [34, 76], we use the P99 request latency and the request throughput at given P99 latency as the two main

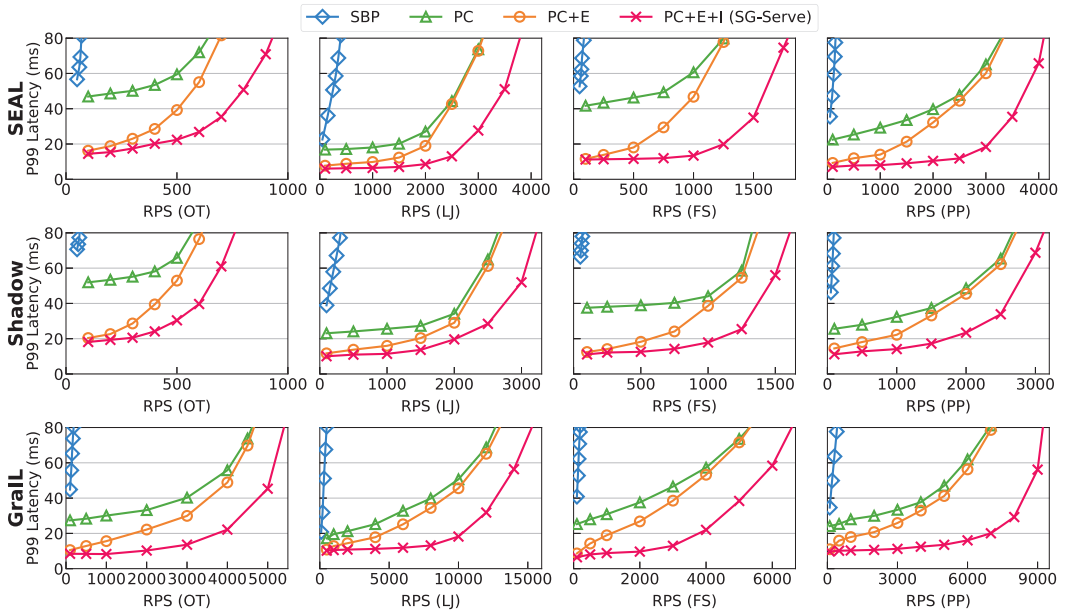


Fig. 13. The P99 latency of SG-Serve and baselines, SBP is the baseline PyG, *PC* adopts the producer-consumer design, *PC+E* also adds our subgraph extraction optimizations, and *PC+E+I* is the complete SG-Serve system.

Table 3. The request throughput of SG-Serve over the baseline SBP when serving requests at a P99 latency of 80ms.

	SEAL	Shadow	GraIL
OT	12.4	13.8	26.3
LJ	8.75	10.7	29.4
FS	21.9	23.1	31.6
PP	26.7	33.3	25.7

performance metrics. Note that SBP, *PC*, and *PC+E* batch the requests according to count, and for a fair comparison, we tune their batch sizes to minimize the P99 latency (for each request throughput).

Testbed. We conducted all the experiments on a machine hosting an Nvidia T4 GPU with 15GB memory, an Intel Xeon 8259CL CPU with 24 cores, and 187GB main memory[2]. The operating system is Ubuntu-18.04 with Linux kernel version 5.4.0. The version of Nvidia CUDA Toolkit is 11.6.

7.2 Main Results

Figure 13 reports the P99 request latency of all systems under different input request throughput (i.e., request per second, RPS). We ensure the P99 latency falls in a reasonable range to ensure the QoS of online applications [22, 25, 26], and the request throughput varies for the datasets and algorithms because of their differences in computation workload. The results in Figure 13 show that the P99 latency of SBP grows quickly with request throughput, and given a latency of 80ms, its throughput is below 500 RPS for all cases. Compared with SBP, PC reduces the P99 latency by up to 4.76x at the same throughput and also improves the throughput by up to 25x at the same P99 latency.

[2]The reported GPU memory and main memory are the values shown by the system monitor.

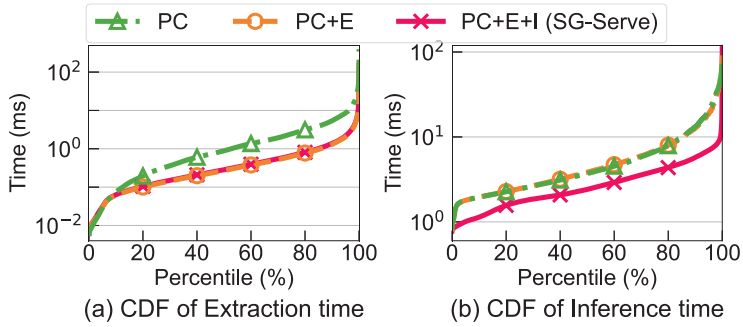


Fig. 14. The subgraph extraction and model inference time distributions for SEAL on Papers100M at 2k RPS.

This is because PC avoids the batch synchronization barrier in SBP with the producer-consumer design, which solves the batch straggler and HoL blocking problems in the subgraph extraction stage.

We can also observe from Figure 13 that PC+E achieves lower P99 latency than PC, especially when the request throughput is low. In particular, at an input request throughput below 1000 RPS, PC+E reduces the P99 latency of PC by 1.24x-3.60x. This is because PC+E employs our subgraph extraction optimizations, which parallelize extraction for the heavy requests to reduce their extraction time. However, the performance gap between PC+E and PC diminishes when the request throughput increases. This is because the model inference time begins to dominate the end-to-end request processing time. Such an explanation is also evidenced by the large improvement of SG-Serve over PC+E at high request throughput, which is attributed to the effect of our workload-aware model inference in reducing the inference time. When the throughput is over 1000 RPS, SG-Serve reduces the latency of PC+E by 1.21x-3.92x due to our inference optimizations.

Overall, the results in Figure 13 suggest that SG-Serve significantly outperforms SBP, and both parallelized subgraph extraction and workload-aware model inference make non-trivial contributions to the performance improvements. Table 3 reports the throughput improvement of SG-Serve over SBP at a request latency of 80ms. The results show that SG-Serve boosts the throughput of SBP by over 10x in most cases and up to over 30x. Such improvements allow SG-Serve to significantly reduce the operating resources when serving a given request load at target latency.

Case study. To better understand the performance gains of our designs, we plot the cumulative distribution function (CDF) of the subgraph extraction and model inference time in Figure 14. We employ the SEAL model on the Papers100M dataset and use a request load of 2000 RPS. The subgraph extraction time is the duration from a request's arrival to the completion of its subgraph extraction, while the inference time spans from the moment a request enters the subgraph queue to the GPU finishing its inference computation. We do not include SBP in Figure 14 because its extraction time and inference time are much longer than the others and thus would make the figure difficult to read.

As shown in Figure 14(a), SG-Serve and PC+E have identical extraction time distribution, and they both significantly outperform PC. This is because both SG-Serve and PC+E adopt our parallelized subgraph extraction optimizations. Moreover, the gap between PC+E and PC is small when the extraction time is short but enlarges when the extraction time increases. This is because small requests finish quickly and thus their tasks are seldom executed by threads other than their primary thread, while heavy requests have more chances for task streaking due to their long extraction time and thus benefit more from parallelism. We can observe from Figure 14(b) that SG-Serve has a much shorter inference time than both PC and PC+E due to our irregularity-aware model inference.

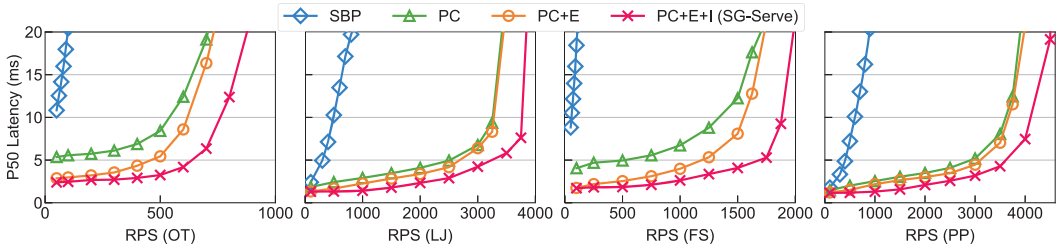


Fig. 15. The P50 latency of SG-Serve and the baselines for the SEAL model at varying input request loads.

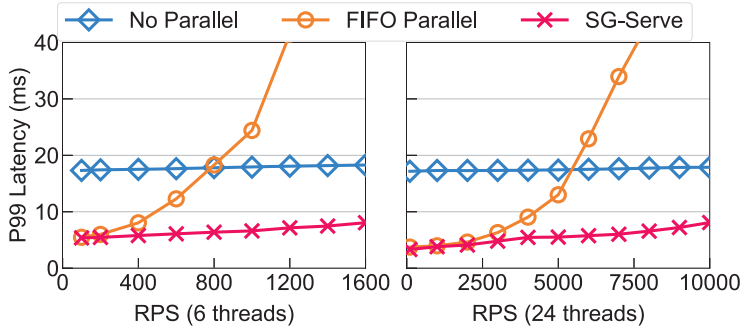


Fig. 16. The P99 subgraph extraction latency of SEAL on Papers100M for different task scheduling polices.

The improvement of SG-Serve over PC+E is similar for different inference time values since both small and heavy requests benefit from avoiding HoL blocking and processing the other requests faster to reduce the queuing time.

P50 request latency. Figure 15 reports the P50 request latency of SG-Serve as well as the baseline systems. We use the SEAL model and note that the observations are similar to the other models. We can observe from the figure that the relative performance of all the systems is the same as their P99 latency results in Figure 13, that is, SG-Serve consistently outperforms all baseline systems, and both parallelized subgraph extraction and irregularity-aware model inference contribute to performance improvements. In particular, SG-Serve reduces the P50 latency by up to 19.3X, 4.28x, and 3.99x for SBP, PC, and PC+E, respectively. For P50 latency, the performance gaps between the systems are smaller than P99 latency because P99 latency is largely determined by the heavy requests, which are challenging to process efficiently.

7.3 Benefits of the Design Choices

Task scheduling policy. Recall that we use a task-stealing policy when scheduling the CPU threads to process the subgraph extraction tasks. We compare it with two alternative policies, namely, no-parallel and FIFO-parallel scheduling. We measure the P99 latency of subgraph extraction using three policies and report the results in Figure 16. In particular, no-parallel uses a single thread to conduct subgraph extraction for each request and is used in SBP. As shown in Figure 10(b), FIFO-parallel puts the tasks from all requests into a single queue and executes the tasks in the FIFO order. We run the SEAL model on the Papers100M graph and test different throughputs using 6 and 24 threads but the results are similar in other cases. Note that the two subgraphs of Figure 16 use different scales for throughput due to the difference in thread count.

We can observe from Figure 16 that the P99 extraction latency of no-parallel stays almost constant when changing the request throughput and thread count. This is because no-parallel uses a single thread to process each request, and thus its P99 latency is determined by the single-thread running

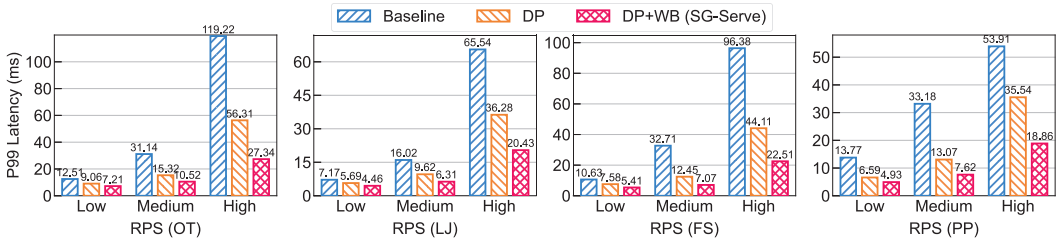


Fig. 17. The P99 model inference latency of SEAL at low, medium, and high request throughput, DP refers to dual-process dispatching and WB means workload-ware batching.

time of the heavy requests. The latency of FIFO-parallel is similar to our task stealing at low throughput but becomes even higher than no-parallel at high throughput. The reason is, at low throughput, there are sufficient threads to process the tasks. Thus, the HoL blocking problem of FIFO-parallel is not significant. However, at high throughput, a single heavy request may occupy all available CPU threads in FIFO-parallel, causing tasks from this request to block not only light tasks but also subsequent heavy tasks, thereby leading to severe HoL blocking. For no-parallel, each request is assigned to a single CPU thread, so a heavy request occupies only one thread while the remaining threads remain available to serve light requests, which allows no-parallel to outperform FIFO-parallel. This also explains why the performance transition throughput for FIFO-parallel and no-parallel is larger when using more threads. To be specific, with only 6 threads available, the shift is observed around 800 QPS, whereas, with a pool of 24 threads, the threshold shifts to approximately 5500 QPS. In comparison, our task-stealing policy consistently achieves lower latency than the two alternative scheduling policies. Since it balances between exploiting multi-thread parallelism and preventing HoL blocking as discussed in Section 4.3.

Subgraph batching and dispatching. For model inference, we adopt two key designs, i.e., dual-process dispatching with MPS and workload-aware batching. To validate the two designs, we incrementally add them to a baseline that utilizes a single GPU process for inference and batches the requests according to request count. We measure the P99 latency of the model inference for the SEAL model on the three graphs and report the results in Figure 17. We set the low, medium, and high request throughputs as 20%, 50%, and 80% of the peak throughput, i.e., the throughput of SG-Serve when the P99 latency is 80ms in Figure 13.

We can observe from Figure 17 that the dual-process dispatching is able to reduce the P99 latency by 1.38x-2.11x on Orkut, 1.26x-1.81x on Livejournal, 1.41x-2.18x on FriendSter, and 1.52x-2.54x on Papers100M, respectively. This is because the dual-process dispatching reduces the chances that heavy requests take up the entire GPU by using two worker processes, and thus the other requests have a better chance to being served within a short time. We can also observe that the workload-aware batching further reduces the latency of dual-process dispatching by 1.25x-2.05x on Orkut, by 1.28x-1.78x on Livejournal, 1.41x-1.96x on FriendSter, and 1.34x-1.88x on Papers100M, respectively. This is because the workload-aware batching identifies the heavy requests and prevents them from prolonging the inference time of the other requests in the same batch. The trend is that the benefit of workload-aware batching is more significant at high request throughput. This is because the requests are more likely to be straggled by heavy requests at high throughput, and thus handling the heavy requests separately yields more performance improvement.

Hot embedding caching. We measure the P99 data communication latency and end-to-end request latency of three models on the Friendster dataset with and without hot-embedding caching, as shown in Figure 18. During the experiments, the request throughput is set to 50% of the peak throughput. We observe that, compared to storing all embeddings in host CPU memory, caching

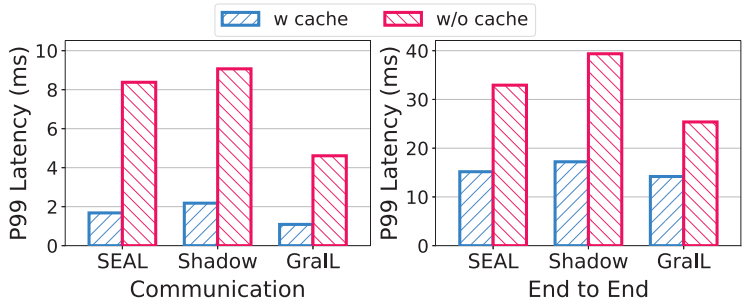


Fig. 18. The P99 latency for CPU-GPU data transfer (left) and end-to-end request serving (right) for the SEAL, Shadow, and GraIL models with and without hot embedding caching on GPU for the Friendster dataset.

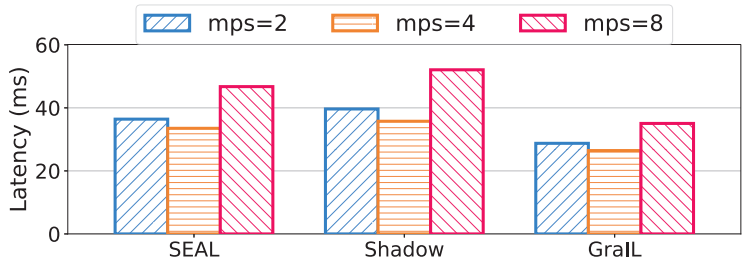


Fig. 19. The P99 latency for SEAL, Shadow, and GraIL on Orkut dataset under 2, 4, and 8 MPS processes.

hot embeddings in GPU memory reduces the data communication latency by up to 5.48× and the end-to-end request latency by up to 2.29×. In particular, the absolute time reductions (i.e., in milliseconds) in end-to-end latency are greater than the reductions in data communication latency. This is because faster data transfers also help shorten the request queuing time. These results show that hot-embedding caching is an effective optimization. We note that all baselines in our main results employ hot-embedding caching, and thus the reported performance gains of SG-Serve come from more sophisticated optimizations.

Varying numbers of GPU processes. We compare the P99 request latencies under 2, 4, and 8 MPS processes, and the results are shown in Figure 19. During the experiments, we set the request throughput to 80% of the peak throughput. We observe that, compared to using 2 MPS processes (the default setting in SG-Serve), using 4 MPS processes provides a marginal latency reduction (within 10%). This is because separating heavy requests from the normal ones already alleviates head-of-line blocking to a large extent. However, when the number of MPS processes increases to 8, the latency rises by up to 1.31× compared to using 2 processes. This degradation occurs because excessive MPS processes reduce GPU execution efficiency due to smaller kernel sizes and increased hardware scheduler overhead. Moreover, using too many MPS processes also consumes additional CPU threads, which slows down subgraph extraction. As a result, we choose to use 2 MPS processes by default as this configuration already reaps most performance gains.

7.4 Performance on Advanced GPU

To evaluate the effectiveness of SG-Serve on more advanced GPUs, we have conducted experiments on a machine equipped with an NVIDIA A100 GPU and an Intel Xeon Gold 6230R CPU with 26 cores. The experiments are conducted with CUDA Toolkit 12.4. We measure the P99 request latency of all systems on Orkut under varying input request throughput, and the results are shown in Figure 20. We can observe that SG-Serve still consistently outperforms all baselines, exhibiting a similar trend to the results obtained with the T4 GPU. In particular, SG-Serve reduces the P99 latency by

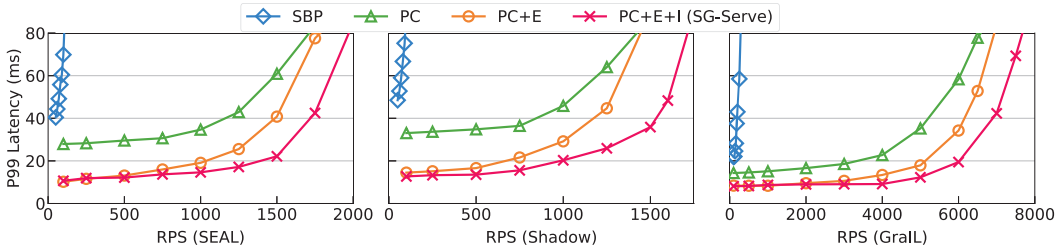


Fig. 20. The P99 latency of SG-Serve and the baselines on Orkut with an A100 GPU at varying input loads.

up to 10.7×, 3.01×, and 1.85× over SBP, PC, and PC+E, respectively, at the same input throughput. These results suggest that the advantages of SG-Serve remain consistent across faster GPUs and newer CUDA Toolkit versions. This consistency stems from the fact that SG-Serve addresses the fundamental challenges of serving SGRL models.

8 Related Work

GNN systems. Many systems are developed to train GNN models, e.g., DGL [72], PyG [30], AliGraph [100], NeuGraph [56], P3 [33], and GNNAdvisor [74]. Some of them accelerate neighbor aggregation, which is the basic operation in GNNs, via optimizing the aggregation pattern [40, 56] or adopting CUDA kernel fusion [21, 77]. Some parallelize GNN training on multiple GPUs and typically focus on reducing the cost of inter-GPU or CPU-GPU communication, e.g., by finding efficient communication plans on NVLink [14, 15, 75], data caching [59, 81], and hybrid parallelism [73, 94, 95]. SG-Serve benefits from optimized neighbor aggregation by using a GNN system (i.e., PyG) for model computation. However, we mainly consider scheduling and parallelizing the requests to achieve a short tail latency while GNN training systems aim to reduce the bulk processing time of large training batches.

Some systems [35, 55, 78, 91] claim to conduct model serving for classical GNN models such as GCN [46] and GAT [41], which produce an embedding for each node in the data graph. However, these systems compute the output embeddings offline, e.g., for model evaluation or preparing downstream tasks. For instance, DGI [85] decomposes GNN models into layers to reduce the cost of data loading from CPU memory to GPU memory when the computing node embeddings. GCNP [96] speeds up embedding computation via dimension reduction, which trades accuracy for efficiency. Traditional GNN models adopt offline computation because they use the node embeddings for downstream tasks, e.g., feeding two node embeddings for link prediction. However, SGRL models are more expressive than classical GNN models due to the joint subgraph extraction for multiple seed nodes [10, 52, 84, 93]. Precomputation is infeasible because the number of possible seed node sets is too large (e.g., 10^{16} node pairs for the 10^8 nodes in the Papers100M graph, and the number of seed nodes can be more than 2). As such, we design SG-Serve to conduct online model serving for SGRL models.

Subgraph Matching. Many systems have been designs for subgraph matching, including GAMMA [64], PBE [37], VSGM [43], Graphflow [58], and Circinus [44]. Given a graph G and a query q, these systems first construct a query graph g_q and then search for all subgraphs of G that are isomorphic to g_q. In contrast, SGRL models typically extract an enclosed subgraph centered around a set of seed vertices. Therefore, the execution patterns of subgraph matching and SGRL-based subgraph extraction are fundamentally different and cannot be applied to serve SGRL models.

SGRL Training. Recent works have explored algorithm–system co-design for efficient SGRL model training, such as GENTI [88], SUREL [84], and SUREL+ [83]. These approaches first propose

new SGRL models with specialized subgraph extraction methods and then optimize the training throughput of these specific models. Consequently, the system optimizations they employ are tightly coupled with their model designs and cannot be easily generalized to accelerate other SGRL models. In contrast, the SG-Serve system proposed in this paper targets the general subgraph extraction procedure of SGRL models and focuses on reducing the latency of serving SGRL models.

Model serving. Machine learning (ML) model serving runs a trained ML model to process user requests, and numerous systems are developed for this purpose due to the widespread adoption of ML models. General-purpose model serving systems such as Triton Inference Server [6], TensorFlow Serving [61], TorchServe [63], and Clipper [24] are widely used in industry. They manage the client requests and schedule their execution on computation engines (i.e., machine learning frameworks) such as Pytorch [62], Tensorflow [8], MXNet [20], and Caffe [42]. These systems perform well for classical machine learning models such as multi-layer perceptron (MLP) and convolution neural networks (CNNs) but lack specialized optimizations for models with special characteristics.

Many model serving systems are designed for specific types of ML models [34, 47, 49, 76, 82, 86, 87, 97–99]. These systems focus on analyzing the unique characteristics of the target models, identifying the core efficiency challenges, and tailoring the system designs to tackle these challenges. For example, BatchMaker [34] observes the recursive computation pattern of recurrent neural networks (RNNs) and thus decomposes RNN requests into cells at the unit for batching to avoid padding all requests to the same length. FastServe [76] exploits the auto-regressive pattern of Transformer and adopts iteration-level preemption to stop the short requests early such that they are not blocked by the long requests. For deep learning recommendation models (DLRM) that work with many embedding entries, MERCI [49] and GRACE [82] precompute the partial aggregations of correlated embedding entries to reduce data access during request processing. Similarly, SG-Serve proposed in this paper is also a specialized serving system for SGRL models, which tailors scheduling and parallelization strategies to handle the unique execution patterns and irregular workload of SGRL requests.

9 Conclusion

We present SG-Serve, an online request serving system for the emerging subgraph representation learning (SGRL) models. We identify the main challenge of SGRL serving as the irregular workload of its requests, i.e., heavy requests have much more computation than regular requests. To achieve a low request processing latency, we tailor system designs for the two stages of SGRL models, i.e., subgraph extraction and model inference stages. For subgraph extraction, we propose API to parallelize the extraction process and schedule task execution for CPU threads with a work-stealing policy, which enjoys parallelism while avoiding head-of-line blocking. For model inference, we batch the requests according to their workload instead of count and run two GPU processes to prevent the heavy requests from monopolizing the entire GPU. Experimental results show that SG-Serve significantly reduces request latency and improves system throughput compared with existing systems.

Acknowledgments

We thank the reviewers for their valuable comments. This work was partially supported by Young Collaborative Research Grant No. C2005-24Y from the RGC of Hong Kong.

References

[1] 2020. GraIL - Graph Inductive Learning. https://github.com/kkteru/grail.
[2] 2020. Link prediction with GCN. https://stellargraph.readthedocs.io/en/stable/d emos/link-prediction/gcn-link-prediction.html.

[3] 2020. Open Neural Network Exchange (ONNX). https://github.com/onnx/onnx.git.
[4] 2020. SEAL – learning from Subgraphs, Embeddings, and Attributes for Link prediction. https://github.com/muhanzhang/SEAL.
[5] 2021. Decoupling the Depth and Scope of Graph Neural Networks. https://github.com/facebookresearch/shaDow_GNN.
[6] 2021. Triton. https://github.com/triton-inference-server.
[7] 2022. Graph Neural Networks: Link Prediction (Part II). https://blog.dataiku.com/graph-neural-networks-link-prediction-part-two.
[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi
[9] Baole Ai, Zhou Qin, Wenting Shen, and Yong Li. 2022. Structure Enhanced Graph Neural Networks for Link Prediction. *CoRR* abs/2201.05293 (2022). arXiv:2201.05293 https://arxiv.org/abs/2201.05293
[10] Uri Alon and Eran Yahav. 2021. On the Bottleneck of Graph Neural Networks and its Practical Implications. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=i80OPhOCVH2
[11] Emily Alsentzer, Samuel G. Finlayson, Michelle M. Li, and Marinka Zitnik. 2020. Subgraph Neural Networks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. https://proceedings.neurips.cc/paper/2020/hash/5bca8566db79f3788be9efd96c9ed70d-Abstract.html
[12] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail Latency Aware Caching - Dynamic Reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 195–212. https://www.usenix.org/conference/osdi18/presentation/berger
[13] Maciej Besta, Raphael Grob, Cesare Miglioli, Nicola Bernold, Grzegorz Kwasniewski, Gabriel Gjini, Raghavendra Kanakagiri, Saleh Ashkboos, Lukas Gianinazzi, Nikoli Dryden, and Torsten Hoefler. 2022. Motif Prediction with Graph Neural Networks. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*. ACM, 35–45. https://doi.org/10.1145/3534678.3539343
[14] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 130–144. https://doi.org/10.1145/3447786.3456233
[15] Zhenkun Cai, Qihui Zhou, Xiao Yan, Da Zheng, Xiang Song, Chenguang Zheng, James Cheng, and George Karypis. 2023. DSP: Efficient GNN Training with Multiple GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023*, Maryam Mehri Dehnavi, Milind Kulkarni, and Sriram Krishnamoorthy (Eds.). ACM, 392–404. https://doi.org/10.1145/3572848.3577528
[16] Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jie Zhang, Yifan Hou, and Xiao Yan. 2019. Grasper: A High Performance Distributed System for OLAP on Property Graphs. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 87–100. https://doi.org/10.1145/3357223.3362715
[17] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. ACM, 32:1–32:12. https://doi.org/10.1145/3190508.3190545
[18] Jiajun Chen, Huarui He, Feng Wu, and Jie Wang. 2021. Topology-Aware Correlations Between Relations for Inductive Link Prediction in Knowledge Graphs. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. AAAI Press, 6271–6278. https://doi.org/10.1609/aaai.v35i7.16779
[19] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*. ACM, 1:1–1:15. https://doi.org/10.1145/2741948.2741970
[20] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Min Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *Proceedings of the NIPS 2015 Workshop on Machine Learning Systems*.

[21] Zhaodong Chen, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. 2020. fuseGNN: Accelerating Graph Convolutional Neural Network Training on GPGPU. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*. IEEE, 60:1–60:9. https://doi.org/10.1145/3400302.3415610

[22] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. 2016. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, DLRS@RecSys 2016, Boston, MA, USA, September 15, 2016*. ACM, 7–10. https://doi.org/10.1145/2988450.2988454

[23] Daniel Crankshaw. 2019. *The Design and Implementation of Low-Latency Prediction Serving Systems*. Ph. D. Dissertation. University of California, Berkeley, USA. https://www.escholarship.org/uc/item/42r093p5

[24] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. USENIX Association, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[25] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 183–198. https://www.usenix.org/conference/atc22/presentation/cui

[26] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2021. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*. ACM, 15. https://doi.org/10.1145/3458817.3476143

[27] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80. https://doi.org/10.1145/2408776.2408794

[28] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. ACM, 621–637. https://doi.org/10.1145/3477132.3483571

[29] Zhihong Fang, Shaolin Tan, Yaonan Wang, and Jinhu Lü. 2023. Elementary Subgraph Features for Link Prediction With Neural Networks. *IEEE Trans. Knowl. Data Eng.* 35, 4 (2023), 3822–3831. https://doi.org/10.1109/TKDE.2021.3132352

[30] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019). arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[31] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1433–1445. https://doi.org/10.1145/3183713.3190657

[32] Mikhail Galkin, Xinyu Yuan, Hesham Mostafa, Jian Tang, and Zhaocheng Zhu. 2024. Towards Foundation Models for Knowledge Graph Reasoning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=jVEoydFOl9

[33] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi

[34] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency RNN inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. ACM, 31:1–31:15. https://doi.org/10.1145/3190508.3190541

[35] Xinyi Gao, Wentao Zhang, Yingxia Shao, Quoc Viet Hung Nguyen, Bin Cui, and Hongzhi Yin. 2022. Efficient Graph Neural Network Inference at Large Scale. *CoRR* abs/2211.00495 (2022). https://doi.org/10.48550/arXiv.2211.00495 arXiv:2211.00495

[36] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[37] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 1067–1082. https://doi.org/10.

1145/3318464.3389699

[38] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 1024–1034. https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html

[39] 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. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. https://proceedings.neurips.cc/paper/2020/hash/fb60d411a5c5b72b2e7d3527cfc84fd0-Abstract.html

[40] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. FeatGraph: a flexible and efficient backend for graph neural network systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*. IEEE/ACM, 71. https://doi.org/10.1109/SC41405.2020.00075

[41] Junjie Huang, Huawei Shen, Liang Hou, and Xueqi Cheng. 2019. Signed Graph Attention Networks. *CoRR* abs/1906.10958 (2019). arXiv:1906.10958 http://arxiv.org/abs/1906.10958

[42] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR* abs/1408.5093 (2014). arXiv:1408.5093 http://arxiv.org/abs/1408.5093

[43] Guanxian Jiang, Qihui Zhou, Tatiana Jin, Boyang Li, Yunjian Zhao, Yichao Li, and James Cheng. 2022. VSGM: View-Based GPU-Accelerated Subgraph Matching on Large Graphs. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*. IEEE, 52:1–52:15. https://doi.org/10.1109/SC41404.2022.00057

[44] Tatiana Jin, Boyang Li, Yichao Li, Qihui Zhou, Qianli Ma, Yunjian Zhao, Hongzhi Chen, and James Cheng. 2023. Circinus: Fast Redundancy-Reduced Subgraph Matching. *Proc. ACM Manag. Data* 1, 1 (2023), 12:1–12:26. https://doi.org/10.1145/3588692

[45] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*. USENIX Association, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[46] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[47] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. *CoRR* abs/2309.06180 (2023). https://doi.org/10.48550/arXiv.2309.06180 arXiv:2309.06180

[48] Darong Lai, Zheyi Liu, Junyao Huang, Zhihong Chong, Weiwei Wu, and Christine Nardini. 2021. Attention Based Subgraph Classification for Link Prediction by Network Re-weighting. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*. ACM, 3171–3175. https://doi.org/10.1145/3459637.3482060

[49] Yejin Lee, Seong Hoon Seo, Hyunji Choi, Hyoung Uk Sul, Soosung Kim, Jae W. Lee, and Tae Jun Ham. 2021. MERCI: efficient embedding reduction on commodity hardware via sub-query memoization. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. ACM, 302–313. https://doi.org/10.1145/3445814.3446717

[50] Boyu Li, Ting Guo, Xingquan Zhu, Qian Li, Yang Wang, and Fang Chen. 2023. SGCCL: Siamese Graph Contrastive Consensus Learning for Personalized Recommendation. In *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining, WSDM 2023, Singapore, 27 February 2023 - 3 March 2023*, Tat-Seng Chua, Hady W. Lauw, Luo Si, Evimaria Terzi, and Panayiotis Tsaparas (Eds.). ACM, 589–597. https://doi.org/10.1145/3539597.3570422

[51] Cangyuan Li, Ying Wang, Cheng Liu, Shengwen Liang, Huawei Li, and Xiaowei Li. 2021. GLIST: Towards In-Storage Graph Learning. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. USENIX Association, 225–238. https://www.usenix.org/conference/atc21/presentation/li-cangyuan

[52] Pan Li, Yanbang Wang, Hongwei Wang, and Jure Leskovec. 2020. Distance Encoding: Design Provably More Powerful Neural Networks for Graph Representation Learning. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. https://proceedings.neurips.cc/paper/2020/hash/2f73168bf3656f697507752ec592c437-Abstract.html

[53] Qika Lin, Jun Liu, Fangzhi Xu, Yudai Pan, Yifan Zhu, Lingling Zhang, and Tianzhe Zhao. 2022. Incorporating Context Graph with Logical Reasoning for Inductive Relation Prediction. In *SIGIR '22: The 45th International ACM SIGIR Conference on Research and Development in Information Retrieval, Madrid, Spain, July 11 - 15, 2022*. ACM, 893–903.

https://doi.org/10.1145/3477495.3531996

[54] Paul Louis, Shweta Ann Jacob, and Amirali Salehi-Abari. 2022. Sampling Enclosing Subgraphs for Link Prediction. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, October 17-21, 2022*. ACM, 4269–4273. https://doi.org/10.1145/3511808.3557688

[55] Mingxuan Lu, Zhichao Han, Susie Xi Rao, Zitao Zhang, Yang Zhao, Yinan Shan, Ramesh Raghunathan, Ce Zhang, and Jiawei Jiang. 2022. BRIGHT - Graph Neural Networks in Real-time Fraud Detection. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, October 17-21, 2022*. ACM, 3342–3351. https://doi.org/10.1145/3511808.3557136

[56] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 443–458. https://www.usenix.org/conference/atc19/presentation/ma

[57] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 135–146. https://doi.org/10.1145/1807167.1807184

[58] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[59] Seungwon Min, Kun Wu, Mert Hidayetoglu, Jinjun Xiong, Xiang Song, and Wen-Mei Hwu. 2022. Graph Neural Network Training and Data Tiering. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*. ACM, 3555–3565. https://doi.org/10.1145/3534678.3539038

[60] Hebatallah A. Mohamed, Diego Pilutti, Stuart James, Alessio Del Bue, Marcello Pelillo, and Sebastiano Vascon. 2023. Locality-aware subgraphs for inductive link prediction in knowledge graphs. *Pattern Recognit. Lett.* 167 (2023), 90–97. https://doi.org/10.1016/j.patrec.2023.02.004

[61] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. *CoRR* abs/1712.06139 (2017). arXiv:1712.06139 http://arxiv.org/abs/1712.06139

[62] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[63] PyTorch Contributors. 2020. TorchServe: Model Serving Framework for PyTorch. https://github.com/pytorch/serve.

[64] Linshan Qiu, Lu Chen, Hailiang Jie, Xiangyu Ke, Yunjun Gao, Yang Liu, and Zetao Zhang. 2024. GPU-Accelerated Batch-Dynamic Subgraph Matching. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 3204–3216. https://doi.org/10.1109/ICDE60146.2024.00248

[65] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*, James Cheney and Thomas Neumann (Eds.). ACM, 1–10. https://doi.org/10.1145/2815072.2815073

[66] Ryan A. Rossi, Rong Zhou, and Nesreen K. Ahmed. 2018. Deep Inductive Network Representation Learning. In *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*. ACM, 953–960. https://doi.org/10.1145/3184558.3191524

[67] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Mag.* 29, 3 (2008), 93–106. https://doi.org/10.1609/aimag.v29i3.2157

[68] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. ACM, 322–337. https://doi.org/10.1145/3341301.3359658

[69] Junshuai Song, Xiaoru Qu, Zehong Hu, Zhao Li, Jun Gao, and Ji Zhang. 2021. A subgraph-based knowledge reasoning method for collective fraud detection in E-commerce. *Neurocomputing* (2021).

[70] Dezhi Sun, Man Hu, Fucheng You, and Han Xiao. 2022. Hybrid Structure Encoding Graph Neural Networks with Attention Mechanism for Link Prediction. In *34th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2022, Macao, China, October 31 - November 2, 2022*. IEEE, 417–424. https://doi.org/10.1109/ICTAI56018.2022.00068

[71] Komal K. Teru, Etienne G. Denis, and William L. Hamilton. 2020. Inductive Relation Prediction by Subgraph Reasoning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 9448–9457. http://proceedings.mlr.press/v119/teru20a.html

[72] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[73] Qiange Wang, Yanfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar: Distributed GNN Training with Hybrid Dependency Management. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 1301–1315. https://doi.org/10.1145/3514221.3526134

[74] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 515–531. https://www.usenix.org/conference/osdi21/presentation/wang-yuke

[75] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin J. Barker, Ang Li, and Yufei Ding. 2023. MGG: Accelerating Graph Neural Networks with Fine-Grained Intra-Kernel Communication-Computation Pipelining on Multi-GPU Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*. USENIX Association, 779–795. https://www.usenix.org/conference/osdi23/presentation/wang-yuke

[76] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Fast Distributed Inference Serving for Large Language Models. *CoRR* abs/2305.05920 (2023). https://doi.org/10.48550/arXiv.2305.05920 arXiv:2305.05920

[77] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chengguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 359–375. https://doi.org/10.1145/3447786.3456247

[78] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing Graph Neural Networks with Message Passing Data Flow Graph. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*. mlsys.org. https://proceedings.mlsys.org/paper/2022/hash/a87ff679a2f3e71d9181a67b7542122c-Abstract.html

[79] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=ryGs6iA5Km

[80] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.* 42, 1 (2015), 181–213. https://doi.org/10.1007/s10115-013-0693-z

[81] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. ACM, 417–434. https://doi.org/10.1145/3492321.3519557

[82] Haojie Ye, Sanketh Vedula, Yuhan Chen, Yichen Yang, Alex M. Bronstein, Ronald G. Dreslinski, Trevor N. Mudge, and Nishil Talati. 2023. GRACE: A Scalable Graph-Based Approach to Accelerating Recommendation Model Inference. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 282–301. https://doi.org/10.1145/3582016.3582029

[83] Haoteng Yin, Muhan Zhang, Jianguo Wang, and Pan Li. 2023. SUREL+: Moving from Walks to Sets for Scalable Subgraph-based Graph Representation Learning. *CoRR* abs/2303.03379 (2023). https://doi.org/10.48550/arXiv.2303.03379 arXiv:2303.03379

[84] Haoteng Yin, Muhan Zhang, Yanbang Wang, Jianguo Wang, and Pan Li. 2022. Algorithm and System Co-design for Efficient Subgraph-based Graph Representation Learning. *Proc. VLDB Endow.* 15, 11 (2022), 2788–2796. https://www.vldb.org/pvldb/vol15/p2788-yin.pdf

[85] Peiqi Yin, Xiao Yan, Jinjing Zhou, Qiang Fu, Zhenkun Cai, James Cheng, Bo Tang, and Minjie Wang. 2023. DGI: An Easy and Efficient Framework for GNN Model Evaluation. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*. ACM, 5439–5450. https://doi.org/10.1145/3580305.3599805

[86] Peiqi Yin, Qihui Zhou, Xiao Yan, Chao Wang, Eric Lo, Changji Li, Lan Lu, Hua Fan, Wenchao Zhou, Ming-Chang Yang, and James Cheng. 2025. CARINA: An Efficient CXL-Oriented Embedding Serving System for Recommendation Models. *Proc. ACM Manag. Data* 3, 3 (2025), 137:1–137:29. https://doi.org/10.1145/3725274

[87] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[88] Zihao Yu, Ningyi Liao, and Siqiang Luo. 2024. GENTI: GPU-powered Walk-based Subgraph Extraction for Scalable Representation Learning on Dynamic Graphs. *Proc. VLDB Endow.* 17, 9 (2024), 2269–2278. https://doi.org/10.14778/

3665844.3665856

[89] Yiwen Yuan, Zecheng Zhang, Xinwei He, Akihiro Nitta, Weihua Hu, Manan Shah, Blaz Stojanovic, Shenyang Huang, Jan Eric Lenssen, Jure Leskovec, and Matthias Fey. 2025. ContextGNN: Beyond Two-Tower Recommendation Systems. In *ICLR 2025*.

[90] Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor K. Prasanna, Long Jin, and Ren Chen. 2021. Decoupling the Depth and Scope of Graph Neural Networks. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 19665–19679. https://proceedings.neurips.cc/paper/2021/hash/a378383b89e6719e15cd1aa45478627c-Abstract.html

[91] Dalong Zhang, Xianzheng Song, Zhiyang Hu, Yang Li, Miao Tao, Binbin Hu, Lin Wang, Zhiqiang Zhang, and Jun Zhou. 2023. InferTurbo: A Scalable System for Boosting Full-graph Inference of Graph Neural Network over Huge Graphs. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 3235–3247. https://doi.org/10.1109/ICDE55515.2023.00248

[92] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 5171–5181. https://proceedings.neurips.cc/paper/2018/hash/53f0d7c537d99b3824f0f99d62ea2428-Abstract.html

[93] Muhan Zhang, Pan Li, Yinglong Xia, Kai Wang, and Long Jin. 2021. Labeling Trick: A Theory of Using Graph Neural Networks for Multi-Node Representation Learning. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 9061–9073. https://proceedings.neurips.cc/paper/2021/hash/4be49c79f233b4f4070794825c323733-Abstract.html

[94] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: Efficient Graph Neural Network Training at Large Scale. *Proc. VLDB Endow.* 15, 6 (2022), 1228–1242. https://www.vldb.org/pvldb/vol15/p1228-zheng.pdf

[95] Chenguang Zheng, Guanxian Jiang, Xiao Yan, Peiqi Yin, Qihui Zhou, and James Cheng. 2024. GE^2: A General and Efficient Knowledge Graph Embedding Learning System. *Proc. ACM Manag. Data* 2, 3 (2024), 183. https://doi.org/10.1145/3654986

[96] Hongkuan Zhou, Ajitesh Srivastava, Hanqing Zeng, Rajgopal Kannan, and Viktor K. Prasanna. 2021. Accelerating Large Scale Real-Time GNN Inference using Channel Pruning. *Proc. VLDB Endow.* 14, 9 (2021), 1597–1605. https://doi.org/10.14778/3461535.3461547

[97] Qihui Zhou, Peiqi Yin, Xiao Yan, Changji Li, Guanxian Jiang, and James Cheng. 2024. Atom: An Efficient Query Serving System for Embedding-based Knowledge Graph Reasoning with Operator-level Batching. *Proc. ACM Manag. Data* 2, 4 (2024), 193:1–193:29. https://doi.org/10.1145/3677129

[98] Qihui Zhou, Peiqi Yin, Pengfei Zuo, and James Cheng. 2025. SparseServe: Unlocking Parallelism for Dynamic Sparse Attention in Long-Context LLM Serving. *CoRR* abs/2509.24626 (2025). https://doi.org/10.48550/ARXIV.2509.24626 arXiv:2509.24626

[99] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. 2022. PetS: A Unified Framework for Parameter-Efficient Transformers Serving. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 489–504. https://www.usenix.org/conference/atc22/presentation/zhou-zhe

[100] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12, 12 (2019), 2094–2105. https://doi.org/10.14778/3352063.3352127

[101] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

[102] Zhaocheng Zhu, Zuobai Zhang, Louis-Pascal A. C. Xhonneux, and Jian Tang. 2021. Neural Bellman-Ford Networks: A General Graph Neural Network Framework for Link Prediction. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 29476–29490. https://proceedings.neurips.cc/paper/2021/hash/f6a673f09493afcd8b129a0bcf1cd5bc-Abstract.html

[103] Yao Zou, Sheng Xiang, Qijun Miao, Dawei Cheng, and Changjun Jiang. 2024. Subgraph Patterns Enhanced Graph Neural Network for Fraud Detection. In *DASFAA 2024*.

Received July 2025; revised October 2025; accepted November 2025