GCNScheduler: Scheduling Distributed Computing Applications using Graph Convolutional Networks

Mehrdad Kiamari
University of Southern California
Los Angeles, USA
kiamari@usc.edu

Bhaskar Krishnamachari
University of Southern California
Los Angeles, USA
bkrishna@usc.edu

ABSTRACT

We provide a highly-efficient solution to the classical problem of scheduling task graphs corresponding to complex applications on distributed computing systems. A number of heuristics have been previously proposed to optimize task scheduling with respect to different metrics (e.g. makespan and throughput). However, they tend to be slow to run, particularly for larger problem instances, limiting their applicability in more dynamic systems. Motivated by the goal of solving these problems more rapidly, we propose, for the first time, a graph convolutional network-based scheduler (GCNScheduler). By carefully integrating the inter-task data dependency structure and the computational network into a single input graph, the GCNScheduler can efficiently schedule tasks of complex applications for a given objective. We use simulations to illustrate that not only can our scheme quickly and efficiently learn from existing scheduling schemes, but also it can easily be applied to large-scale settings that current scheduling schemes fail to handle. We demonstrate the generalization of GCNScheduler to unseen real-world applications and show that it achieves almost the same makespan and throughput as benchmarks, while providing several orders of magnitude faster scheduling times.

CCS CONCEPTS

• **Computing methodologies** → Neural networks.

KEYWORDS

Graph Convolutional Networks, GCNScheduler.

ACM Reference Format:
Mehrdad Kiamari and Bhaskar Krishnamachari. 2022. GCNScheduler: Scheduling Distributed Computing Applications using Graph Convolutional Networks. In *Graph Neural Networking Workshop (GNNet'22), December 9, 2022, Roma, Italy*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3565473.3569185

1 INTRODUCTION

Successfully running complex graph-based applications, ranging from edge-cloud processing in IoT systems [4, 15] to processing astronomical observations [10], heavily relies on executing all sub-components of such applications through an efficient task-scheduling. Not only does efficient task scheduling play a crucial role in improving the utilization of computing resources and reducing the required time to executing tasks, it can also lead to significant profits to service providers [3]. In this framework, any application consists of multiple tasks with a given inter-task data dependency structure, i.e. each task generates inputs for certain other tasks. Such dependencies can be expressed via a directed acyclic graph (DAG), also known as *task graph*, where vertices and edges represent tasks and inter-task data dependencies, respectively. An input job for an application is completed once all the tasks are executed by compute machines according to the inter-task dependencies.

There are two commonly used metrics for schedulers to optimize: makespan and throughput. The required time to complete all tasks for a single input is called the makespan. The maximum steady state rate at which inputs can be processed in a pipelined manner is called throughput. Makespan minimization and throughput maximization can each be achieved through relevant efficient *task-scheduling* algorithms that assign tasks to appropriate distributed computing resources to be executed.

The underlying methodology for task scheduling can be categorized into heuristic-based (e.g. [7]- [19]), meta-heuristic ones (e.g. [1]-[8]) , and optimization-based schemes (e.g. [2]). One of the most well-known heuristic scheduling schemes for makespan minimization is the heterogeneous earliest-finish time (HEFT) algorithm [28], which will be considered as one of our benchmarks. For throughput maximization, we benchmark against the algorithm presented in [9] which we refer to as TP-HEFT.

One of the main disadvantages of all the above-mentioned scheduling schemes is that they work well only in relatively small settings; once a task graph becomes large or extremely large, they require very long computation times. We anticipate that applications in many domains, such as IoT for smart cities will result in increasingly complex applications with numerous inter-dependent tasks, and scheduling may need to be repeated quite frequently in the presence of network or resource dynamics [6, 27]. Therefore, it is essential to design a faster method to schedule tasks for such large-scale task graphs.

A promising alternative is to apply machine learning techniques for function approximation to this problem, leveraging the fact that scheduling essentially has to do with finding a function mapping tasks to compute machines. Given the graph structure of applications, we propose to use an appropriate graph convolutional network (GCN) [17] to schedule tasks through learning the inter-task dependencies of the task graph as well as network settings (i.e.,

GNNet'22, December 9, 2022, Roma, Italy

ACM ISBN 978-1-4503-9933-3/22/12.
https://doi.org/10.1145/3565473.3569185

execution speed of compute machines and communication bandwidth across machines) in order to extract the relationship between different entities. The GCN has attracted significant attention in the literature for its ability in addressing many graph-based applications to perform semi-supervised link prediction [29] and node classification [17]. The idea behind GCN is to construct node embeddings layer by layer. In each layer, a node embedding is achieved by aggregating its neighbors' embeddings, followed by a neural network.

To the best of our knowledge, there is no prior work that has proposed a pure GCN, incorporated with carefully-designed the features of both nodes and edges for task graphs, to perform scheduling over distributed computing systems.

The main contributions of this paper are as follows:

- We propose GCNScheduler, which can quickly schedule tasks by carefully integrating a task graph with network settings into a single input graph and feeding it to a GCN model suitable for directed graphs.
- Any existing scheduling algorithm can be used as a teacher to train GCNScheduler, for any metric. We illustrate this by training GCNScheduler using HEFT for makespan minimization, and TP-HEFT for throughput maximization.
- We show that GCNScheduler gives comparable scheduling performance as the teacher algorithm, i.e. in terms of makespan with respect to HEFT and in terms of throughput with respect to TP-HEFT, respectively.
- We show that our GCNScheduler can be trained in a very short period of time, for instance, it takes around a minute to train the model using a graph with 5,000 nodes.
- We show that GCNScheduler is several orders of magnitude faster than previous algorithms in obtaining the schedule for a given task graph. For example, for makespan minimization, GCNScheduler schedules 100-node task graphs in about 3.8 milliseconds while READYS and HEFT respectively takes around 288 milliseconds and 25 seconds; and for throughput maximization, GCNScheduler schedules 200-node task graphs in about 4 milliseconds, compared to about 27 seconds for TP-HEFT.
- We show that GCNScheduler is able to efficiently perform scheduling for *any size* task graph. In particular, we show that our proposed scheme is able to operate over large-scale task graphs where existing schemes require excessive computational resources.

2 RELATED WORK

Heuristic, meta-heuristic, and optimization-based are three categories of task scheduling schemes. Since heuristic algorithms (e.g. [21]) may sometimes perform poorly compared to optimal task scheduling, meta-heuristic (e.g. Particle Swarm Optimization [16], Simulated Annealing [1], Genetic-based approach [23]) and optimization-based schemes (e.g. [2]), which aim at approximating the NP-hard optimization of task scheduling, have attracted significant attention. However, all the above-mentioned schemes tend to run extremely slowly as number of tasks becomes large due to iterative nature of these methods, which requires excessive computations. Moreover, this issue makes the aforementioned schemes unable to handle large-scale task graphs.

As obtaining the optimal scheduler is basically the same as finding an appropriate mapper function, which maps tasks to compute machines, machine-learning based scheduling has begun emerging as an alternative thanks to advances in fundamental learning methods, such as deep learning [11] and reinforcement learning (RL) [26]. Sun *et. al.* proposed DeepWave [25], a scheduler which reduces job completion time using RL while specifying a priority list [1] as the action and the completion time of a job DAG as the reward. Furthermore, Decima [18] schedules tasks over a spark cluster by training a neural network using RL with scheduling the next task for execution as the action and a high-level scheduling objective of minimizing the makespan as the reward. The aforementioned RL-based schemes suffer from having a huge action space (i.e., the space of scheduling decisions).

While Decima [18] only operates in homogeneous environments, Grinsztajn *et. al.* proposed READYS [12] to operate over heterogeneous environments. READYS combines two different components, a GCN and actor-critic algorithm. There are three main differences from our work with respect to their use of GCN: first, they use the GCN to embed task nodes only without taking network settings into account as we do; second, they use a regular GCN which does not explicitly account for directed nodes while we use an EDGNN [13] which does; and finally, the GCN in READYS does not do scheduling (their GCN only does task node embedding, while the scheduling is done via the actor-critic algorithm), whereas we are the first to propose to use a GCN directly for task scheduling.

3 PROBLEM STATEMENTS

We now elaborate upon formally representing the minimization of makespan and the maximization of throughput as optimization problems. In order to finish a job, all its tasks require to be executed at least on a single compute machine. Before expressing the definition of makespan and throughput, let us explain about task dependencies, referred to as *task graph*, and network settings.

Task Graph: Every application/job is comprised of inter-task data dependencies. Since there are dependencies across different tasks, meaning that a task generates inputs for certain other tasks, we can model this dependency through a *DAG*.Suppose we have N_T tasks $\{T_i\}_{i=1}^{N_T}$ with a given task graph $G_{Task} := (V_{Task}, E_{Task})$ where $V_{Task} := \{T_i\}_{i=1}^{N_T}$ and $E_{Task} := \{e_{i,i'}\}_{(i,i')\in\Omega}$ respectively represent the set of vertices and edges (task dependencies) with $\Omega := \{(i,i')|$ if task T_i generates inputs for task $T_{i'}$ $\}$. Let us define vector $\mathbf{p} := [p_1, \ldots, p_{N_T}]^T$ as the amount of computations required by tasks. For every tasks T_i and T_j, $\forall i, j$ where $e_{i,j} \in E_{Task}$, task T_i produces $d_{i,j}$ amount of data for task T_j after being executed by a machine.

Network Settings: Each task is required to be executed on a *compute* node (machine) which is connected to other compute nodes (machines) through communication links (compute node and machine are interchangeably used in this paper). Let us suppose to have N_C compute nodes $\{C_j\}_{j=1}^{N_C}$. Regarding the execution speed

[1] Which indicates the scheduling priority of edges in a job DAG.

of compute nodes, we consider vector $\mathbf{e} := [e_1, \ldots, e_{N_C}]^T$ as the executing speed of machines. The communication link delay between any two compute nodes can be characterized by bandwidth. Let us denote $B_{i,j}$ as the communication bandwidth of the link from compute node C_i to compute node C_j. In case of two machines not being connected to each other, we can assume the corresponding bandwidth is zero (infinite time for communication delay).

In general, a task-scheduling scheme maps tasks to compute nodes according to a given objective. Formally speaking, a task scheduler can be represented as a function $m(.) : V_{Task} \rightarrow \{C_k\}_{k=1}^{N_C}$ where task T_i, $\forall i$, is assigned to machine $m(T_i)$. The main goal is to find a scheduler that assigns tasks to compute machines with respect to optimizing a given objective (e.g. makespan minimization). Our proposed scheme aims at obtaining such scheduler by utilizing a suitable GCN (incorporated with carefully-designed features) where it is able to classify tasks into machines[2].

4 PROPOSED GCNSCHEDULER

We present a novel machine-learning based task scheduler which can be trained with respect to aforementioned objectives. To capture the underlying graph-based relationships of the task-scheduling problem, we employ a suitable GCN[13], in order to obtain a model which can automatically assign tasks to compute machines. One unique challenge in this problem is that there are two different graphs that need to be considered - a task graph and the compute network graph. We address this challenge by combining the two graphs into one graph where there is a node for each node in the task graph, but the node features and edge features combine the features of both the task graph and compute network graph. This unified graph is then used as the basis for the GCN training and inference. There is no prior GCN-based scheduling scheme that has done such a unification. This novel idea has significant advantages over the conventional scheduling schemes. First, it can remarkably reduce the computational complexity compared to previous scheduling algorithms. Second, after training an appropriate GCN, our scheme can handle any large-scale task graph while conventional schemes severely suffer from the scalability issue.

4.1 Overview of GCNs for Directed Graphs

A suitable framework for problems that have to do with directed graphs is EDGNN [13] where incoming and outgoing edges are treated differently in order to capture nonreciprocal relationship between nodes. In particular, the embedding of node v would be as follows:

$$\mathbf{h}_{n,v}^{(t)} = \sigma(\mathbf{W}_1^{(t)}\mathbf{h}_{n,v}^{(t-1)} + \mathbf{W}_2^{(t)} \sum_{u:u\in\mathcal{N}(v)} \mathbf{h}_{n,u}^{(t-1)} + \mathbf{W}_3^{(t)} \sum_{u:e_{u,v}\in E_{Task}} \mathbf{h}_{e,(u,v)}^{(t-1)} + \mathbf{W}_4^{(t)} \sum_{u:e_{v,u}\in E_{Task}} \mathbf{h}_{e,(v,u)}^{(t-1)}), \quad (1)$$

where $\mathbf{W}_1^{(t)}$, $\mathbf{W}_2^{(t)}$, $\mathbf{W}_3^{(t)}$, and $\mathbf{W}_4^{(t)}$ represent weight matrices of layer t for embedding of self node, neighboring nodes, incoming edges, and outgoing edges, respectively. Moreover, $\mathbf{h}_{n,v}^{(t)}$ and $\mathbf{h}_{e,(u,v)}^{(t)}$ respectively denote embedding of node v and the embedding of the edge from node u to node v at layer t.

[2]Each machine represents a class in our problem.

4.2 Proposed Input Graph

In order to train an EDGNN-based model, we need to carefully design the input graph components, namely adjacency matrix, nodes' features, edges' features, and labels. It should be noted that our scheme is not tailored to a particular criterion as we will show later that it can learn from two scheduling schemes with different objectives. Our designed input graph can be fed into the EDGNN and the model will be trained according to labels generated from a given scheduling scheme. We next explain how we carefully design the input graph.

Designed Input Graph: We start from the original task graph and consider the same set of nodes and edges for our input graph as the task graph. In other words, by representing the input graph as $G_{input} := (V_{input}, E_{input})$, we have $V_{input} := V_{Task}$ and $E_{input} := E_{Task}$. The crucial part for having an efficacious GCN-based scheduler has to do with carefully designing the features of nodes and edges as well as the labels. According to the definition of makespan in [28] and throughput in [9], these objectives are a function of the required computational time of all tasks across all machines (i.e. $\{\frac{p_i}{e_j}\}_{\forall i,j}$) and the required communication delay to transfer the result of executing all tasks to their successor tasks across all pair-wise machines. Therefore, we incorporate the following features:

- The feature of node T_i, $\forall T_i \in V_{input}$, is denoted by $\mathbf{x}_{\mathbf{n},i} := \left(\frac{p_i}{e_1}, \frac{p_i}{e_2}, \cdots, \frac{p_i}{e_{N_C}}\right)^T \in \mathbb{R}^{N_C}$. The intuition behind $\mathbf{x}_{\mathbf{n},i}$ is that these features represent the required computational time of task T_i across all compute machines.
- The feature of edge $e_{u,v}$, $\forall e_{u,v} \in E_{input}$, is denoted by $\mathbf{x}_{\mathbf{e},(u,v)}$ and it has the following features:

$$\mathbf{x}_{\mathbf{e},(u,v)} := \left(\frac{d_{u,v}}{B_{1,1}}, \frac{d_{u,v}}{B_{1,2}}, \cdots, \frac{d_{u,v}}{B_{N_C,N_C}}\right)^T \in \mathbb{R}^{N_C^2}.$$

The intuition behind $\mathbf{x}_{\mathbf{e},(u,v)}$ is that these features represent the required time for transferring the result of executing task T_u to the following task T_v across all possible pair-wise compute machines. As far as ablations studies are concerned, the makespan of a real-world application of [5] on a network of machines (where one of them has poor connections to others) is 0.5844s, while the makespan degrades to 1.5844s (×2.71) and 3.5845s (×6.13) for the case of removing nodes features and edges features, respectively.

Objective-Dependent Labeling: Based on what task scheduler our method should learn from (which we refer to as the "teacher" scheduler, namely, HEFT for makespan minimization and TP-HEFT for throughput maximization), we label all nodes as well as edges. Let us define $L_{\mathbf{n},v}$ and $L_{\mathbf{e},(u,v)}$ as labels of node v and edge $e_{u,v}$, respectively. Regarding nodes' labeling, we consider the label of node T_i, $\forall i$, as the index of compute node that the teacher algorithm assigns task T_i to run on. We label each edge according to the label of the ending vertex it is from. In other words, $L_{\mathbf{e},(u,v)} = L_{\mathbf{n},u}$ $\forall u, v$ such that $e_{u,v} \in E_{input}$. We should note that this edge-labeling is crucial in enforcing the model to learn to label out-going edges of a node with same label as its corresponding node's label. We find empirically that the alternative of labeling in-coming edges with a node's label results in performance degradation.

4.3 Implementation and Training

As far as the model parameters are concerned, we consider a 3-layer EDGNN with 32 nodes per layer and *LeakyReLU* activation function. Since we suppose nodes and edges have features, we let both nodes and edges to be embedded.

We train GCNScheduler with respect to each of makespan minimization and throughput maximization only once. For a given objective (either makespan or throughput), we consider dataset $\{G^{(i)}_{input}, \mathbf{L}^{(i)}\}_{i=1}^{N_g}$ for sufficiently large N_g (e.g. $N_g = 100$) where $\mathbf{L}^{(i)}$ represents the labels (which comes from the teacher scheduler) of nodes and edges of ith input graph $G^{(i)}_{input}$. Regarding input graphs, $G^{(i)}_{input}$ is constructed from task graph $G^{(i)}_{Task}$, tasks computation amount $\mathbf{p}_i$, machines execution speed $\mathbf{e}_i$, and communication bandwidth matrix $\mathbf{B}_i$. The components of different input graphs are set independently from each other. Regarding task graphs $G^{(i)}_{Task}$'s, we generate DAGs randomly. Furthermore, since the HEFT and TP-HEFT algorithms are extremely slow in performing task-scheduling for large-scale task graphs, obtaining labels (i.e. determining the machine each task needs to be executed on) for a single large graph is cumbersome. Therefore, we use medium size task graphs (with $\leq$ 50 tasks for each) such that HEFT and TP-HEFT can handle scheduling tasks over each of them.

With respect to network settings of input graphs, we let network operate in both normal heterogeneous mode with bandwidths and execution speeds randomly drawn from uniform distribution (for 80% of input graphs) and straggler mode (for 20% of input graphs) where one or more machines experience either extremely poor execution speed or extreme poor communication bandwidths to other machines.

We use a batch size of 16, a learning rate of 10^{-3}, and Adam optimizer. We also train the model for at most 50 epochs with early stopping with respect to the validation set cross-entropy loss. All our experiments are performed with 10-fold cross validation. The accuracy of GCNScheduler in labeling the nodes with respect to the schedules produced by the teacher scheduler is around 70%; however, we should note that ultimately the makespan or throughput are more important rather than the accuracy of labeling individual tasks. As far as training time, it only takes less than a minute[3] for the model to be trained with respect to a given objective. This is a key advantage of GCNScheduler compared to RL-based schedulers such as READYS [12] which take around 20 minutes to train their models.

5 EXPERIMENTAL RESULTS

We measure the performance of GCNScheduler (in terms of the makespan and the throughput) as well as time to find schedule (TTFS). We further compare the performance of GCNScheduler against our benchmarks (i.e. HEFT and READYS for makespan minimization, while TP-HEFT and the random task-scheduler for throughput maximization). We evaluate all schemes by running them on our local cluster which has 16 CPUs (with 8 cores and 2 threads per core) of Intel(R) Xeon(R) E5-2620 v4 @ 2.10GHz.

[3]On our local cluster which has 16 CPUs (with 8 cores and 2 threads per core) of Intel(R) Xeon(R) E5-2620 v4 @ 2.10GHz.

In order to show that GCNScheduler can generalize well, we test our trained model on both synthetic (medium-scale and large-scale) task graphs as well as the task graphs of real-world applications. Regarding the applications, we use Cycles [24] (time-step simulations of crop production and the water, carbon (C) and nitrogen (N) cycles in the soil plant atmosphere continuum), Epigenomics [14] (executing various genome sequencing operations), Montage [22] (an astronomical image mosaic engine), Seismology [5], and the task graphs of three real-world perception applications, namely Face recognition, Object-and-pose recognition, and Gesture recognition, presented in [20]. For simplicity, we assume each task produces the same amount of data after being executed. As far as network settings are concerned, we consider 40 machines where their execution speeds and the communication bandwidths are drawn randomly from uniform distributions.

5.1 Makespan Minimization

We measure the performance of trained GCNScheduler (with respect to makespan minimization) on the task graphs of real-world applications provided in [5, 14, 20, 22, 24]. Table 1 shows the makespan and TTFS of GCNScheduler (with the makespan minimization objective) against HEFT and READYS for the seven real-world applications. While our GCNScheduler leads to almost the same makespan compared to benchmarks, it reduces the time taken to find the schedule by several orders of magnitude as it is shown in Table 1.

5.2 Throughput Maximization

Similarly, we measure the throughput of GCNScheduler over the task graph of real-world applications and compare it against benchmarks. Table 2 shows the throughput and the time taken to find the schedule of GCNScheduler compared to TP-HEFT for the real-world applications. While GCNScheduler leads to a marginally better throughput performance compared to TP-HEFT scheduler, it significantly (2-3 orders of magnitude) reduces the time taken to perform task-assignment as it is shown in Table 2.

6 CONCLUSION

We proposed GCNScheduler, a scalable and fast task-scheduling scheme which can perform scheduling according to different objectives. Not only can GCNScheduler easily handle large-scale settings where existing scheduling schemes are unable to do, but also it can lead to almost the same performance with several orders of magnitude lower required time to schedule.

7 ACKNOWLEDGMENTS

This material is based upon work supported in part by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0053 and the Army Research Laboratory under Cooperative Agreement W911NF-17-2-0196. Any views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] S. Addya, A. Turuk, B. Sahoo, M. Sarkar, and S. Biswash. 2017. Simulated annealing based VM placement strategy to maximize the profit for Cloud Service Providers.

Table 1: Makespan and time to find schedule (TTFS) (in seconds) of different schedulers for real-world applications.

Application	Scheduler					
	HEFT		READYS		GCNScheduler	
	Makespan	TTFS(s)	Makespan	TTFS(s)	Makespan	TTFS(s)
Face Recog. [20]	0.248	0.3543	0.244	0.0437	0.248	**0.0043**
Pose Recog. [20]	0.222	0.6583	0.219	0.0493	0.222	**0.0044**
Gesture Recog. [20]	0.744	0.7015	0.731	0.0562	0.756	**0.0050**
Montage [22]	633.401	14.6947	630.523	1.4902	631.740	**0.0061**
Epigenomics [14]	111.521	8.1977	110.926	0.9645	112.167	**0.0058**
Cycles [24]	237.566	2.3316	235.997	0.2725	237.752	**0.0057**
Seismology [5]	3.549	0.4607	3.112	0.0536	3.149	**0.0042**

Table 2: Throughput and time to find schedule (TTFS) in (milliseconds) of different schedulers for real-world applications.

Application	Scheduler			
	TP-HEFT		GCNScheduler	
	TP	TTFS(ms)	TP	TTFS(ms)
Face Recog.	1.198	87.34	1.592	**0.488**
Pose Recog.	1.245	257.8	1.578	**0.511**
Gesture Recog.	1.461	290.1	1.462	**0.560**
Montage	1.474	5,832.337	1.497	**5.596**
Epigenomics	2.581	26,444.576	2.542	**5.799**
Cycles	8.541	4,267.352	8.372	**5.289**
Seismology	3.853	9,971.013	3.874	**4.803**

Engineering Science and Technology, an International Journal 20 (2017), 1249–1259.

[2] Y. Azar and A. Epstein. 2005. Convex Programming for Scheduling Unrelated Parallel Machines. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing* (Baltimore, MD, USA). Association for Computing Machinery, New York, NY, USA, 331–337.

[3] L. André Barroso, J. Clidaras, and U. Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.*

[4] N. Cheng, F. Lyu, W. Quan, C. Zhou, H. He, W. Shi, and X. Shen. 2019. Space/Aerial-Assisted Computing Offloading for IoT Applications: A Learning-Based Approach. *IEEE Journal on Selected Areas in Communications* 37 (2019), 1117–1129.

[5] T. Coleman, H. Casanova, L. Pottier, M. Kaushik, E. Deelman, and R. Silva. 2022. WfCommons: A Framework for Enabling Scientific Workflow Research and Development. *Future Generation Computer Systems* (2022).

[6] S. Dustdar, S. Nastić, and O. Šćekić. 2017. Smart Cities. In *The Internet of Things, People and Systems.* Springer.

[7] L. Eskandari, J. Mair, Z. Huang, and D. Eyers. 2018. Iterative Scheduling for Distributed Stream Processing Systems. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems* (Hamilton, New Zealand). 234–237.

[8] Z. Fan, H. Shen, Y. Wu, and Y. Li. 2013. Simulated-Annealing Load Balancing for Resource Allocation in Cloud Environments. In *2013 International Conference on Parallel and Distributed Computing, Applications and Technologies.* 1–6. https://doi.org/10.1109/PDCAT.2013.7

[9] M. Gallet, L. Marchal, and F. Vivien. 2009. Efficient scheduling of task graph collections on heterogeneous resources. In *IEEE International Symposium on Parallel Distributed Processing.* 1–11.

[10] Garcia-Piquer, A., Morales, J. C., Ribas, I., Colomé, J., Guàrdia, J., Perger, M., Caballero, J. A., Cortés-Contreras, M., Jeffers, S. V., Reiners, A., Amado, P. J., Quirrenbach, A., and Seifert, W. 2017. Efficient scheduling of astronomical observations - Application to the CARMENES radial-velocity survey. *A&A* 604 (2017), A87.

[11] I. Goodfellow, Y. Bengio, and A. Courville. 2016. *Deep Learning.* MIT Press.

[12] N. Grinsztajn, O. Beaumont, E. Jeannot, and P. Preux. 2021. READYS: A Reinforcement Learning Based Strategy for Heterogeneous Dynamic Scheduling. In *2021 IEEE International Conference on Cluster Computing (CLUSTER).* 70–81.

[13] G. Jaume, A. Nguyen, M. Rodríguez Martínez, J. Thiran, and M. Gabrani. 2019. edGNN: a Simple and Powerful GNN for Directed Labeled Graphs. arXiv:1904.08745 [cs.LG]

[14] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. 2013. Characterizing and profiling scientific workflows. *Future Generation Computer Systems* 29 (2013).

[15] K. Kaur, S. Garg, G. Aujla, N. Kumar, J. Rodrigues, and M. Guizani. 2018. Edge Computing in the Industrial Internet of Things Environment: Software-Defined-Networks-Based Edge-Cloud Interplay. *IEEE Communications Magazine* 56, 2 (2018), 44–51.

[16] J. Kennedy and R. C. Eberhart. 1997. A discrete binary version of the particle swarm algorithm. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, Vol. 5. 4104–4108 vol.5.

[17] T. Kipf and M. Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. arXiv:1609.02907 [cs.LG]

[18] H. Mao, M. Schwarzkopf, S. Venkatakrishnan, Z. Meng, and M. Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China). New York, NY, USA.

[19] X. Pham, N. Man, N. Dao, N. Quang, and E. Huh. 2017. A cost- and performance-effective approach for task scheduling based on collaboration between cloud and fog computing. *International Journal of Distributed Sensor Networks* 13, 11 (2017), 1550147717742073. arXiv:https://doi.org/10.1177/1550147717742073

[20] M. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. 2011. Odessa: Enabling Interactive Perception Applications on Mobile Devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services.* ACM, NY, USA, 43–56.

[21] H. Ren, Y. Lan, and C. Yin. 2012. The load balancing algorithm in cloud computing environment. In *Proceedings of 2012 2nd International Conference on Computer Science and Network Technology.* 925–928.

[22] M. Rynge, G. Juve, J. Kinney, J. Good, G. Berriman, A. Merrihew, and E. Deelman. 2013. Producing an Infrared Multiwavelength Galactic Plane Atlas using Montage, Pegasus and Amazon Web Services. In *23rd Annual Astronomical Data Analysis Software and Systems Conference.*

[23] Y. Shishido, J. Estrella, C. Toledo, and M. Arantes. 2018. Genetic-based algorithms applied to a workflow scheduling algorithm with security and deadline constraints in clouds. *Computers and Electrical Engineering* 69 (2018).

[24] R. Silva, R. Mayani, Y. Shi, A. Kemanian, M. Rynge, and E. Deelman. 2019. Empowering Agroecosystem Modeling with HTC Scientific Workflows: The Cycles Model Use Case. In *2019 IEEE International Conference on Big Data (Big Data).*

[25] P. Sun, Z. Guo, J. Wang, J. Li, J. Lan, and Y. Hu. 2020. DeepWeave: Accelerating Job Completion Time with Deep Reinforcement Learning-based Coflow Scheduling. In *IJCAI.*

[26] R. Sutton and A. Barto. 2018. *Reinforcement Learning: An Introduction.* A Bradford Book, Cambridge, MA, USA.

[27] A. Syed, D. Sierra, A. Kumar, and A. Elmaghraby. 2021. IoT in Smart Cities: A Survey of Technologies, Practices and Challenges. *Smart Cities* 4 (2021), 429–475.

[28] H. Topcuoglu, S. Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (2002), 260–274.

[29] M. Zhang and Y. Chen. 2018. Link Prediction Based on Graph Neural Networks. arXiv:1802.09691 [cs.LG]