GiPH: Generalizable Placement Learning for Adaptive Heterogeneous Computing

Yi Hu [1] **Chaoran Zhang** [1] **Edward Andert** [2] **Harshul Singh** [1] **Aviral Shrivastava** [2] **James Laudon** [3]
Yanqi Zhou [3] **Bob Iannucci** [3] **Carlee Joe-Wong** [1]

Abstract

Careful placement of a distributed computational application within a target device cluster is critical for achieving low application completion time. The problem is challenging due to its NP-hardness and combinatorial nature. In recent years, learning-based approaches have been proposed to learn a placement policy that can be applied to unseen applications, motivated by the problem of placing a neural network across cloud servers. These approaches, however, generally assume the device cluster is fixed, which is not the case in mobile or edge computing settings, where heterogeneous devices move in and out of range for a particular application. To address the challenge of scaling to different-sized device clusters and adapting to the addition of new devices, we propose a new learning approach called **GiPH**, which learns policies that generalize to dynamic device clusters via 1) a novel graph representation **gpNet** that efficiently encodes the information needed for choosing a good placement, and 2) a scalable graph neural network (GNN) that learns a summary of the gpNet information. GiPH turns the placement problem into that of finding a sequence of placement improvements, learning a policy for selecting this sequence that scales to problems of arbitrary size. We evaluate GiPH with a wide range of task graphs and device clusters and show that our learned policy rapidly finds good placements for new problem instances. GiPH finds placements that achieve up to 30.5% better makespan, searching up to $3\times$ faster than other search-based placement policies.

1 Introduction

When running a compute application across a network of computing devices, careful choice of which parts of the application to run on which device can significantly affect application performance. This is particularly true when devices are heterogeneous: e.g., compute-intensive tasks should be run on devices with more computation resources, unless those devices have insufficient communication resources to transmit the task results. Moreover, for applications involving mobile entities (e.g., autonomous vehicles, mobile users) devices can be volatile: they may unexpectedly enter and/or exit the system. Their capabilities may also vary, e.g., due to competing processes or battery drainage. Finding an adaptive placement solution is therefore challenging. Salaht et al. (2020), for example, survey prior works on service placement in edge computing settings, which feature a *heterogeneous* and *dynamic* mix of edge devices, edge servers, and cloud servers.

[1]Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA [2]School of Computing and Augmented Intelligence, Arizona State University, Tempe, Arizona, USA [3]Google Brain, Mountain View, California, USA. Correspondence to: Yi Hu <yihu@andrew.cmu.edu>.

Proceedings of the 6^{th} MLSys Conference, Miami Beach, FL, USA, 2023.

We consider applications in heterogeneous computing that must adapt to device network changes. One example is cooperative sensor fusion for intelligent traffic systems, where vehicles combine sensor data from other cars and traffic cameras for localization and autonomous driving. In general, the applications can be represented as directed acyclic graphs (DAGs), in which each node represents a discrete computation task or sensor input, and edges represent data links between tasks that determine the sequence of computations. Input data enters the DAG and flows through the tasks, e.g., camera images moving through a trained convolutional neural network (CNN) for object detection. We focus on the common objective of minimizing application completion time, e.g., to receive CNN inference results as fast as possible. Our proposed framework, however, can generalize to other objectives, e.g., if one wishes to balance energy cost with completion time.

Challenges: Due to the NP-hardness and combinatorial nature of the placement problem (Topcuoglu et al., 2002), heuristic methods that rely on simple strategies and handcrafted features have been proposed. However, they are often sub-optimal. Many heuristics also assume an overly simplified performance model (e.g., for task computation and communication times) to enable a closed-form formulation. This paper, instead, follows another recent line of

work that automatically learns highly efficient placement policies with reinforcement learning (RL), which learns directly from simulated or real runtime environments.

RL also has the potential to generalize across different problem instances, which can reduce the need for re-training when new applications need to be placed on new clusters of devices. In practice, *a wide range of applications may run on constantly changing networks of devices*, e.g., training a federated learning model (Tu et al., 2020), rendering augmented reality holograms (Wang et al., 2018), or analyzing camera video feeds (Hoque et al., 2021), all of which may be (partially) run on mobile devices like phones or vehicle-mounted cameras whose availability changes over time as they move (Salaht et al., 2020). It is thus imperative to design RL representations and learning algorithms that *can adapt placements to new device clusters*. Existing approaches cannot do so as they either assume all device characteristics are fixed or a fixed number of devices. As a result, whenever the device network changes, the learned policies will perform poorly without significant re-training. In heterogeneous computing environments, placement feasibility constraints may also exist due to hardware functionality, which in general can be challenging to handle in RL.

In this work, we propose **GiPH**, an RL-based approach to learning efficient and fully Generalizable Placement with the ability to adapt to dynamic Heterogeneous networks. To the best of the authors' knowledge, GiPH is the very first RL approach to learn a placement policy that not only generalizes to new task graphs that are not in the training set, but also explicitly adapts to changing device networks with the ability to *relocate* the tasks. While we consider minimization of completion time, GiPH generalizes to other performance objectives. After reviewing related work in Section 2, we outline the **key contributions** and findings of this work as follows:

- We formulate application placement as a *search problem* where the placement is found through applying a sequence of iterative task relocations that improve upon a given placement (Section 3).
- We propose **GiPH**, an RL-based framework to learn *generalizable* placement policies. The learned policy efficiently searches for good placement of a task graph on a heterogeneous device cluster using a novel and *universal* graph representation called **gpNet** that explicitly encodes task-level and device-level information (Section 4).
- We *evaluate GiPH* in terms of the placement quality and generalizability. GiPH finds placements that achieve up to 30.5% lower completion times, with higher search efficiency than prior work. It is comparable to HEFT, a state-of-the-art but slow (Kiamari & Krishnamachari, 2021) heuristic specific to completion times (Section 5).
- We present a case study on applying GiPH to a complex real-world application using cooperative sensor fusion and demonstrate the practical effectiveness of GiPH.

2 Related Work

Application placement is relevant to a variety of domains. GiPH is inspired by the line of work on efficient device placement for distributed neural network training with RL (Mirhoseini et al., 2017; 2018; Zhou et al., 2019; Paliwal et al., 2019; Addanki et al., 2019), where a policy for optmizing the device placement is trained through repeated trials. Some of these works (Mirhoseini et al., 2017; 2018; Zhou et al., 2019) use a recurrent neural network or an attention network to predict a placement for each task and combine them into a placement of the whole graph. Placeto (Addanki et al., 2019) uses graph embedding with RL and searches for better placements through a sequence of iterative placement improvements. It is the closest prior work to ours and one of our comparison baselines (Section 5).

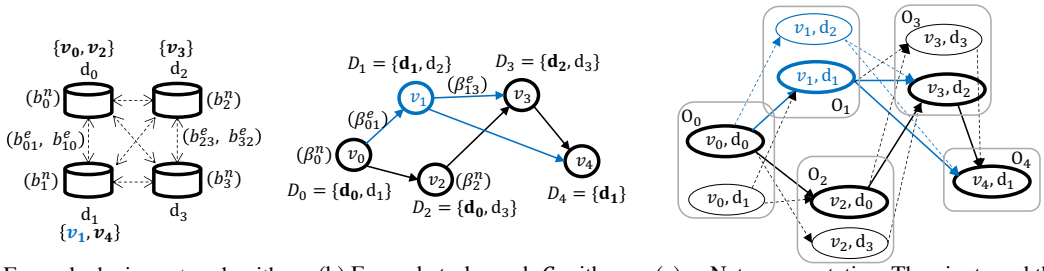
These RL methods for device placement have two limitations: 1) they consider a fixed device cluster; 2) the cost of communicating between devices is not directly considered. As a result, they generalize poorly to new device clusters.

Many prior works have also focused on specific types of systems. GiPH, instead, solves *general* placement problems without domain-specific assumptions. For example, unlike fog/edge computing placement (Nayeri et al., 2021; Salaht et al., 2020) with IoT (Pallewatta et al., 2022; Goudarzi et al., 2021) that assumes a layered or hierarchical network structure, GiPH can be applied to any network topologies. Yan et al. (2020); Dolati et al. (2019) solve the virtual network embedding problems assuming a relatively static physical network with a fixed number of devices, while GiPH allows topological changes and targets application contexts that are more dynamic (i.e., physical nodes can move and dynamically enter or exit the network).

We consider the objective of **makespan (i.e., completion time) minimization**. The Heterogeneous Earliest Finish Time (HEFT) (Topcuoglu et al., 2002) scheduling algorithm is a highly efficient scheduling algorithm that combines task prioritization with a heuristic device placement strategy, and is another one of our baselines. More recent work (Mao et al., 2019; Luo et al., 2021) learns efficient online schedules with GNNs and RL.

3 Placement in Heterogeneous Computing

We consider a general placement problem in heterogeneous computing that aims to properly assign each task in a distributed application to a device in a target computing net-



(a) Example device network with symmetric communication links

(b) Example task graph G with placement constraints

(c) gpNet representation. The pivots and the edges between pivots are identified in bold.

Figure 1. (a) Device network and (b) task graph of an example placement problem. The set of feasible devices for each task (placement constraints) is shown in (b). Node and edge features are in parentheses. (c) shows the gpNet representation for (a) and (b).

work to optimize a performance criterion. We consider a distributed computing environment with 1) heterogeneous compute capabilities, 2) heterogeneous communication capabilities, and 3) placement constraints.

A **target computing network** (e.g., Fig. 1(a)) consists of a cluster $D = \{d_0, ..., d_{m-1}\}$ of interconnected devices. The compute devices in the system can be CPUs and/or GPUs that run at different speeds. For example, a CNN can be much faster on a GPU than a CPU, and its actual running time can differ on different GPU modules. Devices have *different compute capabilities* (compute features) depending on the processor types, clock rates, etc. We model the compute capability of a device d_i by a vector of compute features b_i^n (i.e., device compute speed, hardware type). There are also *placement constraints* resulting from hardware functionality and feasibility of devices, e.g., the task of acquiring LIDAR data can only be placed on a LIDAR sensor but not on a camera or an edge server.

We also consider a general *heterogeneous communication network*. Devices can have different communication mechanisms (wired connection or wireless networks) to send data to each other. The specific means of communication defines the speed of data transmission, which can be different between pairs of devices. For example, a camera can be directly wired to a server that communicates wirelessly with mobile phones. We sometimes call a device cluster a device *network* to emphasize the role of device connectivity in application performance. Each pair of devices (d_i, d_j) has communication link features b_{ij}^e (e.g., bandwidth, delay). For the purpose of this paper, we assume the devices are fully connected and we only consider single-link paths (represented by edges) between devices. It is easy to generalize to more complex topologies by attaching very high communication losses to links that do not exist.

A distributed application is defined by a directed acyclic **task graph** $G = (V, E)$, as shown in Fig. 1(b), where nodes $V = \{v_0, ..., v_{n-1}\}$ represent computation tasks of the application and edges $E \subset V \times V$ represent inter-task data dependencies and communication. We define *parents* of a task v as those tasks on which v has a data dependency, i.e., $\{u|(u, v) \in E\}$, and reversely v is a *child* of its parent. Edges capture the precedence constraint that a child can only start after it receives all data inputs from its parents. The features of each node $v_i \in V$ and each edge $(v_i, v_j) \in E$, represented by β_i^n and β_{ij}^e, respectively, should be defined according to the optimization objective, e.g., to minimize the completion time, node and edge features may include the amount of compute of each task and data of each link.

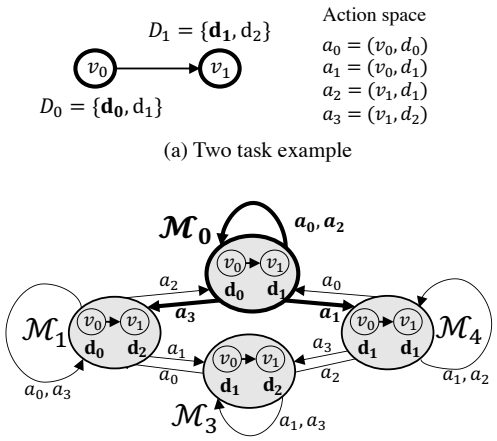
Given a distributed application $G = (V, E)$ and a device network N, a **placement** maps each task in the application to a device in the network $\mathcal{M} : V \to D$. Each computation task v_i can only be mapped to a subset of devices $D_i \subseteq D$ (Figs. 1(a) and (b)) due to placement constraints. The goal of the placement is to optimize a *performance criterion* $\rho(\mathcal{M}|G, N)$ while satisfying $\mathcal{M}(v_i) \in D_i$ for all $v_i \in V$. (G, N) defines a specific problem instance, and we denote a general placement as a triple $\mathcal{P} = (G, N, \mathcal{M}^{G \to N})$.

We take performance to be the **makespan** (i.e., completion time) of an application, which is the time duration from the start of the first task's execution to the end of the last task's execution for a given application input. To simplify the problem, we assume the makespan is input-agnostic, e.g., CNN inference will have similar latency for any image.

Due to the data dependencies that exist among tasks, a child task can only start after it receives all the data from its parents. Therefore, task execution must follow the precedence constraints defined by the application DAG (i.e., a partial order). If we associate each node i with a cost c_i, and each edge (i, j) with a cost c_{ij}, representing the computation and communication latency, respectively, the objective is

$$\min_{\mathcal{M}} \rho(\mathcal{M}|G, N) = \min_{\mathcal{M}} \max_{p \in P(G)} \left(\sum_{i \in p} c_i + \sum_{(i,j) \in p} c_{ij} \right),$$

where $P(G)$ is the set of all paths from an entry node to an exit node in graph G, and p is composed of all nodes and



(a) Two task example

(b) MDP state action diagram for the example in (a).

Figure 2. MDP (Markov decision process) of the placement search for a 2-task graph on 3 devices.

edges along the path. (The dependence of c_i and c_{ij} on $\mathcal{M}$ is omitted for simplicity.) Given a placement, the makespan is the total cost along the critical path (i.e., the path with the highest cost), which can be determined in $\mathcal{O}(|E| + |V|)$ time by traversing the graph in topological order.

This placement problem is NP-hard, which means finding an exact optimal solution requires exhaustive search. A simpler version of the problem, which assumes unit-time computation and no communication delay, has been proven to be NP-complete (Ullman, 1975). Therefore, a simple one-step classification approach, where each node's "class" is the device on which it is placed, is not a viable solution. Like most prior work, GiPH tackles this challenge by considering task placement individually instead of attempting to determine the placement of the entire graph all at once, making the problem more tractable.

4 GIPH

This section introduces GiPH, an RL method that improves placement by relocating tasks. In §4.1, we describe the formulation of the search problem and the associated Markov decision process (MDP) that underlies our RL approach. In §4.2, we propose GiPH, which utilizes a novel gpNet representation to encode task-level and device-level features, along with a scalable neural network design that summarizes the graph information and makes relocation decisions.

4.1 Markov Decision Process Formalism

We formulate the placement problem as a search problem, where given an initial placement, a learned policy iteratively relocates some of the tasks. Through making these incremental changes, the policy is able to search through the solution space and find better placements. Instead of trying to learn a policy that places the whole graph at once, our search approach makes the learning simpler by only considering a small local search space at a time. Focusing on incremental changes further allows us to generalize to arbitrary application task graphs and device clusters (§4.2).

State: Consider a single problem instance (G, N). For the search problem, we define the *state space* as the set of all feasible placements $S_{G,N} = \{\mathcal{M} | \mathcal{M}(v_i) \in D_i, \forall v_i \in V\}$. (*State* and *placement* terms are interchangeably used in the paper.) The size of the state space $|S_{G,N}| = \prod_{i=0}^{|V|-1} |D_i|$, since each task can be placed on any of the feasible devices for it. When there is no placement constraint (i.e., $D_i = D$ for all tasks), $|S_{G,N}| = |D|^{|V|}$. For the two-task example shown in Fig. 2(a), there are a total of 4 feasible placements, all shown as states in the transition diagram of Fig. 2(b).

Action: Observing that different placement configurations (states) differ in the placements of specific tasks, we define an *action* to be a task and device pair $(v_i, d_j) \in V \times D$ that places v_i on device d_j. By including task selection in the action, we allow the policy to relocate tasks in any order and relocate for more than once. In this way, the search-based policy can go back and adjust the placement of the same task again after moving other tasks, exploring the state space. This is different from Placeto, which traverses each node in the graph only once and in an arbitrary order, impeding its ability to fully explore possible placement options.

We only consider feasible actions (v_i, d_j) such that $d_j \in D_i$. The size of the action space is thus $|A_{G,N}| = \sum_{i=0}^{|V|-1} |D_i|$, which is $|V||D|$ if there are no placement constraints. Fig. 2(b) lists all four actions for the simple two-task example and shows the deterministic state transition given an action taken at each state. Note that the diameter, i.e., the length of the longest shortest path between any two states in the state transition diagram, is $|V|$ because one can always change from one placement to any other placement by moving each task node at most once. Therefore, even though the size of the state space grows exponentially with $|V|$, we can reach any state from any other state in $|V|$ steps.

Reward: The *objective function* $\rho(\mathcal{M}|G, N)$ reflects how good a state $s = \mathcal{M}$ is. We assign intermediate reward $r_t = \rho(s_{t+1}|G, N) - \rho(s_t|G, N)$, which mimics the advantage function (Sutton & Barto, 1998) indicating the performance improvement after taking an action a_t at a given state s_t. The goal of RL is to learn to take actions in order to maximize the expected return $\sum_{t=0}^{T} \gamma^t r_t$, where T is the episode length. When $\gamma = 1$, the expected return is the expected performance improvement between the final state s_T and an initialized state s_0, i.e., the policy tries to maximize $\mathbb{E}[\rho(s_T|G, N)] - \mathbb{E}[\rho(s_0|G, N)]$. When the distribution of

the initial placement is fixed, the latter term is constant for a (G, N) pair and RL is effectively improving the expected performance of the final placement through maximizing the expected return. When $\gamma < 1$, the policy seeks more immediate reward as future rewards are discounted. In this case, the policy learns to search more efficiently (i.e., increase the reward the most) at the beginning of the RL episodes.

4.2 GiPH Framework

By leveraging the MDP, GiPH can learn to iteratively optimize a placement by relocating tasks. The framework of GiPH is shown in Fig. 3. Given a placement problem of an arbitrary task graph G_i and target network N_i, the placement agent starts a search from an initial placement $\mathcal{M}_0^{G_i \to N_i}$. This initial placement can be generated using some simple strategies, or it can be a placement that requires improvement. Following the MDP, at each step, the agent takes as input the current state of the search, $s_t = \mathcal{M}_t^{G_i \to N_i}$, decides a task relocation step $a_t \in A_{G_i,N_i}$ that modifies the current placement to s_{t+1}, and observes the improvement of the objective ρ as the reward r_t.

GiPH comprises three key components. Firstly, the novel graph representation gpNet (§4.2.1) explicitly encodes both task-level and device-level features of the current placement state. This facilitates the learning of a fully generalizable policy that can be applied to different placement problems. Secondly, a graph neural network (§4.2.2) computes an embedding for each action (i.e., a task relocation step) based on the gpNet representation. The GNN leverages the graph structure of the placement problem to summarize relevant information and generate high-quality embeddings. Finally, a policy network (§4.2.3) uses a score function to make decisions on which action to take. The GNN and the policy network are jointly trained to optimize the placement policy.

4.2.1 gpNet Representation

We have formulated a discrete MDP for the placement search problem given (G, N). For the learned policy to be fully generalizable across different problem instances, we need a representation for a general placement $\mathcal{P} = (G, N, \mathcal{M}^{G \to N})$. This representation must capture the compute and communication requirements of the task graph G and the compute and communication capabilities of the device network N, enabling the policy to learn the performance function ρ given $\mathcal{P}$. This general representation enables the learned policy to be applied to different placement problems without being specific to a particular task graph or device network, improving the policy's ability to generalize and achieve good performance in various scenarios.

To this end, we present **gpNet**, a novel and universal graph representation of the placement that encapsulates features of both the task graphs and the device networks with placement constraints. gpNet generates a unique graph $H = (V_H, E_H)$ given a general placement $\mathcal{P} = (G, N, \mathcal{M}^{G \to N})$, where $G = (V, E)$ is an arbitrary task graph with node features $\boldsymbol{\beta}^n$ and edge features $\boldsymbol{\beta}^e$, and N is an arbitrary device network composed of a device cluster D with device compute features $\boldsymbol{b}^n$ and communication link features $\boldsymbol{b}^e$. $H \leftarrow gpNet\left(G = (V, E, \boldsymbol{\beta}^n, \boldsymbol{\beta}^e), N = (D, \boldsymbol{b}^n, \boldsymbol{b}^e), \mathcal{M}^{G \to N}\right)$.

Node generation: $V_H = \{(v_i, d_j) | d_j \in D_i \text{ for } i = 0, ..., |V| - 1\}$. Each node in H represents a feasible placement of $v_i \in V$ on device $d_j \in D_i$, and is labeled (v_i, d_j), e.g., Fig. 1(c). The node features x_u^n of a node $u = (v_i, d_j)$ are a function, f_n, of the task features β_i^n and the device features b_j^n, i.e., $x_u^n = f_n(\beta_i^n, b_j^n)$. The set of nodes for all possible placements of a task v_i forms a group of placement options $O_i = \{(v_i, \cdot)\} \subseteq V_H$. Nodes whose labels are in the current placement $\mathcal{M}^{G \to N}$ are called *pivots* and form a set $V_{H,\mathcal{P}} \subseteq V_H$. The subgraph induced by $V_{H,\mathcal{P}}$ thus contains all information about the current placement $\mathcal{M}^{G \to N}$. Non-pivot nodes, on the other hand, represent a potential task *re-placement*. Each node in H also corresponds to one action defined in the search problem described in §4.1.

Edge generation: We add edge (u_1, u_2) with $u_1 = (v_i, d_k), u_2 = (v_j, d_l)$ to the gpNet H if $(v_i, v_j) \in E$ and at least one of u_1 and u_2 is a pivot, i.e., $u_1 \in V_{H,\mathcal{P}}$ or $u_2 \in V_{H,\mathcal{P}}$. In this way, each non-pivot node (v_i, d_j) only has edges pointing to or from pivots that contain the current placement information of its parents and children. For example, in Fig. 1(c), (v_1, d_2) has an incoming edge from (v_0, d_0), which contains the current placement information of v_1's parent task v_0, and outgoing edges to (v_3, d_2) and (v_4, d_1), which contain the current placement information of v_1's child tasks v_3 and v_4. Thus, *non-pivot node (v_i, d_j) has a local graph structure corresponding to v_i being re-placed to d_j*. The edge feature $x_{u_1 u_2}^e$ of $(u_1 = (v_i, d_k), u_2 = (v_j, d_l))$ is a function, f_e, of the data link features β_{ij}^e and the communication link features b_{kl}^e. The resulting graph H has $|V_H| = \sum_{i=0}^{|V|-1} |D_i|$ nodes and $|E_H| = \sum_{i=0}^{|V|-1} (|D_i||E_i|) - |E|$ edges, where $|E_i|$ is the degree of v_i in G. Both f_n and f_e can be any functions that combine the features of the task graph and the device network (e.g., concatenation).

See Appendix B.1 for the full algorithm. Our proposed gpNet unifies a given application graph G and device cluster N into a single graph that *captures all device- and task-related features* for making a placement update decision. The original task dependencies in G and placement constraints are implicitly present in the output gpNet H by construction. gpNet also generalizes to different problem instances: we can construct a gpNet for any placement of an arbitrary task graph-device network problem pair (G, N).

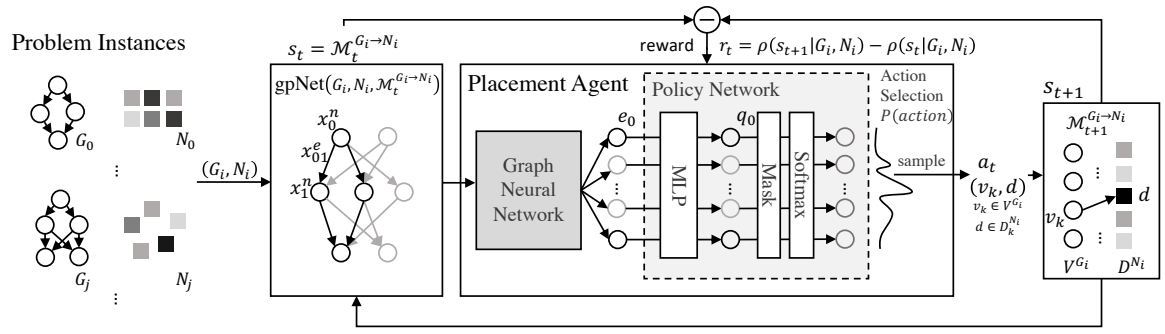


Figure 3. GiPH neural network design. A given problem instance (G_i, N_i) is first transformed into a graphical gpNet representation, which is then passed through a GNN to generate feature embeddings for each feasible task-device placement pair. The policy network then chooses an action (an incremental task-device re-placement) by estimating its performance improvement.

4.2.2 *Scalable and Generalizable Graph Embedding*

GiPH must first convert the placement information, represented in graphical form by $gpNet(G_i, N_i, \mathcal{M}_t^{G_i \to N_i})$, into features that can be passed to the policy network. Creating a flat vector representation is not scalable because it cannot handle graphs of arbitrary sizes and shapes (which depend on the specific task graph, target network and constraints).

GiPH achieves scalability using a graph neural network (GNN) (Dai et al., 2017; Battaglia et al., 2018) that embeds the state information in a set of embedding vectors. Taking a gpNet as input with node features $\boldsymbol{x}^n$ and edge features $\boldsymbol{x}^e$ composed as described in §4.2.1, GiPH propagates information in a sequence of message passing

$$e_u = h_2 \left(\sum_{v \in \xi(u)} h_1 \left([e_v \parallel x_{vu}^e] \right) \right) + x_u^n, \qquad (1)$$

for each node u in the gpNet, where $\xi(u)$ is the set of parents of u, who have aggregated messages from all of their parents. $h_1(\cdot)$ and $h_2(\cdot)$ are non-linear transformations over vector inputs with trainable parameters. The message passing is done in both forward and backward directions with separate parameters, each summarizing information about the subgraph of nodes that can be reached from u and nodes that can reach u. GiPH concatenates the two summaries along each direction as the node embeddings. For a node with label (v, d), this embedding thus captures the local placement information if v is placed on d (i.e., if an action (v, d) is taken). Adopting a GNN also helps generalizability because it automatically learns high-level features that are statistically important through end-to-end training, and the model learned can generalize (and scale) to unseen graphs.

4.2.3 *Policy Network and Actions*

The policy network consists of a multi-layer perceptron (MLP), an optional mask layer, and a softmax layer (Fig. 3). We use the per-node embedding from the GNN to compute a score $q_a = g(e_a)$ for each action a in the action space A_{G_i,N_i} (represented as nodes in the gpNet). $g(\cdot)$ is a score function implemented as a MLP that computes a scalar value for an embedding vector. The score q_a quantifies how good an action is given the current state s. GiPH then uses a softmax layer to output a probability of selecting each action based on the score $P(a|s) = \exp(q_a) / \sum_{b \in A_{G_i,N_i}} \exp(q_b)$. An optional mask layer can be placed before the softmax to mask out undesired actions. The final output is a probability distribution over all feasible actions.

Since we may have input gpNets of arbitrary sizes, instead of using a fixed-size policy network, the GiPH policy network adopts a score function to evaluate individual actions (i.e., nodes in gpNet), based on their local graph connectivity and encoded placement information. Thus, it *does not depend on the size of gpNet*.

To improve the sample efficiency and force exploration, we mask out actions that do not change the current placement (e.g., a_0, a_1 at state $\mathcal{M}_0$ in the example Fig. 2(b)) because no new information will be acquired by taking those actions. We also mask out actions that will result in moving the same task consecutively twice because we expect the policy to find a better device for a task within one move.

5 Evaluation

In this section, we evaluate the performance of our proposed GiPH for makespan minimization. We assume a heterogeneous computing environment where the computation time and communication time can be estimated from compute (task) and communication (data link) features of the device network. We first evaluate GiPH using synthetic data with randomly generated task graphs and device networks that cover a wide range of cases in §5.1, and specifically test on deep learning graphs in §5.2. In §5.3 we present a case study of applying GiPH to a realistic setting of autonomous inter-

section management for Connected Autonomous Vehicles (CAVs) using cooperative sensor fusion.

We compare GiPH with the following baseline algorithms:

- **Random placement sampling**: generating random placements of the task graph by sampling a feasible placement for each task from a uniform random distribution. This random baseline is representative of the average placement "quality" without GiPH's intelligent search.
- **HEFT** (Topcuoglu et al., 2002): the state-of-the-art heuristic scheduling algorithm for heterogeneous computing that we use as benchmark. It prioritizes tasks with high-level features and allocates each task, in order of priority, to a device that finishes the task the earliest (Earliest Finish Time (EFT) device selection).
- **Random task selection + EFT device selection**: a heuristic of placement search, where at each step a task in the graph is randomly selected and placed according to EFT. It is a direct adaption of HEFT as a search-based policy.
- **GiPH task selection + EFT device selection**: the version of GiPH without gpNet (ablation study). At each step, instead of deciding a task-device pair, the RL agent only selects a task. The task is then placed according to EFT.
- **Placeto** (Addanki et al., 2019): a search-based device placement algorithm that is the closest prior work to ours. It also does incremental placement, but does not consider re-placing the same task or device network features.
- **RNN-based placer**: another RL baseline based on the hierarchical model for device placement (HDP) (Mirhoseini et al., 2018). The Placer traverses the graph in topological order and directly decides the device assignment of each operator through an RNN-based policy network.

Evaluation metrics: We evaluate each algorithm's *placement quality* and *adaptivity*. We evaluate the placement quality through the completion time of a task graph, i.e., makespan. Since the makespan can vary significantly on different problem instances, we follow Topcuoglu et al. (2002) in normalizing the makespan to an instance-dependent lower bound, defining the Schedule Length Ratio:

$$SLR = \frac{makespan}{\sum_{v_i \in CP_{MIN}} \min_{d_j \in D_i} w_{i,j}},$$

where $w_{i,j}$ is the expected time of running task v_i on device d_j and CP_{MIN} is the critical path based on the minimum computation cost of each task node. The placement algorithm that gives the *lowest* SLR is the best with respect to the placement performance. We report the average SLR of different problem instances. We evaluate the adaptivity of the algorithms by measuring the average SLR achieved after each network change following a random device addition and deletion procedure.

Experiment details: Both GiPH and Placeto use a two-layer feed-forward neural network (FNN) with the same number of hidden units as the input dimension for node and/or edge feature pre-embedding before message passing. Messages are aggregated by mean. GNN implementations of GiPH and Placeto have comparable sizes, and are detailed in Appendix B.7 (e.g., features, neural network sizes, running time). Both are trained using the policy gradient method REINFORCE (Williams, 1992) with 200 episodes.

For the RNN-based placer network, we follow the HDP paper and use a sequence-to-sequence model with a bi-LSTM for the encoder and a unidirectional LSTM with an attention mechanism for the decoder. Since HDP does not aim to generalize to new device networks or new application graphs, we only compare the placement quality by training a new Placer policy on each test case with 4 Placer samples each time until the latency is no longer improved.

All policies are trained using Adam optimizer with a fixed learning rate 0.01. ReLU activation is used. The discounting factor $\gamma = 0.97$. The placement performance is evaluated using a runtime simulator (execution model and latency model detailed in Appendix B.5).

We separately generate training and test datasets, each composed of a set of task graphs and a set of device networks. For testing, all search-based policies start from the same initial placement for fair comparison. Since the action space grows linearly with the number of computational tasks, we set the episode length to be multiples of the number of tasks in the task graph, and empirically find that twice the size of the graph $2|V|$ step are enough for the policy to converge to a solution. Since Placeto fixes the number of search steps to $|V|$, we start a new search episode for Placeto after $|V|$ steps. Each policy outputs the SLR of the best placement found so far within the episode.

5.1 General Task Graphs and Networks

Dataset: We follow the parametric method used by Topcuoglu et al. (2002) to generate random task graphs and random device networks with various characteristics (e.g., number of devices, average delay of the device networks, etc.) depending on the input parameters. To evaluate our work on a full range of task graphs and device networks, we decouple their parameters so that each can be generated independently. The data generation process is detailed in Appendix B.2. Our generators also randomly specify placement constraints by adding a hardware-requirement property to each task and a corresponding hardware-support property to each device. One input parameter specifies the average number of feasible devices for each task.

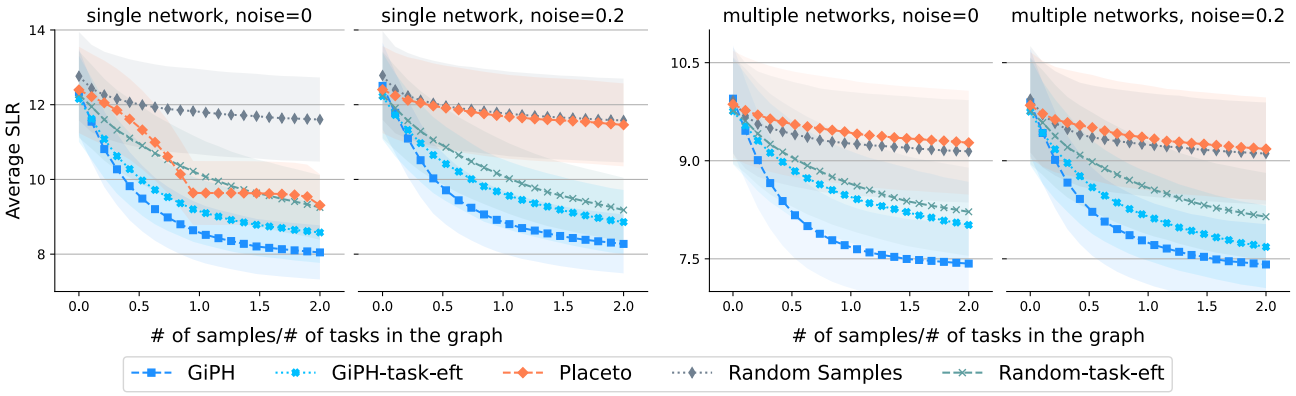


Figure 4. Placement quality and search efficiency of search-based policies. Noise=0.2 means the communication and computation time can vary up to 20% of the average value. GiPH consistently finds placements with the lowest SLR (i.e., best performance).

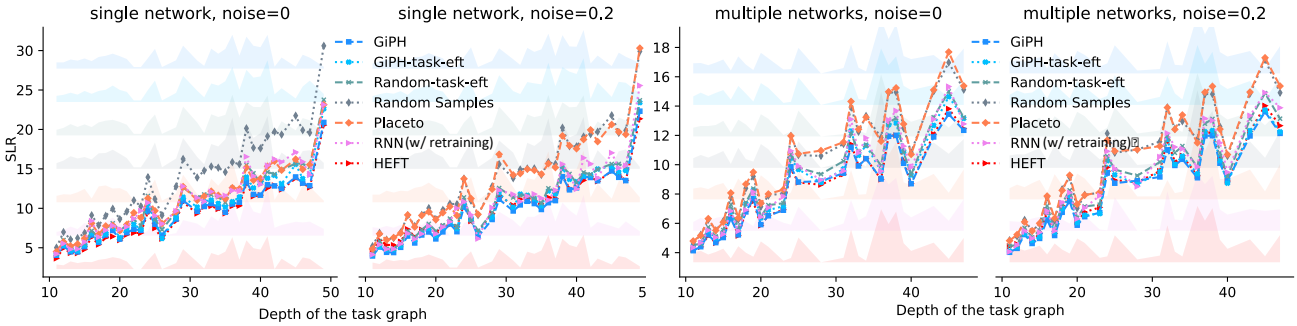


Figure 5. Average SLR with respect to the depth of the task graph. The background colorbars show the standard deviation of SLR for each method in the order they are listed in the legend.

Generalizability: We consider two cases: (1) The *single-device-network case*, where the search-based methods are trained and tested on a single device network. This case mainly considers application-level generalization, and is the problem setting used in prior work, including Placeto. A set of 300 randomly sampled task graphs is split equally for training and testing. (2) The *multiple-device-network case*, where multiple device networks, with varying compute and communication capacities per device, are used for training and testing. This case further includes device-network generalization. 500 test cases are sampled from combinations of 10 device networks and 120 graphs. All experiments are done with and without 20% noise added to the computation and communication times, which can model the random performance of real systems and estimation errors.

The average SLR across test cases as a function of the number of the search steps is shown in Fig. 4. In all cases, our GiPH policy outperforms other search policies and more rapidly finds better placements within fewer search steps. It achieves up to 30.4% lower completion time compared to the random baseline, which represents the average placement "quality". GiPH also exhibits resistance to variations in the communication and computation times (noise). GiPH-task-EFT, without using gpNet, is not as good as GiPH, but the RL on the task selection still allows it to outperform the random selection under the Random-task-EFT policy. In contrast, the performance of the Placeto policy significantly degrades under noise, probably because the agent cannot decouple the noise sources without a proper representation of the device network. When multiple networks are involved, Placeto even becomes worse than random because, without considering device-level features, the policy learns false local optima that no longer exist in a new device network.

Fig. 5 shows the SLR of the final placements found by different algorithms on the testset with respect to the depth of the task graph. As the task graph grows in depth, the SLRs for all methods increase because the critical paths are longer, increasing the makespan of the task graphs. GiPH outperforms other search-based methods in most of the cases and is comparable to the state-of-the-art HEFT.

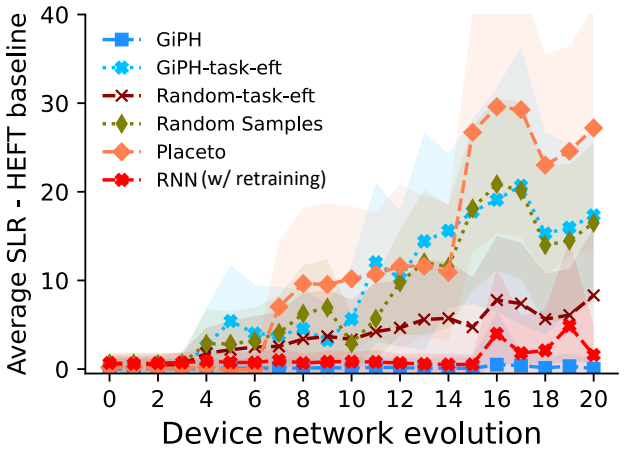
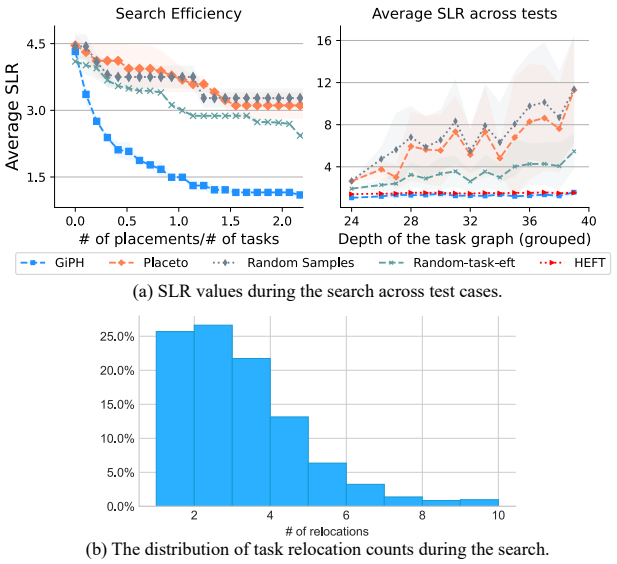


Figure 6. Adaptivity to device network changes. GiPH maintains stable performance, while other search methods achieve worse SLR as the network changes more.



(a) SLR values during the search across test cases.

(b) The distribution of task relocation counts during the search.

Figure 7. (a) Performance evaluation on deep learning computation graphs. (b) Counts distribution (only considering non-zero counts), with some tasks being relocated more frequently than others.

Adaptivity: We evaluate the performance of the learned policies on a changing device network. The network initially has 20 devices, and as the network evolves, some of the devices are randomly removed and later replaced with new devices of lower capacities (i.e., higher cost). The total number of devices is between 16 and 20. This may model devices running out of battery over time; to conserve energy, they may reduce their compute and communication resource expenditures. Fig. 6 shows the average SLRs of different algorithms compared to the HEFT baseline, calculated across 20 different application graphs running on the contrived device network. As the number of high-cost devices grows, the SLR for random samples increases due to higher average compute and communication times. Placeto again performs worse than random, being unable to adjust its policy as the device cluster changes. GiPH-task-eft fails to adapt because, without using gpNet to encode information about alternative placements, the policy cannot correctly decide which task to relocate next. Another baseline is the RNN-based placer, whose policy is retrained every time the network changes. With significant retraining, it maintains a low SLR value. However, GiPH learns a fully generalizable policy, and is the only search-based method that maintains stable performance (with almost the same SLR as HEFT) and adapts well to the changes in the device network without retraining.

Alternative implementations: Our GNN design has two key aspects: (1) It fully incorporates per-edge features. (2) It takes into account the partial ordering that defines the acyclic dependency of computation, by constructing message passing in both forward and backward directions with separate parameters. We further evaluate GiPH (1) without GNN, (2) without per-edge features, (3) with uni-directional GraphSAGE layers, (4) with k-step two-way message passing, and (5) without gpNet. Our proposed GNN design shows better convergence properties (Appendix B.6).

5.2 Deep Learning Graphs

We specifically look at deep learning (DL) applications and evaluate GiPH on DL computation graphs generated by ENAS (Pham et al., 2018), an automatic model design approach. A dataset of 300 graphs, using the recurrent cell designs from ENAS on the Penn Treebank benchmark for language models (details in Appendix B.3), is split equally for training and testing. Each graph contains 200-300 operators. Similar to Addanki et al. (2019); Mirhoseini et al. (2018), we partition the operators into predetermined groups and place operators from the same group on the same device. The grouping is done by iteratively merging the operator with in-degree one and lowest cost into its sole predecessor until the graph size is reduced to 40 nodes.

Training and testing are both done on one single simulated device network containing 8 devices. The evaluation results are shown in Fig. 7(a), where all search-based policies start from the same initial placement for fair comparison. GiPH again outperforms all the baselines by selectively relocating specific task groups based on their current placement, resulting in improved search efficiency compared to Placeto. Placeto traverses the graph and visits all nodes equally, whereas GiPH can adjust the placement of "critical" nodes more frequently within the same number of search steps. Figure 7(b) depicts the frequency distribution of relocation counts during the search using GiPH.

Table 1. Running time measurements of each task on device types A, B, and C with standard deviation values (in milliseconds)

	TYPE A	TYPE B	TYPE C
CAMERA	53±22	36±8	9±4
LIDAR	14±3	7±3	3±2
CAV DATA FUSION	35±9	35±4	11±9
RSU DATA FUSION	250±430	250±370	28±22

5.3 Case Study: Cooperative Sensor Fusion

We present an experimental case study of applying GiPH to a realistic setting of autonomous intersection management in CAVs. We consider an autonomous driving scenario where Roadside Units (RSUs) at major intersections gather real-time image data from connected infrastructure camera sensors (ISs) and camera and LIDAR data from CAVs around the intersection to plan the CAV trajectories in a timely manner. The data collection, communication, and processing form a time-sensitive dataflow task graph whose feasible placements need to be rapidly determined as CAVs move and their communication costs change.

Realistic application traces: We use a traffic simulation tool Simulation of Urban MObility (SUMO) (Lopez et al., 2018) to simulate traffic within a 6-block area in the center of Tempe AZ (Figure 8(a)). Each major intersection is equipped with one RSU and four CISs, and will "interact" with CAVs if they are within 400m of the RSU. We consider the sensor fusion pipelines in Andert & Shrivastava (2022) for sensing and localization, which involve camera object detection, LIDAR object detection, and fusion of the processed data for localization. The first two tasks need to run on GPUs, and the last one can run on any compute devices (CPU). We simulate the traffic with 3980 vehicles generated in an hour period (10% of CAVs) and collect application traces at 10 second intervals (Figure 8(b) as an example).

Placement experiment: We first establish a realistic latency model by measuring the actual running time of each task on Jetson Nano (Type A), Jetson TX2 (Type B), and Core i7 7700K with GTX1080 (Type C). The measurements are summarized in Table 1.

There are 36 RSUs located in the major intersections. We model them as Type-C devices. To introduce alternative placement options for running the sensor fusion pipelines, we randomly place 40 edge compute devices (10 type-A, 10 type-B, and 20 type-C devices) in the simulated area that can serve as additional RSUs for sensor fusion and trajectory planning. Finally, we model the communication network with a data rate that decays exponentially with distance. Details are included in the Appendix B.4.

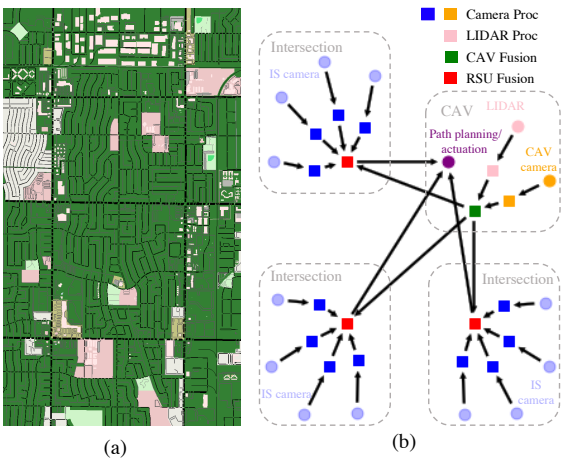


Figure 8. (a): SUMO traffic simulation area. (b): A snapshot of the application graph of autonomous traffic management for a CAV between three intersections.

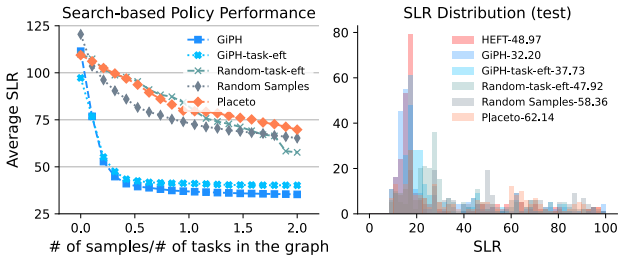


Figure 9. Case study using the application traces for autonomous intersection traffic management. (a) Search efficiency of search-based policies. (b) SLR distribution (under 100) with mean values included in the legend. Data distribution above 100 is very sparse and not shown in the figure for visualization.

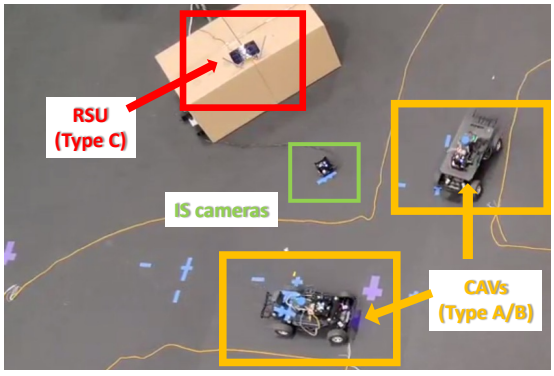


Figure 10. Real-world deployment of the sensor fusion pipeline (source: Andert & Shrivastava (2022))

Table 2. Measurements of the relocation overhead of each task in a small-scale deployment.

	DATA MIGRATION (BYTES)	STATIC INITIALIZATION DATA (KILOBYTES)	STARTUP TIME TYPE A (MS)	STARTUP TIME TYPE C (MS)
CAMERA	11494	72173.525	4273.73	794.66
LIDAR	560	24.576	60.98	9.26
CAV DATA FUSION	11796	38.110	0.39	0.11
RSU DATA FUSION	20907	38.950	2.83	1.00

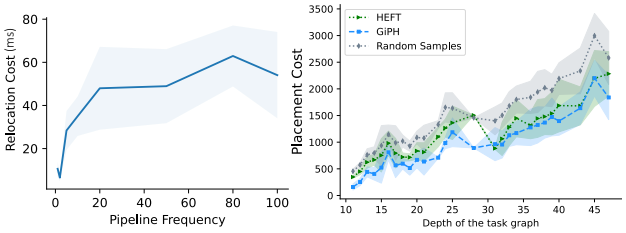


Figure 11. Left: incurred relocation cost with respect to pipeline frequency. Right: Using the placement found by GiPH, HEFT, and Random policy, the total energy cost across test cases.

We evaluate GiPH and other search-based policies on over 900 placement cases that are extracted from the application trace. They are evenly divided for training and testing. Figure 9(a) shows the average SLRs across 300 test cases during the placement search. GiPH again outperforms other policies and more rapidly finds better placements within fewer search steps. Figure 9(b) shows the SLR distribution of the final placement across test cases. GiPH is comparable to HEFT with a lower average SLR across test cases.

Real-world deployment and practical aspects: We measure the real-world relocation cost of each task of the cooperative sensor fusion in terms of the data migration, task initialization (static data), and startup times in a small-scale deployment (Fig. 10). The measured relocation overhead is summarized in Table 2.

To evaluate GiPH's practical handling of task relocation, we have integrated relocation costs into the simulator and measure the cost incurred when applying the learned policy as a certain network change occurs. The relocation cost is defined to be the sum of the data migration time and startup time. It is worth noting that most data processing applications run recurrently to collect and process new data, similar to our sensor fusion pipeline that processes video streams and LIDAR data at a fixed frequency. Thus, a single relocation may benefit future runs of the application.

To balance the trade-off between relocation costs and latency reduction, we divide the relocation cost by the frequency of pipeline runs. This allows us to assess the impact of relocation on pipeline performance over time by considering the potential value of task relocation in reducing the cost of multiple future runs of the application. Fig. 11(left) illustrates the amount of relocation cost that GiPH's policy would introduce at various pipeline frequencies, based on our relocation cost model. Our analysis shows that GiPH is more likely to relocate a task at a higher cost when the pipeline runs at a higher frequency.

6 CONCLUSION AND DISCUSSION

We present GiPH, an RL-based framework for learning generalizable placement policies that improves upon a given placement through incremental task relocation. We formulate the learning problem as a search problem such that the policy outputs incremental placement improvement steps. Using gpNet, a graph representation that captures relevant task-level and device-level features for placement, our framework accommodates task graphs and device networks of arbitrary size and characteristics. GiPH learns generalizable policies that find better placement results, faster than other search-based algorithms. We present a case study using realistic application traces for autonomous intersection management that utilizes the cost measurements in a real-world deployment. Although GiPH is only comparable to HEFT on makespan minimization, HEFT runs slowly in practice (Kiamari & Krishnamachari, 2021) and is specific to makespan optimization, while GiPH can optimize a wide range of objectives defined by the reward function (e.g., resource utilization, energy cost). Fig. 11(right) shows that GiPH outperforms random and HEFT in energy cost minimization by simply switching to a different reward function. We plan to further deploy GiPH on real-world device clusters with realistic dynamics that account for potential relocation overhead and dynmic application arrivals. Our preliminary results suggest that GiPH's results may vary depending on the stopping criterion for the placement search, and we will explore different criteria to ensure that GiPH learns a "good" application placement.

ACKNOWLEDGEMENTS

This work was supported by funding from NSF grants CPS-1645578, CPS-1646235, CNS-2106891, and USDOT UTC grant 69A3551747111. We also wish to thank the reviewers for providing valuable feedback on earlier paper drafts.

REFERENCES

Addanki, R., Venkatakrishnan, S. B., Gupta, S., Mao, H., and Alizadeh, M. Placeto: Learning generalizable device placement algorithms for distributed machine learning, 2019.

Andert, E. and Shrivastava, A. Accurate cooperative sensor fusion by parameterized covariance generation for sensing and localization pipelines in cavs. In *2022 IEEE 25th International Conference on Intelligent Transportation Systems (ITSC)*, pp. 3595–3602. IEEE, 2022.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V. F., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Çaglar Gülçehre, Song, H. F., Ballard, A. J., Gilmer, J., Dahl, G. E., Vaswani, A., Allen, K. R., Nash, C., Langston, V., Dyer, C., Heess, N. M. O., Wierstra, D., Kohli, P., Botvinick, M. M., Vinyals, O., Li, Y., and Pascanu, R. Relational inductive biases, deep learning, and graph networks. *ArXiv*, abs/1806.01261, 2018.

Dai, H., Khalil, E. B., Zhang, Y., Dilkina, B., and Song, L. Learning combinatorial optimization algorithms over graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pp. 6351–6361, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Dolati, M., Hassanpour, S. B., Ghaderi, M., and Khonsari, A. Deepvine: Virtual network embedding with deep reinforcement learning. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 879–885, 2019. doi: 10.1109/INFCOMW.2019.8845171.

Goudarzi, M., Palaniswami, M., and Buyya, R. A distributed deep reinforcement learning technique for application placement in edge and fog computing environments, 2021. URL https://arxiv.org/abs/2110.12415.

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf.

Hoque, M. A., Hasan, R., and Hasan, R. R-cav: On-demand edge computing platform for connected autonomous vehicles. In *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, pp. 65–70. IEEE, 2021.

Kiamari, M. and Krishnamachari, B. Gcnscheduler: Scheduling distributed computing applications using graph convolutional networks. *CoRR*, abs/2110.11552, 2021. URL https://arxiv.org/abs/2110.11552.

Lopez, P. A., Behrisch, M., Bieker-Walz, L., Erdmann, J., Flötteröd, Y.-P., Hilbrich, R., Lücken, L., Rummel, J., Wagner, P., and Wießner, E. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018. URL https://elib.dlr.de/124092/.

Luo, J., Li, X., Yuan, M., Yao, J., and Zeng, J. Learning to optimize dag scheduling in heterogeneous environment. *ArXiv*, abs/2103.06980, 2021.

Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., and Alizadeh, M. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pp. 270–288, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450359566. doi: 10.1145/3341302.3342080. URL https://doi.org/10.1145/3341302.3342080.

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., Kumar, N., Norouzi, M., Bengio, S., and Dean, J. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 2430–2439. JMLR.org, 2017.

Mirhoseini, A., Goldie, A., Pham, H., Steiner, B., Le, Q. V., and Dean, J. A hierarchical model for device placement. In *ICLR*, 2018.

Nayeri, Z. M., Ghafarian, T., and Javadi, B. Application placement in fog computing with ai approach: Taxonomy and a state of the art survey. *Journal of Network and Computer Applications*, 185:103078, 2021. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2021.103078. URL https://www.sciencedirect.com/science/article/pii/S1084804521000989.

Paliwal, A. S., Gimeno, F., Nair, V., Li, Y., Lubin, M., Kohli, P., and Vinyals, O. Regal: Transfer learning for fast optimization of computation graphs. *ArXiv*, abs/1905.02494, 2019.

Pallewatta, S., Kostakos, V., and Buyya, R. Qos-aware placement of microservices-based iot applications in fog computing environments. *Future Generation Computer Systems*, 131:121–136, 2022. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2022.01.012. URL https://www.sciencedirect.com/science/article/pii/S0167739X22000206.

Pham, H., Guan, M., Zoph, B., Le, Q., and Dean, J. Efficient neural architecture search via parameters sharing. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4095–4104. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/pham18a.html.

Salaht, F. A., Desprez, F., and Lebre, A. An overview of service placement problem in fog and edge computing. *ACM Computing Surveys (CSUR)*, 53(3):1–35, 2020.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: an Introduction*. MIT Press, 1998.

Topcuoglu, H., Hariri, S., and Wu, M.-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002. doi: 10.1109/71.993206.

Tu, Y., Ruan, Y., Wagle, S., Brinton, C. G., and Joe-Wong, C. Network-aware optimization of distributed learning for fog computing. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pp. 2509–2518. IEEE, 2020.

Ullman, J. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10 (3):384–393, 1975. ISSN 0022-0000. doi: https://doi.org/10.1016/S0022-0000(75)80008-0. URL https://www.sciencedirect.com/science/article/pii/S0022000075800080.

Wang, L., Jiao, L., He, T., Li, J., and Mühlhäuser, M. Service entity placement for social virtual reality applications in edge computing. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pp. 468–476. IEEE, 2018.

Weaver, L. and Tao, N. The optimal reward baseline for gradient-based reinforcement learning. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, UAI'01, pp. 538–545, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1558608001.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, may 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

Yan, Z., Ge, J., Wu, Y., Li, L., and Li, T. Automatic virtual network embedding: A deep reinforcement learning approach with graph convolutional networks. *IEEE Journal on Selected Areas in Communications*, 38(6):1040–1057, 2020. doi: 10.1109/JSAC.2020.2986662.

Zhou, Y., Roy, S., Abdolrashidi, A., Wong, D., Ma, P. C., Xu, Q., Zhong, M., Liu, H., Goldie, A., Mirhoseini, A., and Laudon, J. Gdp: Generalized device placement for dataflow graphs, 2019. URL https://arxiv.org/abs/1910.01578.

A ARTIFACT APPENDIX

A.1 Abstract

The artifact contains the implementation of the learning-based placement algorithm GiPH in the paper GiPH: Generalizable Placement Learning for Adaptive Heterogeneous Computing. This artifact appendix contains information on how to use the reference implementations to reproduce the main experiment results presented in the paper. We show (1) how to create the synthetic program graph and device network datasets, (2) how to train GiPH and other baselines, and (3) how to evaluate the learned model on a test set.

A.2 Artifact check-list (meta-information)

- **Data set:** Synthetic data, workload trace
- **Output:** Model parameters, latency trace, intermediate sample of placement configuration
- **Publicly available:** Yes, at https://github.com/uidmice/placement-rl
- **Code licenses:** MIT
- **Archived:** Zenodo. DOI: 10.5281/zenodo.7879679

A.3 Description

A.3.1 How delivered

The artifact is publicly achieved using Zenodo with DOI 10.5281/zenodo.7879679 and available at https://github.com/uidmice/placement-rl.

A.3.2 Hardware dependencies

No hardware dependency required. All experiments in the paper are done on CPUs only, but the code allows the use of CUDA. If running on an arm mac you need to run the x86 version of conda through rosetta as DGL requires x86 architecture.

A.3.3 Software dependencies

This project is best run in a conda environment to satisfy version requirements, these are contained in `requirements.txt`. The prototype is done using DGL 0.9.1 with a Pytorch 1.13.0 backend.

A.3.4 Data sets

The method used to generate the dataset of program graphs and devices networks are detailed in Appendix B.2. The implementation with detailed explanations are included in `Generate_data.ipynb` available in the repository. The notebook also includes visualization of the distribution of average communication and computation cost given the parameters.

Upon starting the experiment, a set of parameters will be fetched from the path specified by the `--data_parameters` command and used to generate the dataset for training and evaluation. The exact parameters for generating the dataset used in the paper are in `parameters/`. By default, `parameters/single_network.txt` is used.

We also allow passing customized graphs and networks for training and testing. The specific datasets used for the adaptivity experiment are available to download: https://drive.google.com/drive/folders/12nztX3XfJh3uFsTGTY2DdluQGsson2Am?usp=share_link.

A.4 Installation

Run the following commands to create a virtual environment and install relevant packages:

```
$ conda create -n placement matplotlib
    =3.5.1 networkx=2.5 numpy=1.19 python
    =3.8.10 pytorch=1.13.0 requests tqdm -
    c pytorch
$ conda activate placement
$ pip3 install simpy
$ conda install -c dglteam dgl=0.9.1
```

A.5 Experiment workflow

The pipeline can be run to create a trained model and testing results by running `main.py`.

To train and evaluate the model on default parameters:

```
$ python main.py --train
```

To train and evaluate the model with customized dataset:

```
$ python main.py --train
   --load_train_graphs data/single-network/
       train_program.pkl
   --load_test_graphs data/single-network/
       eval_program.pkl
   --load_train_networks data/single-
       network/train_network.pkl
   --load_test_networks data/single-network
       /eval_network.pkl
```

Each run creates a subfolder inside the log directory, `--logdir`, with a date/time stamp. The following are saved:

1. The model parameters of both the GNN (`embedding_*.pk`) and the policy network (`policy_*.pk`).
2. Episodic training data (`train_data.pkl`), including the latency traces, intermediate sample placements, etc.
3. Episodic evaluation results (`eval_data.pkl`), if enabled, performed every after a few model updates are made.
4. The argument inputs (`args.pkl`).

We can also use `main.py` to run Placeto, GiPH-task-eft, and alternative GiPH implementations described in Appendix B.6, by providing additional command line arguments.

A.6 Evaluation and expected result

After finish training, we can use `main.py` to load a learned model from a run directory and run a specified amount of tests on it. For evaluation, `--run_folder` must be provided.

```
$ python main.py --test
   --run_folder (logdir)/yyyy-mm-dd_hh-mm-
       ss_(suffix)
   --num_testing_cases 200
```

By default, the last saved model will be loaded. To load the model parameters from an earlier save, we can do the following:

```
$ python main.py --test
   --run_folder (logdir)/yyyy-mm-dd_hh-mm-
       ss_(suffix)
   --policy_model policy_20.pk
   --embedding_model embedding_20.pk
```

Each test will create a subfolder inside the run directory with a name starting with `test_` and date/time stamp. The evaluation on individual test cases (a pair of a program and a device network) will be saved individually.

A.7 Experiment customization

`main.py` supports command line arguments to modify the operation and parameters of the experiment pipeline, including the learning rate, the random seed, how often we evaluate the model during training, how many times we repeat the same test cases, etc. See `README.md`.

B APPENDIX

B.1 gpNet Algorithm

Algorithm gpNet

Require: $G = (V, E, \boldsymbol{\beta}^n, \boldsymbol{\beta}^e), N = (D, \boldsymbol{b}^n, \boldsymbol{b}^e), \mathcal{M}^{G\to N}$

Initialize an empty graph $H = (V_H, E_H, \boldsymbol{x}^n, \boldsymbol{x}^e)$
$V_{H,\mathcal{P}} = \{\}$ {The set of pivots}
for $v_i \in V$ **do**
  $O_i = \{\}$ {The placement options of v_i}
  for $d_j \in D_i$ **do**
    add node $u = (v_i, d_j)$ to V_H and O_i
    node feature of u: $x_u^n = f_n(\beta_i^n, b_i^n)$
    if $\mathcal{M}^{G\to N}(v_i) = d_j$ **then**
      add $u = (v_i, d_j)$ to $V_{H,\mathcal{P}}$
    end if
  end for
end for
for $(v_i, v_j) \in E$ **do**
  for $u_1 = (v_i, d_k) \in O_i, u_2 = (v_j, d_l) \in O_j$ **do**
    if $u_1 \in V_{H,\mathcal{P}}$ or $u_2 \in V_{H,\mathcal{P}}$ **then**
      add edge $c = (u_1, u_2)$ to E_H
      edge feature of c: $x_c^e = f_e(\beta_{ij}^e, b_{kl}^e)$
    end if
  end for
end for
return H {gpNet representation of the placement}

B.2 Synthetic Data Generation

Following the parametric method used by Topcuoglu et al. (Topcuoglu et al., 2002), we implement a random graph generator and a random device network generator to generate task graphs and device networks with various characteristics depending on input parameters.

Task graph generator: The task graph generator builds single-entry and single-exit task graphs, i.e., exactly one task (entry) is without any parent and exactly one task (exit) is without any child in the graph. For application graphs with more than one entry (exit) task, those tasks can be connected to a pseudo entry (exit) task with zero-cost edges. The following input parameters are used to build task DAGs:

- Number of tasks in the graph M.
- Shape parameter α. The depth of the DAG (i.e., the length of the longest path) is randomly generated from a uniform distribution with a mean value $\sqrt{M}/\alpha$. The width for each level is randomly generated from a uniform distribution with a mean value $\alpha\sqrt{M}$.
- Connection probability p_c. The probability of having a directed edge (data link) from a node at a higher level to a node at a lower level.
- Average compute requirement of tasks $\overline{C}$.
- Average amount of data transmission along data links $\overline{B}$.
- Heterogeneity factor for compute requirements among tasks ϵ_C. The compute requirement C_i of a task i is uniformly sampled from the range $\left[\overline{C} \times (1 - \epsilon_C), \overline{C} \times (1 + \epsilon_C)\right]$.
- Heterogeneity factor for the amount of data transmission ϵ_B. The amount of data to be transferred along the data link (v_i, v_j), B_{ij}, is uniformly sampled from the range $\left[\overline{B} \times (1 - \epsilon_B), \overline{B} \times (1 + \epsilon_B)\right]$.

Two example graphs with different shape parameters and heterogeneity factors are shown in Fig. 12. Note that a larger shape parameter ($\alpha = 1$ instead of $\alpha = 0.5$) leads to a visibly wider and shorter graph. Similarly, larger ϵ_B and ϵ_C values exhibit more variability in the amount of data transmission and task compute respectively.

Device network generator: The following input parameters are used to generate device networks:

- Number of devices m.
- Average compute speed of devices $\overline{SP}$.
- Average bandwidth between devices $\overline{BW}$.
- Average communication delay between devices $\overline{DL}$. The communication delay DL_{kl} between device k and l, with $k \neq l$, is uniformly sampled from $[0, 2 \times \overline{DL}]$. The delay to the device itself is zero, i.e., $DL_{kk} = 0$ for all d_k.
- Heterogeneity factor for compute speed among devices ϵ_{SP}. The compute speed SP_k of a device k is uniformly sampled from the range $\left[\overline{SP} \times (1 - \epsilon_{SP}), \overline{SP} \times (1 + \epsilon_{SP})\right]$.

Table 3. Notation used throughout the paper

Symbol	Description
G	Task graph
V	The set of nodes (task)
E	The set of edges (data links/dependencies)
v_i	Task i
β_i^n, β_{ij}^e	Node feature of task i, edge feature of data link (v_i, v_j), for the task graph
N	Device network
D	The set of devices
D_i	The set of feasible devices for task i
d_k	Device k
b_k^n, b_{kl}^e	Compute feature of device k, communication feature between device k and l
$\mathcal{M}^{G\to N}, \mathcal{M}$	A mapping from task V (of G) to device D (of N)
$\mathcal{P}$	A general placement described by a tuple $(G, N, \mathcal{M}^{G\to N})$
ρ	The objective function as a function of the placement
$S_{G,N}, A_{G,N}$	The state space and action space of the MDP for a given problem instance
s_t, a_t, r_t	The state, action, and reward at step t
$H = (V_H, E_H)$	gpNet graph representation, with nodes V_H and edges E_H
O_i	The set of placement options for a task i
$V_{H,\mathcal{P}}$	The set of pivots of H
x_u^n, x_{uv}^e	The composed node feature of u and edge feature of (u, v) in gpNet H
e_u	The embedding of node (action) u
q_u	Node (action) score of u

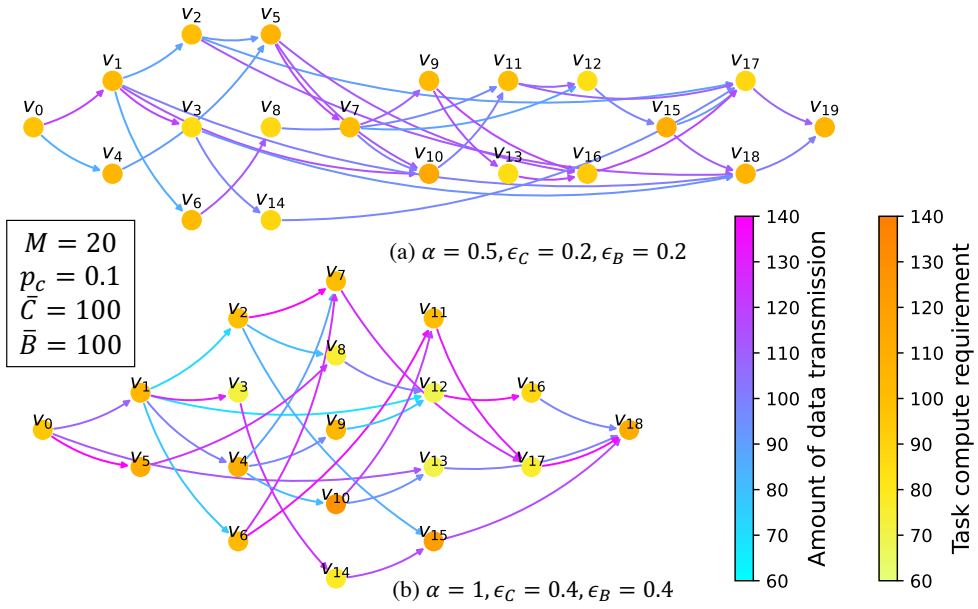


Figure 12. Example task graphs

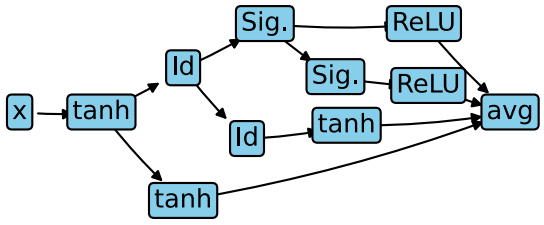


Figure 13. Sample RNN cell design from ENAS

- Heterogeneity factor for the communication bandwidth ϵ_{BW}. The communication bandwidth BW_{kl} between device k and l, with $k \neq l$, is uniformly sampled from $\left[\overline{BW} \times (1 - \epsilon_{BW}), \overline{BW} \times (1 + \epsilon_{BW})\right]$. The bandwidth for sending data to the device itself is infinite, i.e., $BW_{kk} = \infty$ for all d_k.

The placement constraints are specified by adding a hardware-requirement property to each task and a corresponding hardware-support property to each device with non-zero probability. Given a task graph and a device network, each task can only run on devices with the hardware support the task requires. A higher probability for a hardware support results in a larger number of feasible devices for a task requiring that hardware.

Our simulator also allows for assigning multiple values to each parameter used by the generators. A specific combination of parameter values is used to generate data. The source code of the generators are included in the supplementary material together with the parameters used for the experiments in this paper.

B.3 Deep Learning Graphs Generation

To evaluate GiPH on deep learning graphs, we use ENAS (Pham et al., 2018) to generate different neural network architectures. Specifically, we sample 10 recurrent cell designs from ENAS on the Penn Treebank benchmark for language models. One sample graph is shown in Fig. 13. For each cell design, we vary the number of unrolled steps and workload in terms of the batch size to generate 30 different deep learning graphs. The dataset contains 300 graphs in total. The number of unrolled steps is uniformly sampled from 20 to 30, and the batch size from 80 to 150.

B.4 Case Study Details

Task/device features: We define an average compute requirement $\overline{C}$ for each of the four tasks and a pair of compute features $\overline{T}$ and $\overline{S}$ for the CPU and GPU of each type of devices. $\overline{T}$ is a measure of the time used for running a unit of compute and $\overline{S}$ is a measure of the startup time of that type of device, both in milliseconds. The values are determined by fitting a computation latency model that satisfies $\overline{C_i}\overline{T_j} + \overline{S_j} = \mu_{i,j}$, where $\mu_{i,j}$ is the measured average time of running task i on device j according to Table 1.

Communication/link features: We estimate the sizes of the data transfer $\overline{B_{ij}}$ in bytes between tasks i and j based on the hardware and experiment setup as described by Andert & Shrivastava (2022). We assume a communication bandwidth between devices that decays exponentially with distance $\overline{BW} = 60 \exp(-d/100)$Mbps, where d is the distance in meters between two devices.

B.5 Simulator

We built a Python-based runtime simulator to model a distributed computing environment with the following characteristics: (1) Each device executes runnable tasks in a first-in-first-out (FIFO) manner. (2) Task execution is non-preemptive, i.e., not interruptible by other tasks once the execution starts. (3) At most one task can run on a device at a time. (4) Computation can be overlapped with communication. The above aspects match with the real execution models in some existing distributed computing environments (e.g., Tensorflow, as demonstrated in (Addanki et al., 2019)).

Task model: A task is a unit of compute that can run on devices with the hardware support the task requires. A non-entry task v is *runnable* on a device d when all of its parents have finished execution and all of the data inputs from parents that the task takes to run are available on d (i.e., the inputs to v have been either locally produced on d or transferred from other devices to d). An entry task is always runnable. The data outputs of a task v become available for use on d or ready to be transferred from d once task v's execution finishes.

Device model: Each device can execute at most one task at a time, and the task execution is assumed to be *nonpreemptive*. Each device keeps runnable tasks in a *FIFO* quene and executes them in the order they become runnable. Inter-device communication is assumed to perform without contention, and therefore, sending output data to multiple devices at the end of task execution is modeled to happen concurrently without queueing. We also assume that computation can be overlapped with communication.

Latency model (synthetic data): We model the execution (computation) time $w_{i,k}$ of running a task v_i on a device d_k to be proportional to the compute requirement of the task and inversely proportional to the compute speed of the device:

$$w_{i,k} = \frac{C_i}{SP_k}. \tag{2}$$

The data transmission (communication) time $c_{ij,kl}$ of a data link (v_i, v_j), with v_i mapped to device d_k and v_j mapped to device d_l, is calculated as:

$$c_{ij,kl} = DL_{kl} + \frac{B_{ij}}{BW_{kl}}. \tag{3}$$

Note that the communication time is zero if the two tasks are placed on the same device because we set zero delay and infinite bandwidth for local data transmission when generating the device network (Appendix B.2).

With non-zero noise $\sigma \in (0, 1)$, Equations 2 and 3 give the expected computation and communication times, but their realizations are sampled from a uniform distribution with range $[w_{i,k}(1 - \sigma), w_{i,k}(1 + \sigma)]$ for computation time and range $[c_{ij,kl}(1 - \sigma), c_{ij,kl}(1 + \sigma)]$ for communication time. In a more realistic setting, the latency model can also be replaced by real

measurements of communication and computation times from profiling tools.

A discrete event simulation tool for Python, SimPy, is used to handle the interactions between the start and end of data transmission/task execution across devices. Each device d_k keeps a FIFO queue Q_k for all runnable tasks on it. The queue is dequeued when d_k is not busy or have finished executing the previous task. We define the following events with corresponding event handlers:

- *Task start* t_i^s: Used to mark the start time of the execution of a task v_i. Given $\mathcal{M}(v_i) = d_k$, its timestamp t_i^s is the time when the task is dequeued from Q_k. Device d_k becomes busy at t_i^s and stays busy until $t_i^s + w_{i,k}$.
- *Task done* t_i^d: Used to mark the end of the execution of a task v_i. Given $\mathcal{M}(v_i) = d_k$, its timestamp $t_i^d = t_i^s + w_{i,k}$. The device becomes free (not occupied) when the task is done. This event also triggers the data transmission to other devices where the child tasks reside, i.e., data communication from d_k to $\mathcal{M}(v_j)$ for $(v_i, v_j) \in E$.
- *Transmission start* t_{ij}^s: Used to mark the start of the transmission of the data link (v_i, v_j). Its timestamp is the same as the *task done* event of v_i, i.e., $t_{ij}^s = t_i^d$ for all $(v_i, v_j) \in E$.
- *Transmission done* t_{ij}^d: Used to mark the end of the transmission of the data link (v_i, v_j). Given $\mathcal{M}(v_i) = d_k$ and $\mathcal{M}(v_j) = d_l$, the timestamp $t_{ij}^d = t_{ij}^s + c_{ij,kl}$.
- *Task runnable* t_i^r: Used to signal a task v_i becomes runnable after receiving all inputs, i.e., $t_i^r = \max_{(v_j,v_i)\in E} t_{ji}^d$. When a task v_i becomes runnable, it is inserted to the FIFO queue of the device d_k where the task is placed (i.e., $\mathcal{M}(v_i) = d_k$).

After the entry task starts and the data flows through the whole graph, the completion time (makespan) of the task graph is the time duration from the start of the entry task to the end of the exit task, i.e., $t_{exit}^d - t_{entry}^s$.

B.6 Implementation Alternatives

We compare with the following GNN alternatives:

1. **GiPH-NE**: GiPH's two-way message passing without edge features. To compensate for the loss of edge information, the mean feature value of out edges of a node is appended to its node feature.
2. **GraphSAGE-NE**: With the same node features as used by GiPH-NE, GraphSAGE-NE replaces the two-way message passing with a 3-layer uni-directional GraphSAGE network (Hamilton et al., 2017).
3. **GiPH-NE-Pol**: GiPH without GNN. GiPH-NE-Pol directly feeds the same raw node features as used by GiPH-NE to the policy network.

While combining per-edge information with node features, these implementations still use the gpNet as inputs. The effect of using gpNet can be seen by comparing to GiPH-task-eft, which does not use gpNet to encode the placement information.

4. **GiPH-task-eft**: GiPH task selection with EFT device selection. Without using gpNet, selecting a task and deciding where to place it are done separately.

The number of message passing steps in GiPH is equal to the depth of the graph, which can be time-consuming for large graphs. We consider the following GNN alternative that limits the number of steps to propagate local structural information.

5. **GiPH**-k: GiPH k-step two-way message passing defined by:

$$e_u^{t+1} = h_2^c\left(\sum_{v\in\xi(u)} h_1^c\left([e_v^t \parallel x_{vu}^e]\right)\right) + h_3^c(x_u^n), \quad (4)$$

for $t = 0, ..., k$ with $e_u^0 = x_u^n$. $h_1^c(\cdot)$, $h_2^c(\cdot)$ and $h_3^c(\cdot)$ are feed-forward neural networks with trainable parameters shared for message passing steps in each direction.

Neural network implementation: For the k-step message passing GiPH-k, a two-layer FNN (h_3^c) with the same number of hidden units as the input dimension is used to pre-embed node features to a higher-dimensional space. We set the output dimension to be 10. Similar to GiPH, the message passing and message aggregation use single-layer FNNs. Details are included in Table 4 and Table 5.

Policy convergence: We train the policies using different implementations with the same training dataset for 200 training episodes and test the policy convergence by evaluating the learned policies every 5 training episodes. The evaluation is done by applying the learned policies to the same set of 20 evaluation cases, whose task graphs and device networks are not in the training dataset.

The experiment results are shown in Fig. 14. For the result on the right hand side, we use the same device network throughout the training and testing. For the middle plot, we use fixed-sized device networks, and for plot on the right hand side, we further vary the size of the device networks. For the first two experiments, we also include Placeto for comparison. We find that the policies tend to vary less when trained on device networks of various sizes, which suggests the benefit of having diverse training data.

The policies of GiPH, GiPH-3, GiPH-5 and GiPH-NE-Pol converge in both cases. GiPH-task-eft fails to converge in both cases, probably because it does not have a unified placement update policy without using gpNet. The separation of the device selection with the RL policy for task selection makes the policy learning harder. GraphSAGE-NE and GiPH-NE both incorporate edge features into the node features; while GiPH-NE constructs message passing in both forward and backward directions, the message passing of GraphSAGE-NE is uni-directional, which may be the cause of divergence of GraphSAGE-NE in both cases.

We attribute the success of GiPH-NE-Pol to our feature selection described in Appendix B.7. Specifically, the start-time potential as a node feature itself provides aggregated information of the neighbors, which may greatly help the policies to converge. We further repeat the multisized-device-network experiment without using the start-time potential as a node feature. The convergence of GiPH, GiPH-3, GiPH-5 and GiPH-NE-Pol is shown in Fig. 15. In this case, without using GNN, GiPH-NE-Pol performs poorly and the policy does not improve the average SLR at all. In contrast, GiPH still successfully creates a sharp drop of SLR at the beginning of the training. The training efficiency of all four policies decreases after removing the start-time potential node feature, with GiPH being the least-influenced.

Table 4. Dimension Summary

	GiPH	GiPH-k	GiPH-NE	GiPH-NE-Pol	GraphSAGE-NE	Placeto	RNN-Placer
Node feature dim_n	4	4	8	8	8	5	-
Edge feature dim_e	4	4	-	-	-	-	-
Embed-ding dim_o	5	5	5	-	10	5	$n_{type}+1$ $+\max(d_{out})$ $+n_{nodes}$

Table 5. Neural network implementation details.

	Node transform layer	Message function	Aggregation function	Message passing k	Node summery dim	Policy
GiPH	$\mathbf{dim_n}$ dim_n $\mathbf{dim_o}$	$\mathbf{dim_o}+\mathbf{dim_e}$ $\mathbf{dim_o}+\mathbf{dim_e}$	$\mathbf{dim_o}+\mathbf{dim_e}$ $\mathbf{dim_o}$	Graph depth	dim_o*2 $=10$	$\mathbf{10}$ 16 $\mathbf{1}$
GiPH-k	$\mathbf{dim_n}$ dim_n $\mathbf{dim_o}$	$\mathbf{dim_o}+\mathbf{dim_e}$ $\mathbf{dim_o}+\mathbf{dim_e}$	$\mathbf{dim_o}+\mathbf{dim_e}$ $\mathbf{dim_o}$	k	dim_o*2 $=10$	$\mathbf{10}$ 16 $\mathbf{1}$
GiPH-NE	-	$\mathbf{dim_n}$ $\mathbf{dim_n}$	$\mathbf{dim_n}$ $\mathbf{dim_o}$	Graph depth	dim_o*2 $=10$	$\mathbf{10}$ 16 $\mathbf{1}$
GiPH-NE-Pol	-	-	-	-	-	$\mathbf{8}$ 16 $\mathbf{1}$
Graph-SAGE -NE	$\mathbf{dim_n}$ $\mathbf{16}$	$\mathbf{16}$ $\mathbf{16}$	$\mathbf{16}$ $\mathbf{dim_o}$	3	dim_o $=10$	$\mathbf{10}$ 16 $\mathbf{1}$
Placeto	$\mathbf{dim_n}$ dim_n $\mathbf{dim_n}$	$\mathbf{dim_n}$ dim_n $\mathbf{dim_n}$	$\mathbf{dim_n}$ dim_n $\mathbf{dim_n}$	8	dim_o*2*4 $=40$	$\mathbf{40}$ 32 $\mathbf{n_{dev}}$

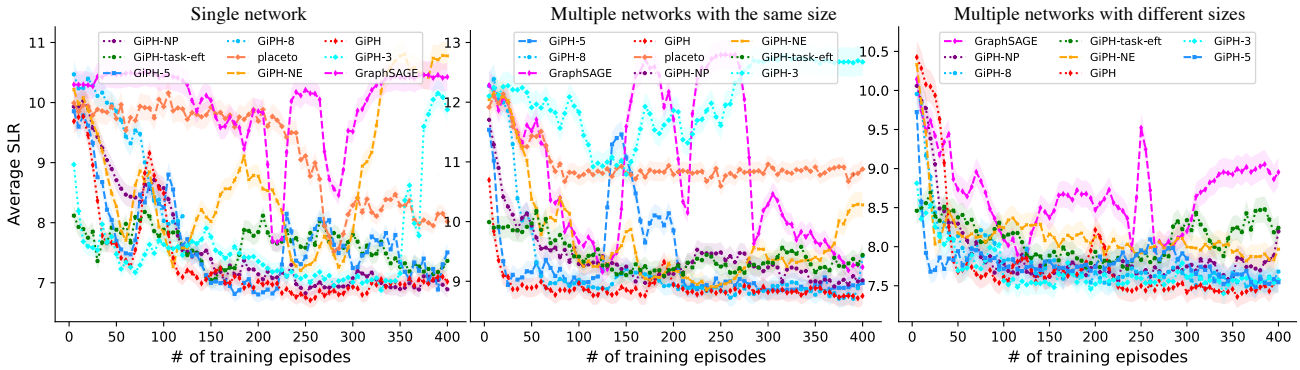


Figure 14. Average SLR across 20 evaluation cases with respect to the number of training episodes. Left: Training and evaluating on a single network. Middle: Training and evaluating on fixed-sized device networks. Right: Training and evaluating on device networks of various sizes.

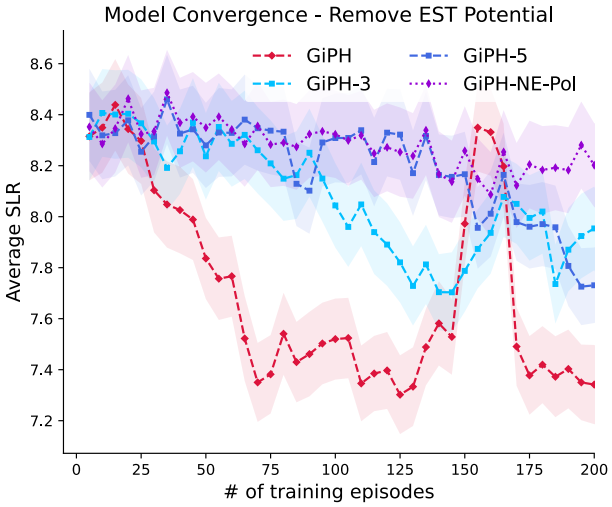


Figure 15. Convergence result after removing the start-time potential as a node feature.

Placement quality: We randomly select 1000 test cases from the test dataset to test learned policies. We count the number of occurrences of better placements among GiPH, its variants and HEFT. The result is summarized in Table 6. In general, GiPH is better than its variants and produces comparable results as HEFT.

B.7 Experiment Details

Features: The node feature vector of (v_i, d_k) in gpNet consists of: (1) the compute requirement of the task C_i, (2) the compute speed of the device SP_k, (3) the expected compute time $w_{i,k}$, and (4) the start-time potential of task v_i on d_k, which is defined as the time difference between the earliest possible start time of v_i on d_k and the actual start time of v_i.

The edge feature vector of $((v_i, d_k), (v_j, d_l))$ in gpNet consists of: (1) the amount of data transmission from v_i to v_j, B_{ij}, (2) the communication bandwidth from device d_k to d_l, BW_{kl}, (3) the communication start-up delay DL_{kl}, and (4) the expected communication time $c_{ij,kl}$.

For Placeto, the node feature vector of each operator is created by concatenating (1) the average compute time, (2) the average output data bytes, (3) the current placement, (4) an indicator of whether the operator is the current one to be placed, and (5) an indicator of whether the operator has been placed in the episode.

For the RNN-based placer, the input embedding of each operator is created by concatenating four vectors: (1) a one-hot encoding of the type of hardware requirement for placement constraints, (2) a scalar of its compute requirement, (3) a vector containing the number of data bytes of all its outgoing edges, of size equal to the maximum out-degree of the graph, and (4) a vector for the adjacency of the operator, of size equal to the number of operators in the graph. The dimensions of the node feature, edge feature, and embedding are summarized in Table 4.

RL training: The policy gradient method REINFORCE is used for training the RL policy (Williams, 1992). During each episode, a placement problem (G, N) is sampled from a training set $\mathcal{G}_T \times \mathcal{N}_T$. Starting from a random placement s_0, the agent collects observations (s_t, a_t, r_t) at each step $t = 0, ..., T$ following the current policy π_θ. It updates its policy parameters at the end of each episode

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} \gamma^t \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} - b_t \right),$$

where α is the learning rate, γ is the discounting factor, and b_t is a baseline for reducing the variance of the policy gradient (Weaver & Tao, 2001). b_t can be any function as long as it does not depend on the action at time t. We set it to be the average reward before step t in an episode.

Running time: We report the average training and running times of each policy. All experiments are done on CPUs only. Both the training time and the running time include the time used to generate the input graphs (with features) and run the policy (for placement updates), and training has additional gradient update steps at the end of each episode. The values reported in Table 7 are averaged over placement samples taken during the training and testing (given the same training and testing datasets). Fig.

Table 6. Pair-wise placement quality comparison, showing the percentage of test cases for which the method in the row index has SLR that is better than/equal to/worse than the SLR for the method in the column index. For example, GiPH's SLR is better than GiPH-3's for 53.0% of the test cases and equal to GiPH-3's for 7.0% of the test cases.

		GiPH	GiPH-3	GiPH-5	GiPH-NE	GiPH-NE-Pol	GiPH-task-eft	HEFT
GiPH	Better		**53.0%**	**55.2%**	**74.8%**	**60.6%**	**82.2%**	**59.0%**
	Equal		7.0%	6.4%	2.4%	6.0%	3.0%	5.2%
	Worse		40.0%	38.4%	22.8%	33.4%	14.8%	35.8%
GiPH-3	Better	40.0%		**51.6%**	**68.8%**	**55.8%**	**74.6%**	**51.4%**
	Equal	7.0%		5.0%	1.8%	5.3%	3.0%	4.4%
	Worse	**53.0%**		43.4%	29.4%	38.9%	22.4%	44.2%
GiPH-5	Better	38.4%	43.4%		**66.0%**	**52.0%**	**74.8%**	**51.2%**
	Equal	6.4%	5.0%		1.8%	3.6%	2.2%	3.0%
	Worse	**55.2%**	**51.6%**		32.2%	44.4%	23.0%	45.8%
GiPH-NE	Better	38.4%	29.4%	32.2%		32.6%	**61.1%**	32.6%
	Equal	2.4%	1.8%	1.8%		1.4%	0.7%	2.2%
	Worse	**74.8%**	**68.8%**	**66.0%**		**66.0%**	38.2%	**65.2%**
GiPH-NE-Pol	Better	33.4%	38.9%	44.4%	**66.0%**		**70.4%**	44.6%
	Equal	6.0%	5.3%	3.6%	1.4%		2.6%	2.8%
	Worse	**60.6%**	**55.8%**	**52.0%**	32.6%		27.0%	**52.6%**
GiPH-task-eft	Better	14.8%	22.4%	23.0%	38.2%	27.0%		29.2%
	Equal	3.0%	3.0%	2.2%	0.7%	2.6%		6.4%
	Worse	**82.2%**	**74.6%**	**74.8%**	**61.1%**	**70.4%**		**64.4%**

17 shows how the policy running and training times vary with the size of the application graphs. Since in GiPH the message passing runs sequentially from entry node to the exit node, the policy running time of GiPH grows with the size of the input graph. Limiting the number of message passing steps to k (GiPH-3, GiPH-5) significantly reduces the overhead of running the policy.

B.8 Supplementary Results

Total cost minimization: To demonstrate that GiPH provides a general learning framework for optimizing a variety of objectives (e.g., resource utilization and balancing, energy cost, etc.), we also test GiPH performance for cost minimization, where the cost is defined as the sum of communication cost of each data link and computation cost of each task, i.e., $Cost = \sum_{i \in V} w_{i,\mathcal{M}(i)} + \sum_{ij \in E} c_{ij,\mathcal{M}(i)\mathcal{M}(j)}$ minimized over feasible mapping $\mathcal{M}$.

We use the same training and testing datasets as in the multiple-device-network case, and simply replace the reward with the cost reduction at each step. Fig. 16 (left) shows the search efficiency for cost minimization compared with a random sampling baseline and Fig. 16 (right) reports the total cost of the final placements across testing cases found by GiPH, random sampling, and HEFT.

Table 7. Policy Running Time per Placement Sample

	GiPH	GiPH-3	GiPH-5	GiPH-NE	GiPH-NE-Pol	Graph-SAGE-NE	Placeto
Training time per placement sample (sec)	0.565± 0.353	0.145± 0.049	0.178± 0.064	0.360 ± 0.257	0.027 ± 0.011	0.157 ± 0.075	0.255 ± 0.040
Running time per placement sample (sec)	0.340± 0.256	0.114± 0.026	0.132± 0.035	0.240 ± 0.187	0.027 ± 0.008	0.138 ± 0.051	0.162 ± 0.051

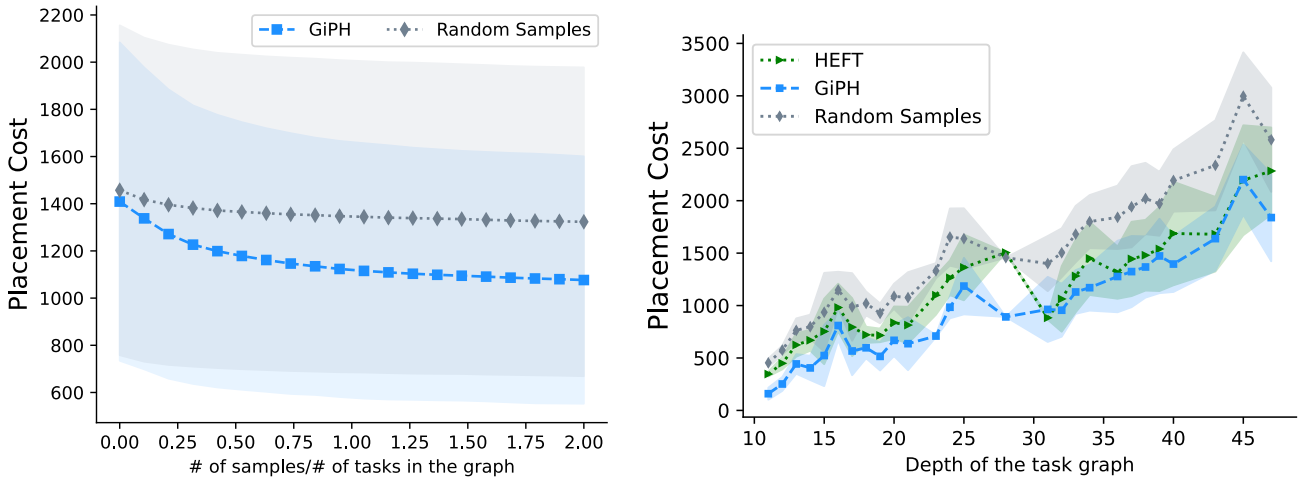


Figure 16. The total communication and computation cost of the placement found by HEFT, GiPH, and random sampling, as a function of depth of the task graph.

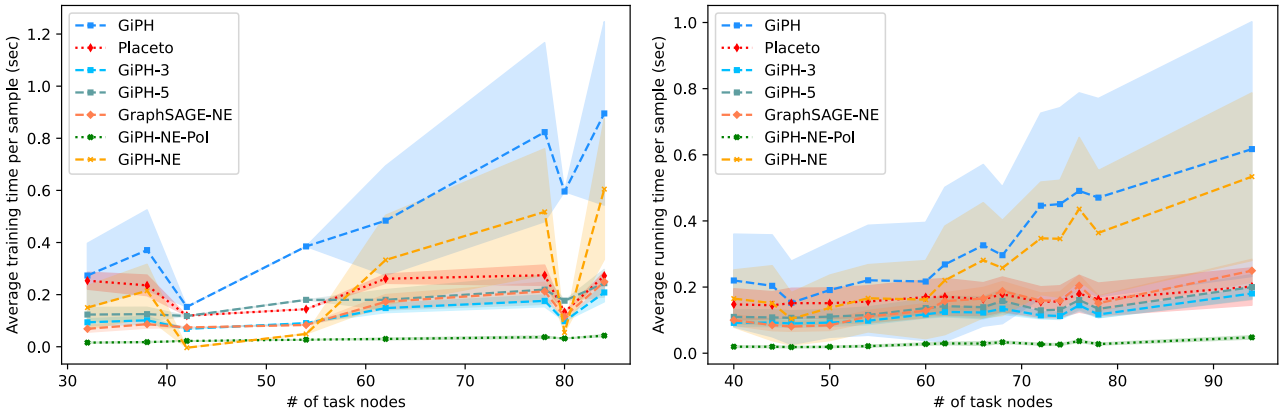


Figure 17. Left: Average training time averaged over all placement samples taken during training. Right: Average policy running time averaged over all placement samples taken during testing.