A Distributed-GPU Deep Reinforcement Learning System for Solving Large Graph Optimization Problems

WEIJIAN ZHENG, Purdue University, USA
DALI WANG, Oak Ridge National Laboratory, USA
FENGGUANG SONG, Indiana University Purdue University Indianapolis, USA

Graph optimization problems (such as minimum vertex cover, maximum cut, travelling salesman problems) appear in many fields including social sciences, power systems, chemistry, and bioinformatics. Recently, deep reinforcement learning (DRL) has shown success in automatically learning good heuristics to solve graph optimization problems. However, the existing RL systems either do not support graph RL environments or do not support multiple or many GPUs in a distributed setting. This has compromised the ability of reinforcement learning in solving large-scale graph optimization problems due to lack of parallelization and high scalability. To address the challenges of parallelization and scalability, we develop *RL4GO*, a high performance distributed-GPU DRL framework for solving graph optimization problems. RL4GO focuses on a class of computationally demanding RL problems, where both RL environment and the policy model are highly computation intensive. Traditional reinforcement learning systems often assume either the RL environment is of low time-complexity or policy model is small.

In this work, we distribute large-scale graphs across distributed GPUs, and use the spatial parallelism and data parallelism to achieve scalable performance. We compare and analyze the performance of the spatial parallelism and data parallelism, and show their differences. To support graph neural network (GNN) layers that take as input data samples partitioned across distributed GPUs, we design parallel mathematical kernels to perform operations on distributed 3D sparse and 3D dense tensors. To handle costly RL environments, we design a parallel graph environment to scale up all RL-environment related operations. By combining the scalable GNN layers with the scalable RL environment, we are able to develop high performance RL4GO training and inference algorithms in parallel. Furthermore, we propose two optimization techniques—replay buffer on-the-fly graph generation and adaptive multiple-node selection—to minimize the spatial cost and accelerate reinforcement learning. This work also conducts in-depth analyses of parallel efficiency and memory cost, and shows that the designed RL4GO algorithms are scalable on numerous distributed GPUs. Evaluations on large-scale graphs show that 1) RL4GO training and inference can achieve good parallel efficiency on 192 GPUs; 2) its training time can be 18 times faster than the state-of-the-art Gorila distributed RL framework [34]; and 3) its inference performance achieves a 26 times improvement over Gorila.

CCS Concepts: • **Reinforcement Learning**; • **Optimization Problems over Graphs**; • **Distributed GPU Computing**; • **Open AI Software Environment**;

Additional Key Words and Phrases: Parallel Machine Learning System; High Performance Computing

1 INTRODUCTION

The study of NP-hard optimization problems over graphs (aka graph optimization problems) is one of the most important research areas in computer science with a wide variety of applications including social sciences [7, 23, 29], power systems [2, 5], chemistry [41], and bioinformatics [6, 40]. Researchers have designed three types of

Authors' addresses: Weijian Zheng, Purdue University, Indianapolis, Indiana, USA, 46202, zheng273@purdue.edu; Dali Wang, Oak Ridge National Laboratory, P.O. Box 2008, Oak Ridge, Tennessee, USA, 37831, wangd@ornl.gov; Fengguang Song, Indiana University Purdue University Indianapolis, 723 W Michigan St, Indianapolis, Indiana, USA, 46202, fgsong@cs.iupui.edu.

2329-4949/2023/3-ART $15.00
https://doi.org/10.1145/3589188

algorithms (i.e., *exact*, *approximation*, and *heuristic* algorithms) to solve these problems in practice. The type of *exact* algorithms can find optimal solutions but only work for small-size graphs or graph problems with fixed parameters [11]. The type of *approximation* algorithms have polynomial time complexities and guarantee an upper bound on their approximation ratios, but their solutions are often much worse than the optimal solutions. The type of *heuristic* algorithms utilize certain heuristics to search for optimal solutions, and can generally find high-quality solutions with faster performance than the approximation algorithms.

However, not only do *heuristic* algorithms require domain-specific expert knowledge, but their effectiveness also depends on special properties of graphs. Recently, a few works start to use reinforcement learning (RL) and graph neural network (GNN) based policy models to learn *good heuristics* automatically (instead of manually designed heuristics) [4, 10, 19, 38], which have shown promising results. The rationale behind this methodology is that many real-world applications need to solve the same type of optimization problems repeatedly, in which the problem instances maintain a similar combinatorial structure but having different data. Hence, reinforcement learning techniques can be used in such situations intelligently to exploit the property to learn effective heuristics for different graph optimization problems. For instance, Dai et al. [10], Barrett et al. [4], and Prouvost et al. [38] demonstrate that RL-based algorithms can significantly outperform the state-of-the-art approximation and heuristic algorithms with less execution time (e.g., for details, please refer to Table 3 in Zheng et al.'s work [50], and Fig. 2 and Fig. 3 in Dai et al.'s work [10]).

Since 2015, several distributed reinforcement learning frameworks have been developed to utilize GPU clusters to speed up the learning of policy models. These frameworks use the widely used *data parallelism* (also called *batch parallelism*) to train policy models on distributed GPUs, and can be integrated with an existing or user-defined RL environment. However, they differ from our proposed framework in three aspects. First, their RL environments execute each episode sequentially while ours can support parallel execution of each episode. Second, they use the data parallelism method that requires each GPU own at least one *whole* graph for both RL training and inference, which results in several issues (to be discussed in Section 7.3). Third, although some of the distributed frameworks support executing multiple RL environments in parallel, they only improve throughput but not latency (i.e., each episode takes the same amount of time due to a lack of parallel support in their RL environments).

In this paper, we build a new RL framework for solving graph optimization problems to achieve high scalability especially for large-scale graphs on distributed GPUs, which is called *RL4GO*. The existing RL work on graph optimizations [4, 19, 38, 50] either utilizes one GPU for RL, or does not scale well to address big graph optimization problems arising from real world applications (e.g., an Amazon review graph can have 6 million nodes and 180 million edges [31], a Twitter graph can have 17 million nodes and 476 million edges [47], and more large graphs from different domains [25]). Here, the objective of scalability is to not only minimize the *inference* time for an individual big graph, but also minimize the *training* time on graphs by using many GPUs. So far, achieving high scalability of RL over large graphs has rarely been studied. Besides reducing execution time, another complexity is that the quality of the solution by using many GPUs must be comparable to that of using a single GPU. And the convergence rate on many GPUs should not degrade either. Hence, our many-GPU based RL4GO framework is carefully developed to offer the same solution quality and to converge with the same number of training steps as the single-GPU implementation, meanwhile achieving high performance. We believe that this deterministic behavior should be a desirable feature of any parallel machine learning system whenever possible.

To obtain high scalability in RL4GO, we introduce the following three methodologies:

(1) *Spatial parallelism method devoted to coping with large graphs and enabling scalable reinforcement learning.* Solving large-scale graph-optimization problems is time consuming, which demands an efficient parallelization strategy. To that end, this work focuses on supporting nontrivial big graphs, in which cases RL inference on a single graph may take a long time, or a graph size may exceed a single GPU's memory. We take a holistic approach to parallelizing RL4GO's environment, agent, replay buffer, and policy-model training, going from the environment frontend to the training backend. Every RL stage's data in RL4GO is partitioned across and

computed by all GPUs. Our graph-based RL environment is realized by scalable parallel graph processing. In our implementation, all GPUs use the same set of random seeds to make unanimous decisions and collectively perform the same computing task. This ensures that the result of using multiple GPUs is the same as that of using one GPU. In addition to spatial parallelism, we also use the most popular method of data parallelism (by extending Gorila [34]) to support distributed RL over large graphs. Section 7.3 compares and analyzes the differences between RL4GO and Gorila.

(2) *Parallel GNN kernels designed for a series of 3D sparse-tensor and 3D dense-tensor operations in a distributed setting*. As Section 4.5 will show, a modern message-passing GNN model such as *structure2vec* [9] consists of multiple non-trivial embedding layers and action-evaluation layers, whose operations involve summation, concatenation, multiplication, aggregation, and transformation of mixed 2D and 3D sparse and dense tensors. Note that other message-passing models have a similar message-passing structure. Today, it is still rather challenging to design efficient parallel machine learning kernels due to the following three reasons. First, the existing distributed machine learning libraries (such as PyTorch, Tensorflow) have no support for parallel computing on 3D sparse tensors (however, 2D sparse tensors are supported). Hence we need to design parallel kernels to support it. Second, it takes great efforts to convert a node-level mathematical model to a parallel computing kernel that can compute on many graphs each with many nodes all at once. Third, every data structure is distributed across many GPUs for achieving perfect scaling. When we design our parallel machine learning kernels, we must make sure the parallel forward propagation algorithm is scalable, and the corresponding backward propagation must be correct based on the automatic differentiation engine. A good alternative approach is to develop customized parallel backward kernels (a method used by CAGNET [42]), instead of using automatic differentiation. However, owing to a long list of operators needed by the *structure2vec* based GNN model, we choose to utilize the automatic differentiation method (a native feature provided by PyTorch and Tensorflow) to avoid rewriting many backpropagation kernels.

(3) *Optimization of replay-buffer memory cost, and optimization of multiple-node selection for RL agent*. Graph-based reinforcement learning is different from traditional reinforcement learning in that the traditional RL's experience tuple of (state_{old}, action, reward, state_{new}) often assumes the size of each *state* is small, however graphs have a nontrivial size to store in memory. A typical replay buffer stores tens of thousands of experience tuples. For graph-based reinforcement learning, this requires that a replay buffer keeps tens of thousands of graphs at any time. In Section 5.1, we describe how we design a dynamic graph-state constructor and a few parallel kernels to save space and augment the capacity of graph-based replay buffer. The other optimization is devised to speed up the RL4GO inference process. Traditional RL methods decide only one action at a time by evaluating the agent's policy model once. It takes up to V steps and V model evaluations to find an optimized solution given a graph with V nodes. Given a large-scale graph, this process can be very time consuming. To accelerate the graph RL inference process, we design an adaptive optimization strategy to use one step to simultaneously select multiple nodes, while still maintaining a comparable quality of solution. In Section 5.2, we show how we design the adaptive strategy, and in Section 7.4 we show its performance improvement.

Our prototype of RL4GO framework has an inherently open design, in which users can add new graph optimization problems, new RL algorithms, and new GNN-based policy models, and play with arbitrary combinations of them. We evaluate RL4GO against the state-of-the-art Google Gorila with various types of graphs using many distributed GPUs. For RL training with synthetic graphs, RL4GO is 14.5 times faster than Gorila on 6 GPUs and 18 times faster on 96 GPUs. As for RL inference, RL4GO is P times faster than Gorila for large graphs due to RL4GO utilizing P GPUs simultaneously.

In summary, this paper makes the following contributions:

- We propose an open-design distributed graph reinforcement learning framework, which allows users to add new graph optimization problems, incorporate different embedding models, and do research with emerging

RL algorithms. We demonstrate that RL4GO is scalable, and can reduce the *RL training time* by 31.3 times and *RL inference time* by 54.5 times as the number of GPUs increases from 6 to 192 (in Section 7.5).
- To handle large-scale graph optimization problems, a new spatial-parallelism based RL algorithm (including both parallel GNN training and parallel RL environment) is created (in Section 4). To enable and evaluate the new algorithm in RL4GO, we design and develop efficient parallel ML kernels that work on many GPUs. An analytical performance model shows that the algorithm is scalable as the number of GPUs increases (in Section 6). Not only do we find that our new method is more scalable than the classic data parallelism method, but also our method requires a much smaller minibatch size than the data parallelism method. Both factors have contributed to the 18 times speedup over Gorila in terms of RL training performance.
- A set of parallel numerical kernels are designed for the complex *structure2vec* message-passing GNN model (in Section 4), which can be extended to realize other message-passing GNN models on distributed GPUs.
- We propose two optimization methods: replay buffer memory optimization and multiple-node selection optimization (in Section 5). The first optimization method can reduce the replay buffer space by $O(E)$ times assuming each graph has E edges, and the second optimization method can accelerate the RL4GO inference speed by up to 4.1 times (in Section 7.4).

2 RELATED WORK

There are two classes of works related to our proposed RL4GO work. First, a few distributed RL frameworks have been designed to solve RL problems, which include Gorila [34], Impala [13], BA3C [1], PARL [35], and Ray RLlib [27]. Although some of them work on distributed GPUs, none of them supports graph optimization problems. For the several frameworks that support distributed GPUs, they only support the conventional data parallelism method for RL training, and their RL inference latency is constrained by using only one GPU (e.g., Gorila [34], Ray RLlib [27]). Our work proposes using the spatial parallelism method and designing new parallel kernels for distributed RL, and reveals their advantages over data parallelism on large graph problems in both training and inference scenarios.

Second, there are recent works that employ RL to solve combinatorial optimization problems. However, most of them only support a single GPU, and are not designed to be an extensible open software framework. Barrett et al. [4] develop the ECO-DQN RL algorithm to allow *exploration at test time* to solve Maximum Cut. Tang et al. [39] deploy RL to select *cuts* to solve Integer Programming. Other related works include OpenGraphGym [50], OR-Gym [19], and Ecole [38]. Among them, OpenGraphGym works on multiple GPUs, but cannot handle large graphs whose data exceed a single GPU's memory, and has limited scalability when using multiple GPUs (e.g., using 4 GPUs is faster than 1 GPU by 29%). In addition, the initial release of OpenGraphGym and the ECO-DQN software use dense matrices to represent and compute graph adjacency matrices. However, using dense matrix computations takes much longer time to compute and much more memory space than using sparse matrices. For example, our experimental results show that the new RL4GO system can be up to 160 times faster than OpenGraphGym when the size of a graph is as big as 1,000. As a final point, we should mention that some research efforts try to use supervised deep learning and GNN models to solve graph-related optimization problems [14, 16, 28, 30, 43, 46, 48, 49]. However, they require a large number of problem instances with known optimal solutions for supervised training, and their research approach completely differs from ours in that we use unsupervised distributed reinforcement learning.

3 BACKGROUND

3.1 Applying RL to Solve Graph Optimization Problems

Reinforcement learning (RL) is a process of *trial-and-error* interactions between the agent and the environment [21]. To apply RL to graph optimization problems, we build three major components in the framework: 1) Graph

Environment, 2) Graph Learning Agent, and 3) Policy Model, as shown in Fig. 1. We introduce each of them as follows.

(1) *Graph Environment* is used to simulate a graph optimization problem. Each graph optimization problem needs its own implementation of graph RL environment. Graph Environment takes as input a graph-specific action (e.g., select/delete a node/edge, etc.), executes the action, then returns the updated graph-problem state and a reward.

(2) *Graph Learning Agent* is trained to learn an optimal Policy Model, with which an optimized action can be decided. Two subcomponents must be considered when creating an agent: which RL algorithm and which policy model to use. In an open-design RL framework, different RL algorithms, such as PPO, DDPG, and DQN, can be used by the Graph Learning Agent. On the other hand (orthogonal to the first subcomponent of RL algorithm), users can play with different policy models and embed them in the RL agent.

(3) *A Policy Model* consists of a graph-embedding model and an action-evaluation model. The graph-embedding model is responsible for generating embeddings for a graph. The action-evaluation model takes embedding vectors as input, and decides which node(s) should be selected and added to a graph's partial solution. There are other graph embedding models [9, 15, 37], and users may try different embedding models than the one presented in this paper. Moreover, our framework can automatically combine a graph embedding and an action-evaluation model into one policy model as a whole to perform distributed reinforcement learning.

3.2 Existing Distributed RL Frameworks

Recent progress in RL leads to a number of distributed RL frameworks [1, 13, 18, 27, 34, 35]. We briefly introduce the parallel software design used by the distributed RL frameworks (e.g., Google Gorila [34] and Ray RLlib [27]) as follows.

First, they all support running multiple or many agents in parallel. Given N GPUs, they can launch N learning bundles (or groups), each of which includes an independent agent, a full copy of the policy model, an independent environment, and a private replay buffer. Each agent interacts with its own environment during training. At every step, each agent generates its own *experience tuples* and pushes them to its private replay buffer.

Second, during the training time, each agent samples a mini-batch of tuples from its replay buffer to compute a gradient vector using the *data parallelism* method. In the data parallelism method, every mini-batch of training samples is divided into subsets of the mini-batch and distributed across all agents. Each agent only executes on a subset of the training samples to compute the gradients. Then, the gradients of each agent are aggregated and synchronized through either collective communications or parameter servers. Finally, each agent will update its policy model based on the aggregated gradients.

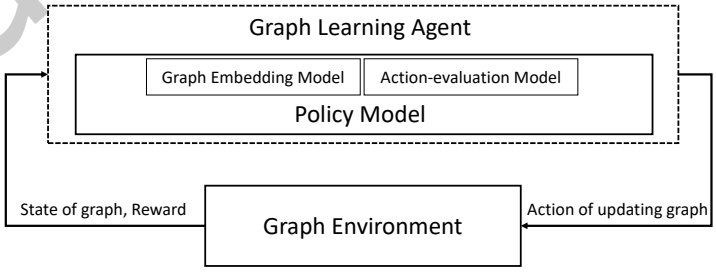


Fig. 1. Major components for applying deep reinforcement learning to graph optimization problems.

However, the existing distributed RL frameworks have some limitations when solving large-scale graph optimization problems. The limitations are described below:

- All their RL environment operations are inherently sequential (i.e., each agent has a sequentially executed environment, and no parallelization supported). For instance, as to Ray RLlib, although each worker can have multiple environment instances to increase throughput, it cannot reduce the latency of each RL environment.
- Given a large number of GPUs, the frameworks are not able to scale up efficiently (e.g., Gorila shows a speedup of 10 times by using 100 GPUs and 30 parameter servers [34] with Atari games).
- Although the most widely used approach of data parallelism is simpler to implement, it is bottlenecked by the serial processing of one graph per GPU in its RL environment and policy model training. Also, due to a similar reason, they do not support large-scale graphs whose size exceeds a single GPU's memory. In this work, we use the spatial parallelism approach to design a distributed RL framework to address the bottlenecks.

To compare the existing distributed RL framework with our proposed RL4GO framework, we develop an implementation based on Gorila, to which we add graph RL environments and the synchronous `all_reduce` communication kernels. Section 7.3 will show a comparison of their performances and an in-depth performance analysis.

3.3 Notation

We now define some notation that will be used throughout this paper.

Variable	Definition	Variable	Definition	Variable	Definition
g	a graph	A	adjacency matrix of a graph	V	nodes of a graph
N	#nodes	S	partial solution of a graph	$N(v)$	neighbor nodes of v
EM	embedding model	Q	evaluation model	$\theta_1 - \theta_7$	model parameters
B	minibatch size	L	#embedding-layers	K	dimension of embedding
P	#GPUs	R	replay buffer size	t	t-th time step for a graph
GPU^i	i-th GPU	A^i	partition of A on GPU^i	S^i	partition of S on GPU^i
v_t	node selected at time t	ρ	edge prob. for ER graphs	d	#connections for BA graphs

Table 1. Notation used in this paper.

4 SYSTEM DESIGN OF RL4GO

This section presents the design of the RL4GO framework. We use the classic Minimum Vertex Cover (MVC) problem as an example to explain the system design. Our current RL4GO prototype also supports the Traveling Salesman Problem (TSP) and Maximum Cut (MC) problem. Since TSP and MC show similar accuracy and performance results as MVC, they are not included in the paper. As a matter of fact, RL environments of TSP and MC are easier to realize than that of MVC since they do not need to update graphs' edges.

The MVC problem is defined as follows: Given a graph $G = (V, E)$, find the smallest set of nodes $S \subseteq V$ such that every edge in E is incident to at least one node in S.

4.1 The Programming Interface

RL4GO provides a simple and generic RL programming interface. A template using the interface is shown in Alg. 1. In the template, users first initialize a Graph Learning Agent that is configured with a pre-defined graph-embedding model and an action-evaluation model. In each *episode*, RL4GO randomly picks a training

graph g, then creates a graph environment *Env* by passing in the name of the graph optimization problem and graph g. At each *step t*, the agent selects a node v_t to be added to the partial solution either randomly or using the policy model. After the environment executes the action v_t, the agent receives a *reward* signal and a termination *done* signal from the environment. After that, the agent adds a new *experience tuple* to its *replay buffer*.

Algorithm 1 A template to apply RL4GO programming interface to graph optimization problems.

Input: Graph_Dataset: a set of training graphs
EM: a graph embedding model
Q: an action-evaluation model

```
1:  B: mini-batch size
2:  R: replay buffer size
3:  /* Create an RL agent using the given policy model EM and Q */
4:  Agent ← GRAPH_LEARNING_AGENT(EM, Q, Replay_Buffer_Size = R)
5:  for each episode e do
6:      Randomly pick a graph g from Graph_Dataset
7:      /* Create a new graph problem environment with graph g */
8:      Env ← GRAPH_ENV(graph_problem_name, g)
9:      for each step t do
10:         v_t = { Select a node randomly, or
                  { Agent.ACT(Env.GET_CURRENT_STATE())
11:         reward, done = Env.STEP(v_t)
12:         /* Push the tuple to replay buffer */
13:         Agent.REMEMBER(Env.GET_PREVIOUS_STATE(), v_t, reward,
                           Env.GET_CURRENT_STATE())
14:         /* Sample a batch of tuples from the replay buffer */
15:         tuples_batch = Agent.SAMPLE(size = B)
16:         /* Apply iterations to train EM and Q */
17:         Agent.TRAIN(tuples_batch)
18:         if (done) break
19:     end for
20: end for
```

To train the policy model (i.e., the *EM* and *Q* models), the agent samples a mini-batch of experience tuples from its replay buffer. Then, the agent performs forward and backward propagations to train the model. When an episode is completed, RL4GO starts another episode by selecting a new graph. RL4GO also provides a higher-level programming interface, in which the two-level nested loop of Alg. 1 is encapsulated into a class named `TrainingRun`. A single line of code `TrainingRun(agent, env, dataset_attr)` can replace the entire two-level nested loop.

4.2 Challenges to Implementing RL4GO

To solve large-scale graph optimization problems using many GPUs, we must parallelize Alg. 1 in an effective way, and address the following issues:

- *How to handle big graph problems where a graph may require more memory than a single GPU can provide?*
- *How to design efficient parallel RL training algorithms on many GPUs?* Parallel RL training may use medium- to large-scale graphs to train a policy model. We need to design a scalable parallel RL training algorithm to do it.

- *How to design efficient parallel RL inference algorithms on many GPUs?* RL inference is different from RL training in that the inference process often targets larger-scale graphs, and its dominant computation may shift from model training to other components (e.g., its graph RL environment). Also, unseen test graphs (one or multiple) are typically at large scales, and will need multiple or many GPUs to store and solve the problem in parallel. Thus, we need to design a scalable RL inference algorithm for many GPUs.
- *How to design optimization techniques to further speed up both training and inference?* We propose an optimization method to minimize the space of replay buffer, and another optimization method (i.e., adaptive multiple-node selection) to reduce the time of parallel RL inference, respectively. Section 5 will present the two optimizations.

4.3 Basic Data Structures

The state of each graph is represented by its adjacency matrix A and current partial solution S. An adjacency matrix A is represented by an $N \times N$ matrix, where N is the number of nodes. In our current prototype, we use the row block data distribution method to partition a graph. More sophisticated methods such as the METIS graph partitioning method [22] can also be added to the framework.

RL4GO is designed to support parallel RL training and inference for big graphs. To handle a number of big graphs at the same time (e.g., during RL training), we stack the graphs together and treat them as a 3D sparse tensor. Given a batch of B graphs and P GPUs, each GPU is assigned with a 3D sub-tensor of dimension $B \times \frac{N}{P} \times N$ for storing the batch's adjacency matrices, and a 3D sub-tensor of $B \times \frac{N}{P} \times 1$ for storing the batch's partial solutions. The set of B graphs may have different numbers of nodes per graph. In our parallel kernel implementation (in Section 4.5), each graph is stored as a sparse matrix A in the COO (Coordinate) format. To enable batched computing on each GPU, the dimension of a smaller graph will be set to the dimension of the largest graph meanwhile the set of non-zero elements in its sparse COO matrix is still the same as before.

Space Cost: As mentioned earlier, each graph's adjacency matrix is stored in the *Sparse COO* format. The amount of memory required for storing one adjacency matrix is $\frac{20N^2\rho}{P}$ bytes on each GPU, where ρ represents the probability that each possible edge is existing or not. Note that $N^2\rho$ is equal to the number of edges. Given a batch of B graphs, the adjacency-matrix tensor consumes $\frac{20N^2\rho B}{P}$ bytes on each GPU. Also, given a replay buffer size of R experience tuples, it takes $8R(\frac{N}{P} + 1)$ bytes to store the R experience tuples on each GPU. As an example, a 16GB GPU can store a graph with up to 400 million edges in the RL4GO framework.

4.4 Designing a Parallel Graph RL Environment

Execution time of RL environments is often considered to be short and not treated with a high priority in traditional RL frameworks. However, as Table 3 will show, the Gorila system can spend 30% of the total execution time on its RL environment for ER graphs with 0.85 million edges. In this subsection, we present how to parallelize the graph RL environment computing operations efficiently.

In RL4GO, a graph RL environment will be initialized with an input graph g that has all the edges, and a partial solution S that is empty at the beginning. As the RL process starts, an RL agent will look at the current graph problem's state (i.e., what is the current partial solution and what is the current graph status), and make intelligent decisions. Whenever an RL agent decides an action (e.g., selecting a new node to be added to the current partial solution to the MVC graph problem), the RL environment immediately updates the graph problem's state and adjacency matrix to denote that the connected edges of the node are not in the graph, as required by the MVC problem. Since a graph's adjacency matrix is distributed across GPUs, we need to develop a parallel graph-state updating method, which will take the following two steps. *At the first step*, each GPU updates its local subgraph by checking whether any of its outgoing/incoming edges is connected to the deleted node. If yes, that edge is deleted.

At the second step, due to those newly deleted edges, some nodes become new isolated nodes (i.e., nodes without edges), which should not be considered as partial solutions. Hence, each GPU starts to search for the newly formed isolated nodes. If the recently deleted node has M neighbors, each GPU takes M iterations to check if each of the neighbor u will become a new isolated node or not. In each iteration, each GPU scans its local subgraph to check if any of its edges (both incoming and outgoing) is incident on the neighbor u. However, given P GPUs, it is possible to take fewer than M iterations since the M neighbors can be scattered across the P GPUs. Eventually, the second step results in a super-linear speedup when the number of neighbors M is a large value. Several experimental results displayed in Section 7 show a super-linear speedup owing to this parallel graph-state updating method. We also want to point out that if we add the hierarchical METIS partitioning method—which tends to divide a graph into clusters or partitions of neighboring nodes—the super-linear speedup described here will happen less frequently and the second step will likely have a linear speedup in our RL4GO framework.

4.5 Implementation of Parallel Numerical Kernels for Graph-Embedding and Action-Evaluation Models

In RL4GO, an agent's policy model consists of two different neural network models: 1) a graph-embedding model, and 2) an action-evaluation model. These two models are connected into a "combined" model, which takes as input the current state of a graph, and yields scores for each node. A modern message-passing model *structure2vec* [9] is used in the current prototype of RL4GO. Other message-passing models [33, 44, 45] often have a similar message-passing structure, and we will extend the current *structure2vec* kernels to add other models to the RL4GO framework. As for a single GPU, other message-passing models can be added to RL4GO directly with little or no modification.

Next, we present how to design and implement parallel kernels for the graph-embedding model and action-evaluation model on distributed GPUs, respectively. As shown in the Notation Table 1, we use B to denote the size of a batch of graphs, K to denote the dimension of graph-embedding vectors, L denote the number of recurrent graph-embedding layers, N the number of nodes of each graph, P the number of GPUs, and $N^i = \frac{N}{P}$ the number of nodes allocated to the i-th GPU (marked by "GPU^i").

1) Parallel Kernels for the Graph-Embedding Model:

The graph-embedding model is defined by mathematical Equation (1):

$$embed_v^L = relu\Big(\theta_1 x_v + \theta_4 \sum_{u \in N(v)} embed_u^{L-1} + \theta_3 \sum_{u \in N(v)} relu(\theta_2 W(v,u))\Big), \tag{1}$$

where $embed_v^L$ is the embedding of node v at the L-th embedding layer, x_v is node v's property, and $W(v,u)$ is the weight on the edge between v and u. θ_1, θ_2, θ_3, and θ_4 are the model parameters to be learned by RL.

The mathematical Equation (1) basically expresses how to decide the embedding for a single node. To handle a large number of nodes from a batch of graphs, we reformulate Equation (1) into tensor operations. We design three parallel kernels that can execute on a number of distributed GPUs, and describe them as follows. The three parallel kernels are named as *NU* (Node Update), *EG* (Embedding Gathering), and *EWG* (Edge-Weight Gathering). During RL execution, all GPUs will call the same kernels, and each GPU^i will compute for $Embed_1^i$, $Embed_2^i$, and $Embed_3^i$ consecutively.

- $Embed_1^i \leftarrow \text{NU}(\theta_1, S^i)$: This kernel corresponds to the mathematical term $\theta_1 x_v$. Argument $S^i \in \mathbb{R}^{B \times N^i \times 1}$ is the current partial solutions to a batch of B graphs stored on the i-th GPU. The kernel first duplicates θ_1 for B times so that the dimension of θ_1 extends from $K \times 1$ to $B \times K \times 1$. Next, it computes a 3D dense tensor and dense tensor multiplication: $Embed_1^i \leftarrow (\theta_1)_{B \times K \times 1} \times ((S^i)_{B \times N^i \times 1})^T$. The output $Embed_1^i$ ($\in B \times K \times N^i$) stores the intermediate embeddings for a $\frac{1}{P}$ portion of graph nodes for B graphs.

- $Embed_2^i \leftarrow \text{EG}(\theta_4, Embed_{pre}^i, A^i)$: It corresponds to the term $\theta_4 \sum_{u \in N(v)} embed_u^{L-1}$. The kernel computes the sum of all the neighbor embeddings of each node. Argument $\theta_4 \in \mathbb{R}^{K \times K}$ is a model parameter. Argument $A^i \in \mathbb{R}^{B \times N^i \times N}$ is a batch of sparse adjacency matrices stored on GPU^i. Due to using the spatial parallelism method, the parallel *EG* kernel divides each adjacency matrix into different GPUs along the matrix row dimension. This way the batch size B does not have to depend on the number of GPUs. In this kernel, the sum of all the neighbors is computed in two steps: 1) Each GPU multiplies a 3D sparse tensor with a 3D dense tensor, i.e., $\text{PartialSum} \leftarrow ((A^i)_{B \times N^i \times N})^T \times (Embed_{pre}^i)_{B \times N^i \times K}$. This step is conducted based on the duality principle between graph operations and matrix multiplication. 2) We use `all_reduce` to compute the final sum of the embeddings across all GPUs (i.e., $Sum_{B \times N \times K}$). Finally, each GPU^i computes $Embed_2^i \leftarrow (\theta_4)_{B \times K \times K} \times (Sum_{B \times N^i \times K})^T$.
- $Embed_3^i \leftarrow \text{EWG}(\theta_2, \theta_3, A^i)$: The kernel aggregates the weights on the edges connected to each node, corresponding to $\theta_3 \sum_{u \in N(v)} relu(\theta_2 W(v, u))$. We first compute the sum over all neighbors by a 3D sparse tensor and dense tensor multiply: $(\text{Sum})_{B \times N^i \times K} \leftarrow (A^i)_{B \times N^i \times N} \times (\theta_2)_{B \times N \times K}$. After *relu*, we compute $Embed_3^i \leftarrow (\theta_3)_{B \times K \times K} \times \text{Sum}_{B \times N^i \times K}^T$.

In the end, we add the three intermediate embeddings (i.e., $Embed_1^i$, $Embed_2^i$, and $Embed_3^i$) to obtain the final embedding for the subset of nodes of B graphs that reside on GPU^i.

Note that given L embedding layers, the above computations will repeat L times by passing the previous embedding layer's output to the next layer.

2) Parallel Kernels for the Action-Evaluation Model:

After getting the above computed embedding result, the action-evaluation model continues to compute scores for all the candidate nodes. This model has three parameters: θ_5, θ_6, θ_7. The equation to compute the score for a single node v is expressed as follows:

$$score_v = \theta_7^T relu[\theta_5 \sum_{u \in V} embed_u \,||\, \theta_6 embed_v], \tag{2}$$

where $[a \,||\, b]$ denotes the operation to concatenate two vectors. To compute scores for many nodes and for a batch of graphs all at once, on distributed GPUs, we design the following three parallel kernels to compute Equation (2). The three kernels are: *ES* (Embedding Sum), *EM* (Embedding Multiply), and *NS* (Node Scores).

- $W_1^i \leftarrow \text{ES}(\theta_5, Embed^i)$: This kernel corresponds to the term $\theta_5 \sum_{u \in V} embed_u$. ES computes the sum of the embeddings of all the nodes for B graphs. It goes through three steps: 1) compute the sum of embeddings for all the local nodes allocated on GPU^i, then 2) use `all_reduce` to get the global sum *sum_embed* from all GPUs, and 3) duplicate θ_5 for B times to form a $B \times K \times K$ tensor, then compute a tensor-tensor multiplication: $W_1^i \leftarrow (\theta_5)_{B \times K \times K} \times (sum\_embed)_{B \times K \times 1}$.
- $W_2^i \leftarrow \text{EW}(\theta_6, Embed^i)$: This kernel corresponds to the term $\theta_6 embed_v$, which computes a tensor-tensor multiplication by $W_2^i \leftarrow (\theta_6)_{B \times K \times K} \times (Embed^i)_{B \times K \times N^i}$.
- $Score^i \leftarrow \text{NS}(\theta_7, W_1^i, W_2^i)$: This kernel calculates the scores for the nodes located on GPU^i. It expands W_1^i from a $B \times K \times 1$ tensor to a $B \times K \times N^i$ tensor. Then, it concatenates W_1^i and W_2^i into a $B \times 2K \times N^i$ tensor. Next, it applies ReLU to the concatenated result. Finally, it extends θ_7 from $1 \times 2K$ to $B \times 1 \times 2K$ and multiplies it with the ReLUed tensor. Its output is a $B \times 1 \times N^i$ tensor, which contains the scores of the partition of N^i nodes of B graphs that reside on GPU^i.

4.6 The Parallel RL4GO Inference Algorithm

Based on the above parallel kernels, we are able to design the parallel RL4GO inference algorithm. Alg. 2 shows the parallel inference algorithm executed by each GPU^i. It takes as input one or multiple graphs, and uses the pretrained embedding model and action-evaluation model to search for an optimal solution for each graph. Each GPU stores a copy of the pretrained model parameters (i.e., θ_1-θ_7).

Algorithm 2 Parallel RL4GO Inference on the i-th GPU $\mathbf{GPU}^i$.

Input: θ_1-θ_4: user-pretrained parameters for the graph embedding model
θ_5-θ_7: user-pretrained parameters for the action-evaluation model
V^i: local subsets of nodes of the test graphs on GPU^i
A^i: local subsets of the adjacency matrices of the test graphs

1: $S^i = \emptyset$: initial partial MVC solutions on GPU^i
2: **for** `step` t = 0 `to` $|V|-1$ **do** ▷ $|V|$: number of nodes in each graph
3: $embed^i = \text{EM}(A^i, S^i, \theta_1 - \theta_4)$ ▷ EM consists of parallel kernels to compute Equation (1)
4: $scores^i = \text{Q}(embed^i, \theta_5 - \theta_7)$ ▷ Q consists of parallel kernels to compute Equation (2)
5: $scores^{all} = \text{All-gather}(scores^i)$ ▷ $scores^i \in \mathbb{R}^{B\times 1\times N^i}$
6: $v_t = argmax_{v\in V} scores^{all}$ ▷ $scores^{all} \in \mathbb{R}^{B\times 1\times N}$
7: S^i += v_t;
8: $A^i \leftarrow$ Update local A^i by removing edges connected to v_t
9: if a graph solution is complete then break
10: **end for**

As shown in Alg. 2, when the RL4GO inference algorithm begins, GPU^i first initializes its local partial-solution subset $S^i \in B\times N^i\times 1$. The inference algorithm takes up to $|V|$ steps to find an optimal solution. In each step t, GPU^i uses the pretrained model parameters (θ_1-θ_7) to predict scores for its local resident candidate nodes (Lines 3-4) by computing the graph-embedding and action-evaluation models. After gathering all scores, the candidate node with the highest score v_t for each graph will be selected and added to the partial solution (Lines 6-7). The local adjacency matrix A^i is also updated accordingly based on the new action of selecting v_t. The parallel inference algorithm terminates when a complete solution is found.

4.7 The Parallel RL4GO Training Algorithm

The design of parallel RL4GO training algorithm is similar to Alg. 1. Given a number of P GPUs, P processes will be launched in parallel. Every process $Proc^i$ utilizes one CPU and one GPU. During RL training, the RL environment operations are executed on CPUs, meanwhile the agent's policy model evaluation and training are executed on GPUs, both in parallel. Each process Proc^i has a copy of the agent's policy model. In Line 15 of Alg. 1, all processes carry out the distributed RL training step, in which Proc^i samples a mini-batch of experience tuples from the replay buffer, and launches a parallel training operation. Due to RL4GO using the spatial parallelism approach, all the processes will work on the same mini-batch of graphs, where each individual graph is split across all GPUs.

In Fig. 2, we use a simple example to illustrate how two parallel processes can conduct one step of RL training. There are three stages in the figure. Stage 1: a new training graph is picked from the training dataset to start a new episode. Stage 2: in the t-th training step, every process selects a node v_t by either *random exploration* or *model-based exploitation*. The unanimously selected v_t leads to a new experience tuple and is added to the replay buffer. Stage 3: each process samples a mini-batch of tuples from the replay buffer, and uses it to train the policy model in parallel. Note that all three stages use the spatial parallelism method, where each graph is partitioned into P GPUs. By contrast, Gorila uses the data parallelism, where each GPU takes one or multiple *whole* graphs.

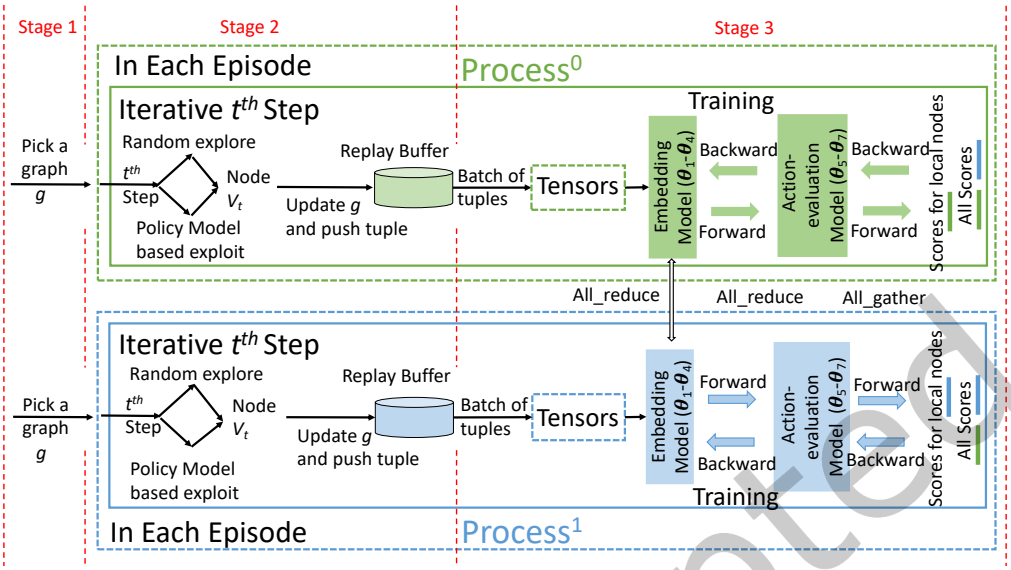


Fig. 2. An example of parallel RL4GO training on two processes. At the beginning of each episode, the agent on each process picks a training graph from the training graph dataset. Then, each agent selects a node V_t randomly or through policy model. Next, the agent updates the corresponding local adjacency matrix A^i, the local partial solutions S^i, and the local set of candidate nodes C^i. The agent then forms a new tuple and pushed it to the replay buffer. Next, the agent samples a mini-batch of tuples and transforms them into 3D tensors of A^i, S^i, C^i, and the target values. Finally, the agent applies multiple gradient descent steps to train the policy model.

5 NEW OPTIMIZATION METHODS

5.1 Optimization of the Replay Buffer Memory

A typical RL replay buffer contains tens of thousands of experience tuples. When solving a large graph problem, an RL agent will see a lot of intermediate graph states, each of which represents a modified graph. It will quickly become too expensive to store many graph states (i.e., many large adjacency matrices). To solve the problem, we store an index pointing to the original graph, and the current partial solution to minimize the memory cost of replay buffer.

A significant amount of memory can be saved because a graph (with E edges) will only need an integer index and a partial solution vector (a few 0's and 1's) to represent it. However, we need to recover the intermediate graph state's adjacency matrix before the training operation starts. To do that, each process uses the experience-tuple's stored partial solution and the original graph's adjacency matrix to generate graphs dynamically at runtime.

This dynamic operation is realized by the *Tuples2Graphs()* function, which converts the information stored in an experience tuple to a concrete adjacency matrix (A^i). Given a batch of B experience tuples, the function converts the batch to a 3D tensor (i.e., a stack of adjacency matrices). Moreover, we stack the lists of partial solution S^i, and target_values from the batch of B tuples to form two additional 3D tensors. Finally, each GPU can start to train the policy model by taking as input the above formed 3D tensors of $batched\_A^i$, $batched\_S^i$, and $batched\_target\_values$.

5.2 Adaptive Multiple-Node Selection Strategy

The parallel RL inference algorithm (as shown in Alg. 2) may take up to $|V|$ steps to solve a graph optimization problem (i.e., it needs up to $|V|$ rounds of policy model evaluations). In each policy model evaluation, only one node (i.e., the one with the highest score) is chosen to become a part of the optimal solution.

To speed up such an essentially "sequential" decision process, we design an adaptive multiple-node-selection optimization strategy that, after all GPUs compute scores for all the candidate nodes, they collectively select d nodes with the top d scores as a subset of the partial solution (Note: the original RL inference algorithm is a special case with $d = 1$).

This adaptive optimization strategy is based upon our observations that, when the size of a graph is large, the set of d nodes selected sequentially (through d steps) is comparable to the set of d nodes with top d scores (at the beginning of the d steps). This way, our optimized RL agent is capable of selecting d best nodes based on 1 evaluation of the policy model, potentially achieving a speedup of d times.

However, we still need to figure out how to choose an appropriate value for d. Selecting a too large value of d can result in a degraded quality of solution due to the decision being overly greedy. Therefore, we introduce an adaptive scheme to gradually decrease the number of d selected nodes per step. The scheme works as follows: when the current candidate node set size $|C|$ is larger than $\frac{N}{2}$, d is set to 8. $|C|$ will become smaller as more nodes become a part of the optimal solution. As $|C| \in (\frac{N}{4}, \frac{N}{2}]$, d is set to 4. As $|C| \in (\frac{N}{8}, \frac{N}{4}]$, d is lowered to 2. When $|C|$ becomes less than $\frac{N}{8}$, d is set to 1. The rationale behind it is that when there are many candidate nodes, we can afford to being aggressive, and gradually we should be conservative when there are fewer nodes left (due to d's bigger impact on smaller graphs). We demonstrate the effectiveness of this optimization strategy in Section 7.4.

6 ANALYTICAL PERFORMANCE MODEL

This section analyzes the parallel efficiency of the RL4GO framework's three major computing components.

(1) Analysis of Computing the Graph-Embedding Model: The estimated execution time of the parallel graph-embedding computation using P GPUs is expressed as follows:

$$T_{em}(B, N, \rho, K, L; P) = \frac{N^2}{P} BK(\rho + L + \frac{2 + K + KL + 3L}{N}) + \alpha L \log_2 P + \beta LBKN, \tag{3}$$

where α is network latency and β is the reciprocal of network bandwidth. There are a number of L `all_reduce` communications, and each has a message of size $B{\times}K{\times}N$. Since the embedding dimension K and minibatch size B are substantially less than the matrix size N for big graphs, each `all_reduce` essentially sends/receives a vector of size $O(N)$.

Because the time of its sequential computation is:

$$T_{em\_seq}(B, N, \rho, K, L) = N^2 BK(\rho + L + \frac{2 + K + KL + 3L}{N}) \tag{4}$$

Its Parallel Efficiency $E_{em}(P) = (\frac{T_{em}(P)}{T_{em\_seq}/P})^{-1} \approx (1 + \frac{\beta P}{N(1+\frac{\rho}{L})})^{-1}$, whose value is almost equal to 1.0 as $N \gg P$. Note that the communication time and its effect on parallel efficiency E are already considered in the formula.

(2) Analysis of Computing the Action-Evaluation Model: The estimated execution time of the action-evaluation computation using P GPUs is:

$$T_{act}(B, N, \rho, K, L; P) = \frac{BKN}{P}(5 + K + \frac{KP}{N}) + \alpha \log_2 P + \beta BK \tag{5}$$

This computation has a single `all_reduce` communication whose message size is $B{\times}K$.

Because the time complexity of its sequential version is:

$$T_{act\_seq}(B, N, \rho, K, L) = BKN(5 + K + \frac{K}{N}) \tag{6}$$

the parallel efficiency of the action-evaluation component is:

$$E_{act}(P) = (\frac{T_{act}(P)}{T_{act\_seq}/P})^{-1} \approx (1 + \frac{P}{cN + 1} + \frac{\beta}{N(K + 5)})^{-1}, \tag{7}$$

where $c = \frac{K+5}{K}$. Since $N \gg P$, the Parallel Efficiency $E_{act}(P)$ is approximately equal to 1.0 too.

(3) Analysis of Executing the Graph RL Environment: In addition to utilizing GPUs, RL4GO also uses a number of P CPUs on the hosts to simulate the graph RL environment, and generates experience tuples for RL training. The RL environment operations computed by the hosts include: 1) Getting a reward signal, 2) sampling experience tuples, 3) tracking and updating the current set of candidate nodes, 4) updating local subgraphs' states, and 5) generating training data based on experience tuples. Among them, the last three operations (i.e., tracking and updating the current set of candidate nodes, updating local subgraph states, and generating adjacency-matrix training data) need more than constant time $O(1)$, and their time complexities are $N^2\rho(\frac{1}{P^2} + \frac{1}{P})$, $\frac{2\rho N}{P}$, and $\frac{2\rho N^2 B}{P}$, respectively.

7 EVALUATIONS

This section describes four different types of experiments, which target showing quality of solutions, comparison between Gorila and RL4GO, improvement by the multiple-node selection optimization, and scalability performance.

7.1 Experimental Setup

Hardware: We conduct experiments on the Summit supercomputer in the Oak Ridge National Laboratory. Each Summit compute node consists of two IBM Power9 CPUs and six Nvidia V100 GPUs with 16GB memory that are connected by NVLink. Each compute node has a host memory of 512GB. All compute nodes are connected by a dual-rail Mellanox EDR 100G Infiniband interconnect.

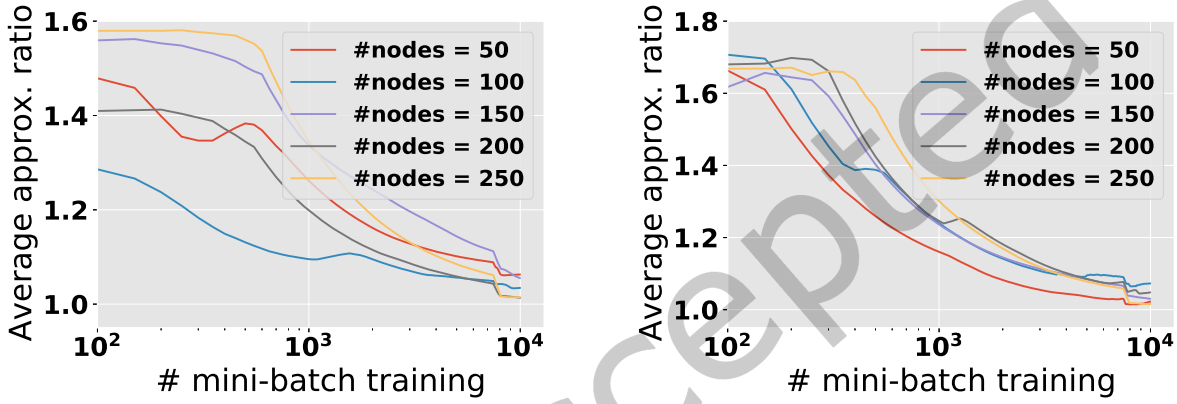
Software: We use PyTorch 1.9.0 [36], NetworkX 2.5.1 [17], and optimization software package IBM-CPLEX 20.1.0.1 [20]. In the experiments, PyTorch is used to implement RL4GO, NetworkX is used to generate synthetic graphs, and IBM-CPLEX is used to compute reference optimal solutions [8] to evaluate the quality of our solutions. IBM-CPLEX is allowed to run up to 30 minutes to find optimized solutions. Here, we want to comment that RL4GO Inference only takes a few seconds to find optimized solutions whose quality is comparable to that of IBM-CPLEX (e.g., it takes RL4GO 22.7 seconds to solve large graphs with 14,400 nodes on 6 GPUs, as shown in Fig. 6.b).

Graph Datasets: We use Erdős-Rényi (ER) graphs [12] and Barabási-Albert (BA) graphs [3] as well as real-world graphs from the Stanford Network Analysis Project (SNAP) [25] to perform experiments. The generation of ER graphs is controlled by the model *ER(n, ρ)*, in which *n* is the number of nodes, and each pair of nodes has a possibility of ρ to be connected with an edge. The generation of BA graphs is controlled by the model *BA(n, d)*. The *BA* model can generate a graph by incrementally adding new nodes to the existing graph. For every newly added node, d edges are connected from the new node to the existing nodes. In addition to synthetic graphs, Table 2 provides information of three real-world graphs used in our experiments.

Hyper-parameter Setting: In our RL4GO experiments, we set the reinforcement learning exploration rate (i.e., the *greedy epsilon* to select a random action) as a decayed rate that decreases from 0.9 to 0.1. We use the Adam optimizer to train the policy model with a learning rate (η) of 1.0e−4. The size of the RL replay buffer is set to

Table 2. Real-world large graphs [25].

	Graph Name		
	Amazon [24]	Berkeley&Stanford [26]	Twitter [32]
#Nodes	403K	685K	81K
#Edges	3.3M	7.6M	1.9M



(a) ER graphs (b) BA graphs

Fig. 3. Quality of RL4GO solutions over ER (a) and BA graphs (b).

50,000. The discount factor (γ) for Bellman Equation is set to 0.9. The number of embedding layers (L) is set to 2. The embedding dimension (K) is set to 16 for the ER and BA graphs, and 64 for real-world graphs.

7.2 Solution Quality of RL4GO

We use a set of unseen test graphs to evaluate the quality of solutions computed by RL4GO on a single GPU. We calculate the *Approximation Ratio* (i.e., the ratio between the solution found by RL4GO and the optimal solution found by CPLEX) for each individual test graph. If Approximation Ratio equals 1, it indicates that RL4GO's solution quality is as good as the optimal solution. We use the *Average Approximation Ratio (AAR)* to denote the average solution quality of a number of test graphs.

Fig. 3 shows the Average Approximation Ratio (AAR) for reinforcement learning with ER graphs and BA graphs, respectively. The graph sizes range from 50 to 250 nodes. From Fig. 3.a, we can see that for different sizes of ER graphs, the RL4GO framework can converge to AAR=1.03 after 10,000 training steps. As for BA graphs (Fig. 3.b), RL4GO can reduce AAR from around 1.70 to 1.02 for different sizes of BA graphs after 10,000 training steps. Furthermore, our RL experiments with larger graphs of 1,920 nodes show that RL4GO can also converge to a good approximation ratio of AAR=1.01 (see Fig. 4).

7.3 Learning Performance on Distributed GPUs: RL4GO vs. Gorila

Next, we compare the difference between RL4GO and Gorila over multiple GPUs. Here, we only show the performance of RL training (not including the RL inference time). This is because Gorila can only effectively utilize one GPU to do RL inference, and hence its RL reference time is overly long (i.e., N times longer than RL4GO on N GPUs).

Experiments Using 1 to 6 GPUs: In Fig. 4, we show the *learning progress* of RL4GO in solid lines, and Gorila in dashed lines, with 1, 2, 4, and 6 GPUs (represented by four different colors, respectively). The training and test graphs are automatically generated ER graphs with 1,920 nodes and around 120,000 edges. As the number of GPUs increases from 1 to 6, both RL4GO and Gorila take shorter training time to reach the same approximation ratio. That is, their lines are shifting from upper-right to lower-left when using more GPUs. For instance, RL4GO takes 53 minutes on 4 GPUs (the red solid line) and 36 minutes on 6 GPUs (the green solid line) to reach AAR=1.01. Using 1 GPU and 2 GPUs takes much longer training time and is not shown in the scope of the figure. When reaching AAR=1.01, using different numbers of GPUs may lead to MVC solutions with little difference due to floating-point roundoff errors and the nonassociativity property of floating point calculations. For instance, Fig. 4 shows that an MVC solution found by 4 GPUs is slightly smaller than that of 6 GPUs (their difference is just a couple of nodes out of hundreds of nodes). At the same time, Gorila demonstrates a performance trend similar to RL4GO when the number of GPUs is from 1 to 4. By contrast, notice that RL4GO only takes 36 minutes to reach AAR=1.01 on 6 GPUs, while Gorila surprisingly needs to take more than 75 minutes on 6 GPUs to converge to AAR=1.01 (the green dashed line). The slowdown of Gorila on 6 GPUs inspires us to continue to examine its performance on an increasing number of GPUs.

Larger Experiments Using 6 to 96 GPUs: Fig. 5 shows the *learning speed* from 6 to 96 GPUs. We conduct this experiment with larger-size ER graphs that have 3,360 nodes. Since Gorila uses the data parallelism method, it requires its mini-batch size B to be at least equal to the number of GPUs (e.g., B is at least 96 when using 96 GPUs). By contrast, RL4GO does not have this limitation, and can utilize any mini-batch size (even B=1). Due to the flexibility of configuring B with RL4GO, we do three different experiments with RL4GO. The first one uses

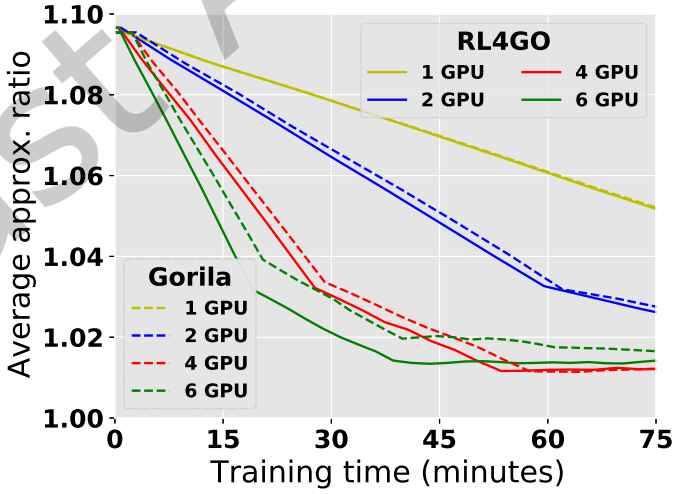


Fig. 4. RL training performance of RL4GO and Gorila.

B=96, which is the same as Gorila. In this way, RL4GO and Gorila have the same amount of training workload since they have the same input size. The second and third experiments use B=16 and B=4, respectively.

As shown in Fig. 5, when B=96, it takes Gorila 8,728s (the red line), and RL4GO 6,279s (the top blue line) to reach the target AAR=1.04 on 6 GPUs. As the number of GPUs increases from 6 to 96, Gorila and RL4GO eventually reduce their RL training time to 2,338s and 468s, respectively. It indicates that RL4GO can outperform Gorila by 5 times when using 96 GPUs and B=96. To understand the reason, we provide a detailed performance analysis in the paragraph after next.

Also in Fig. 5, when the mini-batch size B is configured to a smaller size such as B=16 and B=4, RL4GO takes even less time to reach the target AAR=1.04. *That is, RL4GO is 14.5 times faster than Gorila on 6 GPUs, and 18 times faster than Gorila on 96 GPUs.* We believe that this experimental result shows an advantage of using the spatial parallelism method, because RL4GO does not require B to be at least equal to the number of GPUs. It will be even more beneficial when users run hundreds or even thousands of GPUs. In fact, it is also known that using too large a mini-batch size may make neural network training converge slower or diverge.

Detailed Performance Analysis: To understand the performance difference between RL4GO and Gorila, we measure the total execution time (*Total time*), the time spent on the RL agent (*Agent time*), and the time spent on the graph RL environment (*Env. time*). *Agent Time* includes the time to convert a mini-batch of experience tuples to 3D sparse/dense tensors, as well as the policy model's training time.

Table 3 lists the time breakdown for Fig. 5's four curves, which are Gorila (mini-batch size B=96), and RL4GO using three mini-batch sizes B=96, 16, 4.

First, we compare the difference between Gorila (B=96) and RL4GO (B=96). On six GPUs, RL4GO is 1.4 times faster than Gorila (total time: 6279.34s vs 8570.12s). Regarding *agent time*, Gorila takes 5529.29s while RL4GO takes 5964.99s. That is, RL4GO is 7.8% slower than Gorila because RL4GO takes some time to convert a mini-batch of experience tuples to 3D tensors. However, RL4GO's *environment time* is nine times shorter than Gorila. The reason is that RL4GO is capable of using six GPUs concurrently to execute its RL environment operations, while Gorila can only utilize one GPU for its environment operations. In fact, this nine-times speedup is super-linear

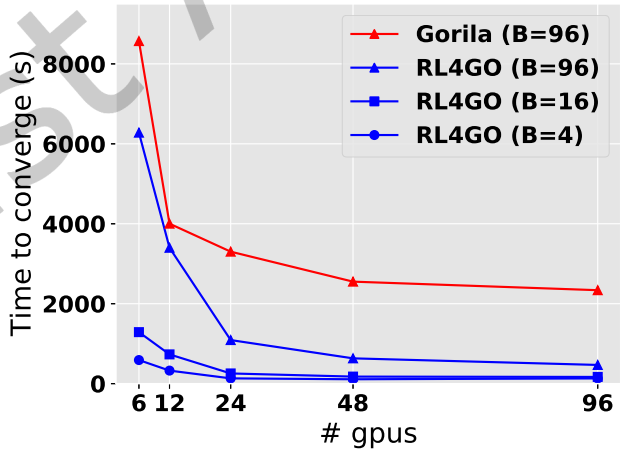


Fig. 5. Evaluation of RL4GO and Gorila on many GPUs with ER graphs. Y-axis is the training time needed to reach an average approximation ratio of 1.04.

(i.e., more than six) since our parallel graph-updating operation (as described in Section 4.4) can take fewer than $\frac{M}{P}$ iterations, where M is the number of neighbors of the selected node and there are P GPUs.

Next, as the number of GPUs increases from 6 to 96 for Gorila (B=96), its total time decreases by 3.7 times (from 8,570.12s to 2,338.09s). Meanwhile, the total time of RL4GO (B=96) decreased by 13.4 times. The reason for achieving a much higher speedup from RL4GO than Gorila is as follows. Gorila's RL environment time is always around 2,000s since it uses one GPU for each of its environment instances (hence having the same latency). However, RL4GO can continuously reduce its environment time when using more GPUs. For instance, its environment time reduces by 3.8 times from 6 to 24 GPUs because of its parallel RL-environment execution (312.81s versus 83.32s). On 48 GPUs, RL4GO's environment time (per learning step) becomes as small as 0.02s. Therefore, when using more than 48 GPUs, RL4GO cannot improve environment-time any further due to Amdahl's law. With respect to the *agent time*, both Gorila and RL4GO with B=96 can scale up efficiently as the number of GPUs increases from 6 to 96.

Table 3. Breakdown of the total execution time of Gorila and RL4GO with different mini-batch sizes (B) on different numbers of GPUs (from 6 to 96 GPUs).

	Number of GPUs				
	6 GPUs	12 GPUs	24 GPUs	48 GPUs	96 GPUs
	Gorila (with mini-batch B = 96)				
Agent time	5529.29	1964.55	918.88	519.2	332.49
Env. time	2813.13	2033.48	1953.32	1999.15	1973.07
Total time	8570.12	4002.35	3301.98	2552.16	2338.09
	RL4GO (with mini-batch B = 96)				
Agent time	5964.99	3191.44	1005.49	551.01	365.28
Env. time	312.81	206.82	83.32	79.24	95.92
Total time	6279.34	3399.75	1090.41	633.12	468.17
	RL4GO (with mini-batch B = 16)				
Agent time	962.53	525.71	168.7	97.75	66.67
Env. time	325.3	205.97	85.88	76.6	91.97
Total time	1288.93	732.21	255.38	176.5	168.36
	RL4GO (with mini-batch B = 4)				
Agent time	274.93	146.29	48.56	29.9	26.37
Env. time	314.43	179.19	82.02	74.85	96.82
Total time	590.43	326.57	132.34	109.96	131.23

The third analysis is to study the effect of reducing the mini-batch size B in RL4GO. Notice that RL4GO does not require that the mini-batch size B be at least equal to the number of GPUs. When B is decreased by n times, the number of floating-point operations per training-step will reduce by n times. As shown in Table 3, when B changes from 96 to 16, RL4GO cuts the agent time by 6.2 times on six GPUs. Since the mini-batch size does not affect the RL environment time, the environment time of RL4GO stays the same. When the number of GPUs is 12, 24, 48, and 96 GPUs, RL4GO can cut its agent time by 6.0, 6.0, 5.6, and 5.5 times, respectively. In the same way, when we change B from 16 to 4, the agent time of RL4GO decreases by another four times. Eventually, by using 96 GPUs, RL4GO (B=4) is 18 times faster than Gorila (B=96).

Discussion: From the above detailed analysis, we can see that Gorila and its used data-parallelism method can speed up the *agent time* in a scalable way. But as the number of GPUs increases, its mini-batch size B needs to increase accordingly. This results in an increasing time complexity for Gorila's agent training step. Differently, RL4GO can avoid this problem by using a smaller B. Another problem is that Gorila and its data-parallelism method only allows for sequential RL environments. By contrast, RL4GO can utilize spatial parallelism to parallelize both its RL environment and RL agent to reduce the total execution time.

7.4 Improvement by Using the Multiple-node Selection Strategy

In the third type of experiments, we use the *adaptive multiple-node selection optimization method* (introduced in Section 5.2) to improve the RL4GO inference time. The experiments are conducted with test graphs of different sizes on 6 GPUs.

Fig. 6.a shows the comparison between the original RL inference algorithm and the multiple-node selection algorithm on ER graphs. Given test graphs with 450 nodes, the original inference algorithm takes 7.9s while the new algorithm takes 2.7s (i.e., 2.9 times faster). The quality of the solutions from MVC_{orig} to $MVC_{multinode}$ remains the same ratio, $\frac{|MVC_{multinode}|}{|MVC_{orig}|}$ = 1.00. When the test graph size increases to 3,600 nodes, the original algorithm takes 265.6s and the new algorithm takes 65.5s (i.e., 4.1 times faster). Their solutions again have a ratio of 1.00. When the test graphs have 7,200 nodes, the new algorithm is 3.9 times faster than the original RL inference algorithm (2473.1 versus 631.4s).

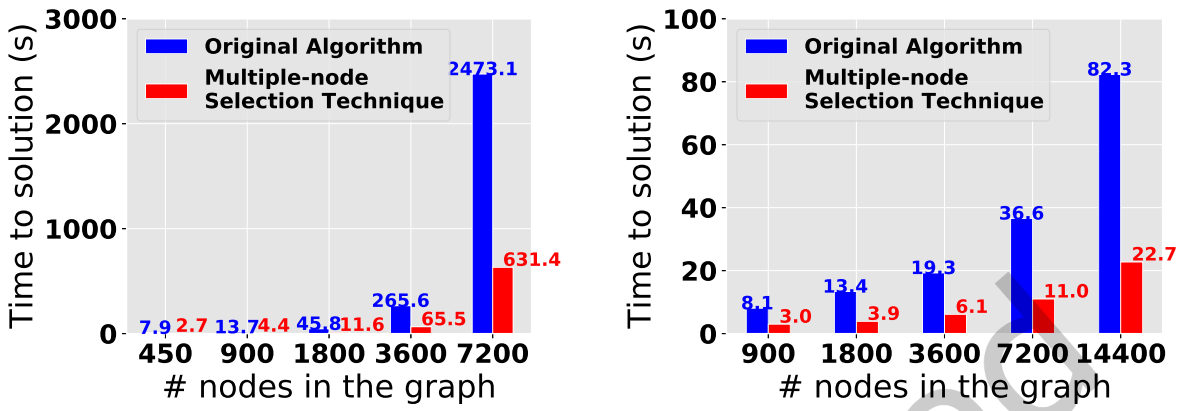
Similarly, in Fig. 6.b, we present the performance improvements on BA graphs by using the proposed optimization method. When BA graphs have 14,400 nodes, the new RL inference algorithm is 3.6 times faster than the original RL inference algorithm, while maintaining a solution ratio of 1.00.

7.5 Scalability of RL4GO

In the last type of experiments, we measure the scalability of RL4GO on distributed GPUs. In particular, we conduct *strong scalability* experiments, in which an increasing number of GPUs are used to solve a fixed-size problem. For all our experiments, we have verified that the numerical result computed by multiple GPUs is the same as that computed by one GPU.

We use the execution time per *inference_step* or *training_step* to measure performance, which is called *"inference or training time-per-step"*, respectively. For RL inference, its time-per-step includes the time to evaluate the agent's policy model, select a node to be a partial solution, and update the graph state. For RL training, its time-per-step includes the time to: 1) explore or exploit, 2) update the graph state, 3) convert a mini-batch of experience tuples to 3D sparse/dense tensors, and 4) train the policy model using the converted 3D tensors. Fig. 7 shows both training and inference time-per-step of RL4GO from 6 to 192 GPUs. We use the solid lines to represent training time-per-step, and dashed lines to represent inference time-per-step.

In Fig. 7.a, we show the RL training time-per-step and inference time-per-step (shown as solid and dashed lines), for ER graphs with 24,000 nodes and 19,200 nodes (shown in red and blue colors), respectively. We first look at the RL training time, which are the two solid lines. For ER graphs with 19,200 nodes, the training time-per-step



(a) ER graphs (b) BA graphs

Fig. 6. Improvement of RL4GO inference time by using the multiple-node selection technique on ER graphs (a) and BA graphs (b).

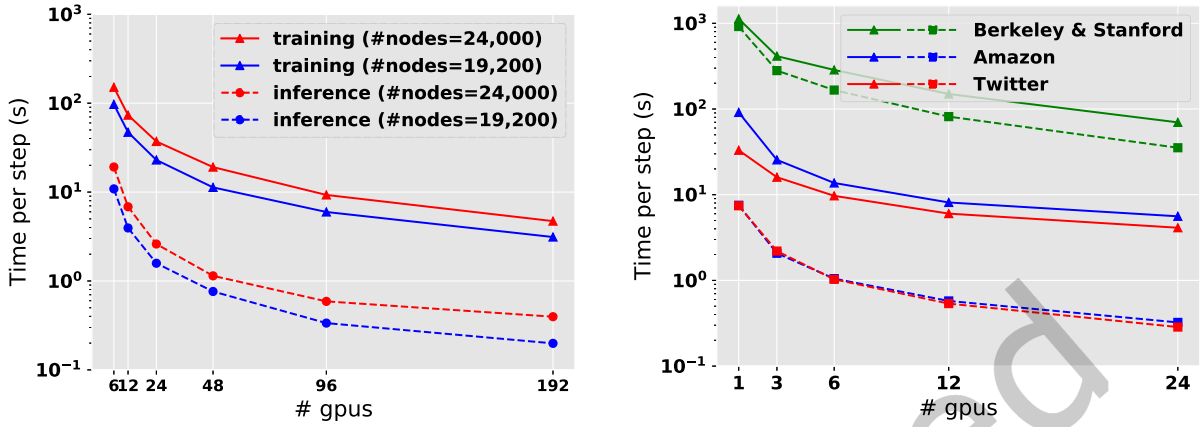
reduces from 96.9s to 3.2s from 6 to 192 GPUs, which is 31.3 times faster. For larger graphs with 24,000 nodes (with 43 million edges), the training time-per-step reduces from 151s to 4.7s, achieving a speedup of 32.1 times.

With respect to the RL inference time, it is shown as two dashed lines in Fig. 7.a: the red one for ER graphs with 24,000 nodes and the blue one for ER graphs with 19,200 nodes. We can see that for ER graphs with 19,200 nodes, the inference time-per-step improves from 10.9s to 0.2s as the number of GPUs increases from 6 to 192. Then, for larger ER graphs with 24,000 nodes, the inference time-per-step improves from 19.2s to 0.4s accordingly from 6 to 192 GPUs.

Fig. 7.b shows the strong scalability experiments for the real-world graphs of Berkeley & Stanford, Amazon, and Twitter. First of all, with the Berkeley & Stanford graph, the RL4GO training and inference time are accelerated by 16.1 and 26 times from 1 to 24 GPUs, respectively. The speedup of RL inference is bigger than that of RL training because RL inference has a higher proportion of super-linear environment operations than RL training. Second, with the Amazon graph, the RL4GO training-time is reduced by 16.3 times and inference-time is reduced by 23.5 times when going from 1 to 24 GPUs. For the third graph of Twitter, we obtain an 8 times speedup in RL4GO training, and 25.6 times in RL4GO inference. In most experiments (except for training with Twitter), RL4GO has achieved significant speedups, providing an average speedup of 16 times in parallel RL training, and a speedup of 24 times in parallel RL inference.

8 CONCLUSION

We presented a high performance parallel reinforcement-learning framework with a generic programming interface to solve large-scale graph optimization problems on distributed many GPUs. Besides introducing the new RL4GO framework with the new computing kernels and parallel algorithms, we also designed two effective techniques to further optimize RL performance. Our theoretical parallel efficiency analysis and memory cost analysis proved RL4GO is efficient and scalable on a large number of GPUs. When applied to large-scale graphs with over 43 million edges, RL4GO reduced the inference and training time significantly with 192 GPUs. Moreover, the comparison between Google's Gorila reinforcement learning system and RL4GO showed that RL4GO can outperform Gorila by up to 18 times.



(a) ER graphs (b) Real-world graphs

Fig. 7. Strong scalability performance of RL4GO. (a) Training and inference time-per-step on ER graphs. (b) Training and inference time-per-step on real-world graphs. Note that solid lines represent RL training time and dashed lines represent RL inference time.

REFERENCES

[1] Igor Adamski, Robert Adamski, Tomasz Grel, Adam Jędrych, Kamil Kaczmarek, and Henryk Michalewski. 2018. Distributed Deep Reinforcement Learning: Learn how to play Atari games in 21 minutes. In *International conference on high performance computing*. Springer, 370–388.

[2] Réka Albert, István Albert, and Gary L Nakarado. 2004. Structural Vulnerability of the North American Power Grid. *Physical review E* 69, 2 (2004), 025103.

[3] Réka Albert and Albert-László Barabási. 2002. Statistical mechanics of complex networks. *Reviews of modern physics* 74, 1 (2002), 47.

[4] Thomas Barrett, William Clements, Jakob Foerster, and Alex Lvovsky. 2020. Exploratory combinatorial optimization with reinforcement learning. In *AAAI Conference on Artificial Intelligence*, Vol. 34. 3243–3250.

[5] Sergey V Buldyrev, Roni Parshani, Gerald Paul, H Eugene Stanley, and Shlomo Havlin. 2010. Catastrophic cascade of failures in interdependent networks. *Nature* 464, 7291 (2010), 1025–1028.

[6] Ed Bullmore and Olaf Sporns. 2009. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature reviews neuroscience* 10, 3 (2009), 186–198.

[7] Wei Chen, Chi Wang, and Yajun Wang. 2010. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1029–1038.

[8] Vasek Chvatal, Vaclav Chvatal, et al. 1983. *Linear programming*. Macmillan.

[9] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*. 2702–2711.

[10] Hanjun Dai, Elias Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*. 6348–6358.

[11] Rodney G Downey and Michael R Fellows. 2013. *Fundamentals of parameterized complexity*. Vol. 4. Springer.

[12] Paul Erdős and Alfréd Rényi. 1960. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci* 5, 1 (1960), 17–60.

[13] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. 2018. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. In *International Conference on Machine Learning*. PMLR, 1407–1416.

[14] Matthias Fey and Jan Eric Lenssen. 2019. *Fast graph representation learning with PyTorch Geometric*. https://arxiv.org/abs/1903.02428

[15] Yoav Goldberg and Omer Levy. 2014. *word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method*. https://arxiv.org/pdf/1402.3722

[16] Graph Nets. 2019. https://github.com/deepmind/graph_nets.

[17] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX.* Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).
[18] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alex Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Caglar Gulcehre, Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. 2020. *Acme: A Research Framework for Distributed Reinforcement Learning.* https://arxiv.org/abs/2006.00979
[19] Christian D Hubbs, Hector D Perez, Owais Sarwar, Nikolaos V Sahinidis, Ignacio E Grossmann, and John M Wassick. 2020. *OR-Gym: A Reinforcement Learning Library for Operations Research Problem.* https://arxiv.org/abs/2008.06319
[20] IBM ILOG CPLEX. 2020. *V20.1.0: User's Manual for CPLEX.*
[21] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.
[22] George Karypis and Vipin Kumar. 1995. METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0. (1995).
[23] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. In *Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining*. 137–146.
[24] Jure Leskovec, Lada A Adamic, and Bernardo A Huberman. 2007. The dynamics of viral marketing. *ACM Transactions on the Web* 1, 1 (2007).
[25] Jure Leskovec and Andrej Krevl. 2023. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[26] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6 (2009), 29–123.
[27] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. 2017. *Ray RLlib: A composable and scalable reinforcement learning library.* https://arxiv.org/abs/1712.09381
[28] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN Training on Large Graphs via Computation-Aware Caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. ACM, 401–415. https://doi.org/10.1145/3419111.3421281
[29] Dean Lusher, Johan Koskinen, and Garry Robins. 2013. *Exponential random graph models for social networks: theory, methods, and applications.* Vol. 35. Cambridge University Press.
[30] 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*. Renton, WA, 443–458. https://www.usenix.org/conference/atc19/presentation/ma
[31] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. 2015. Image-Based Recommendations on Styles and Substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Santiago, Chile) *(SIGIR '15)*. ACM, 43–52. https://doi.org/10.1145/2766462.2767755
[32] Julian J McAuley and Jure Leskovec. 2012. Learning to discover social circles in ego networks.. In *NIPS*, Vol. 2012. Citeseer, 548–56.
[33] Alessio Micheli. 2009. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks* 20, 3 (2009).
[34] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. 2015. *Massively parallel methods for deep reinforcement learning.* https://arxiv.org/abs/1507.04296
[35] PaddlePaddle PARL. 2020. https://github.com/PaddlePaddle/PARL.
[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.
[37] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online learning of social representations. In *ACM SIGKDD international conference on Knowledge discovery and data mining*. 701–710.
[38] Antoine Prouvost, Justin Dumouchelle, Lara Scavuzzo, Maxime Gasse, Didier Chételat, and Andrea Lodi. 2020. *Ecole: A Gym-like Library for Machine Learning in Combinatorial Optimization Solvers.* https://arxiv.org/abs/2011.06069
[39] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. 2020. Reinforcement Learning for Integer Programming: Learning to Cut. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 9367–9376.
[40] Atsushi Tero, Seiji Takagi, Tetsu Saigusa, Kentaro Ito, Dan P Bebber, Mark D Fricker, Kenji Yumiki, Ryo Kobayashi, and Toshiyuki Nakagaki. 2010. Rules for biologically inspired adaptive network design. *Science* 327, 5964 (2010), 439–442.
[41] Nenad Trinajstic. 2018. *Chemical graph theory.* CRC press.
[42] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing communication in graph neural network training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.
[43] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. 2019. *Deep graph library: Towards efficient and scalable deep learning on graphs.* https://arxiv.org/abs/1909.01315

[44] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[45] Danfei Xu, Yuke Zhu, Christopher B Choy, and Li Fei-Fei. 2017. Scene graph generation by iterative message passing. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 5410–5419.

[46] Hongxia Yang. 2019. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 3165–3166.

[47] Jaewon Yang and Jure Leskovec. 2011. Patterns of Temporal Variation in Online Media. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining* (Hong Kong, China) *(WSDM '11)*. ACM, 177–186. https://doi.org/10.1145/1935826.1935863

[48] Jiaxuan You, Zhitao Ying, and Jure Leskovec. 2020. Design space for graph neural networks. *Advances in Neural Information Processing Systems* (2020).

[49] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.

[50] Weijian Zheng, Dali Wang, and Fengguang Song. 2020. OpenGraphGym: A Parallel Reinforcement Learning Framework for Graph Optimization Problems. In *International Conference on Computational Science*. Springer.