Graph Convolutional Network-based Scheduler for Distributing Computation in the Internet of Robotic Things

Jared Coleman*, Mehrdad Kiamari†, Lillian Clark†, Daniel D'Souza*, Bhaskar Krishnamachari†*
*Dept. of Computer Science
†Dept. of Electrical and Computer Engineering
University of Southern California Los Angeles, USA
{jaredcol, kiamari, lilliamc, dmdsouza, bkrishna}@usc.edu,

***Abstract*—Existing solutions for scheduling arbitrarily complex distributed applications on networks of computational nodes are insufficient for scenarios where the network topology is changing rapidly. New Internet of Things (IoT) domains like the Internet of Robotic Things (IoRT) and the Internet of Battlefield Things (IoBT) demand solutions that are robust and efficient in environments that experience constant and/or rapid change. In this paper, we demonstrate how recent advancements in machine learning (in particular, in graph convolutional neural networks) can be leveraged to solve the task scheduling problem with decent performance and in much less time than traditional algorithms.**

***Index Terms*—GCN, Graph Convolutional Network, Scheduling, Internet of Things, Robotics**

I. Introduction

The Internet of Battlefield Things (IoBT) is a subdomain of IoT which considers connected devices (sensors, actuators, compute-nodes, etc.) that support critical military operations [15]. Common assumptions about network connectivity, node reliability, and environmental conditions that are acceptable for civilian IoT applications are unacceptable in high-stakes battlefield environments. Motivated by the IoBT domain, we study the problem of scheduling arbitrarily complex distributed applications over resource-constrained mobile patrol robots. The mobility of the patrolling network nodes results in dynamic communication conditions and demands careful attention to the compute resources available. We consider distributed applications which are modeled as directed acyclic task graphs. Each node in a task graph represents a compute task and edges between tasks represent a dependency relationship (i.e., one task cannot start until its dependencies have terminated).

The purpose of a scheduler is to determine where and when tasks should execute in order to minimize some metric like *makespan* (total execution time). Efficient schedulers must balance the cost and benefit of sending tasks to execute in parallel on different compute nodes. On one hand, sending a task and its input data to execute on another node takes time while on the other, executing two tasks in parallel *saves* time. For this paper, we constructed a task graph that resembles many real-world applications and is a good demonstration of this trade-off (Figure2).

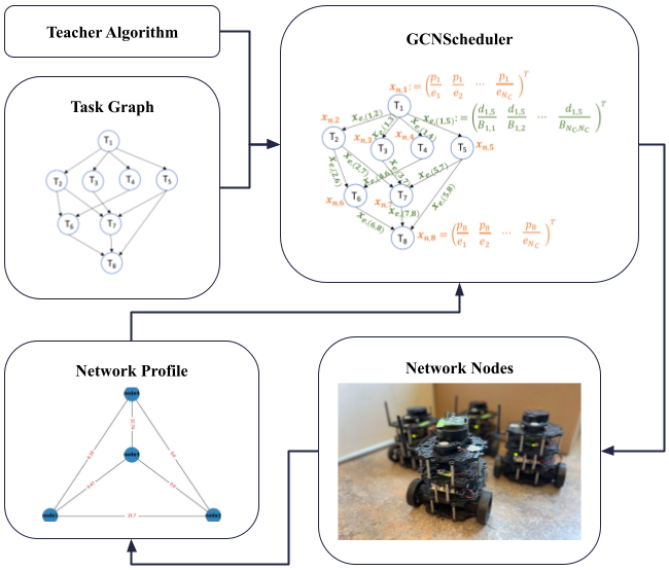

Fig. 1. An overview of our approach: Task graphs are augmented with node and edge feature vectors that encode data from the network profile. GCNScheduler learns to imitate a teacher algorithm and assign tasks to network nodes.

Deploying complex distributed applications over a network of mobile compute nodes is challenging when the network experiences constant and/or rapid change. A schedule that minimizes makespan for the network at a particular time may not be optimal (or anywhere near optimal) for the network at a future time. Existing scheduling algorithms that leverage the entire network state suffer with respect to complexity as the number of tasks and/or network nodes increase. In order to adapt to the rapid dynamics of the IoBT domain, a scheduling algorithm needs to be capable of computing and recomputing complex schedules quickly. More efficient online algorithms that only consider partial network information (i.e. neighborhood information) have been proposed [4], [7], [24]. These algorithms produce schedules quickly and dynamically but often result in larger makespans since they do not leverage all the information available about a network.

Contribution In this paper, we demonstrate how recent advancements in artificial intelligence can be applied to solve practical problems in the Internet of Robotic Things domain. We demonstrate through simulation that Graph Convolutional Networks can learn to produce schedules with low makespans quickly, making it suitable for IoBT applications.

II. Related Work

Efficient task scheduling is a well-researched area and finding an optimal schedule is generally NP-hard [23]. Some approaches seek to optimize directly via convex programming [3] or semidefinite programming [18]. Other approaches rely on heuristics like priority-based scheduling [19] or load balancing [12]. One of the most popular scheduling algorithms is Heterogeneous Earliest-Finish-Time (HEFT) [22]. Online and distributed scheduling algorithms have also been proposed [4], [7], [24]. More recently, metaheuristic-based approaches like simulated annealing [1] and genetic algorithms [8], [17] have been proposed. The primary disadvantage of existing approaches is that scheduling time grows quickly with the size and complexity of the network and application.

Task scheduling on IoT systems has unique challenges: nodes tend to be more severely compute constrained, given their size, and more severely communication constrained, given their wireless nature. Recent work on scheduling perception tasks over an IoT network focuses on how to prioritize tasks based on criticality [13], or optimize makespan using particle swarm optimization [6]. For IoBT settings, highly-volatile network dynamics impose the need for quick schedule computation. Learning-based algorithms can reduce computation times to address this challenge, for example by training a scheduling algorithm to minimize makespan [14], [20]. Despite this, a recent survey of scheduling algorithms for IoT indicates that most work is concerned with static schedules and only a few rely on learning [2].

To effectively exploit the relational information of graph-structured data, graph neural networks (GNNs) have recently become a popular method for approaching optimization problems in wireless networks [14]. In our previous work, we explored GNNs as a method for developing task scheduling algorithms for wireless networks [11], focusing on static networks and scalability. In this work, we focus on dynamic networks with more realistic simulations of inter-node communication and specifically address the feasibility of GNNs for task scheduling in the IoBT domain.

III. Problem Formulation

Consider a system of compute-capable mobile robots, each of which is moving with equal speed and in the same direction along the perimeter of an arbitrary simple polygon in two-dimensional space, and a distributed application that can be modeled as a directed acyclic graph (DAG) of tasks $G = (V, E)$ where each $v \in V$ represents a computational task and each $(u, v) \in E$ represents a dependency between tasks u and v (one or more outputs of task u are necessary inputs to task v). We denote the computational cost of a task $v \in V$ as $\texttt{cost}(v)$ and the size of the data-dependency between u and v for some $(u, v) \in E$ as $\texttt{data}(u, v)$. While the trajectories of the robots may have nothing to do with the distributed application, they do have a direct impact on the communication ability of the mobile robots in the network. Since robots are moving, we model their pairwise communication rates as a function of their positions in the plane. Consider some time $t \geq 0$ and let $N = (R, C)$ denote the robotic communication network at that time. Each $r \in R$ represents a single robot and each edge $(r, s) \in C$ corresponds to the communication channel between r and s at time t. We denote communication rate between robots r and s for every $(r, s) \in C$ as $\texttt{comm}(r, s)$. Finally, we use $\texttt{comp}(r)$ to denote the compute speed of each robot $r \in R$.

Our goal is to schedule tasks to execute on compute nodes (robots) in order to minimize the makespan (total execution time) of the distributed application (task graph) under dynamic network conditions.

IV. Background

In this section, we provide a brief overview of HEFT and our graph convolutional network-based scheduler.

A. Heterogeneous Earliest Finish Time (HEFT)

HEFT is a greedy algorithm that assigns tasks to the processor which, given all previously assigned tasks, would result in the earliest finish time of the task [22]. HEFT aims at minimizing the *makespan* according to the following formal definitions.

Definition 1 (Earliest Start Time)**.** *$EST(u, r)$ denotes the earliest execution start time for task u being executed on compute node (robot) r. Note that if u is a task with no predecessors (a root task), then $EST(u, r) = 0,\ \forall r \in R$.*

Definition 2 (Earliest Finish Time)**.** *$EFT(u, r)$ denotes the earliest execution finish time for task u being executed on compute node r.*

Definition 3 (Actual Start and Finish Times)**.** *$AST(u)$ and $AFT(u)$ denote the actual start time and the actual finish time of task u (depending on which node it actually gets scheduled to execute on).*

Definition 4 (Makespan)**.** *After all tasks are assigned to compute nodes for execution, the makespan is given by:*

$$\max_{u \in V} \{AFT(u)\}.$$

HEFT works by computing the EFT recursively for each task on each node, using an insertion-based policy whereby it considers the first available and sufficiently large window (for a task u and node r, any windows larger than $\texttt{cost}(u)/\texttt{comp}(r)$). The order in which HEFT assigns tasks is based on each task's *rank*, which is its average compute time (across all compute nodes) plus the maximum communication-adjusted rank of all its parent tasks. The communication-adjusted rank for a task with respect to a child task is the sum of its rank and the average communication time required

(across all network connections) for the data dependency between the tasks. Essentially, the rank is a topological sort of the task graph that prioritizes tasks with high computation or communication costs since they are likely to be the bottlenecks of the application.

HEFT continues to be one of the most popular scheduling algorithms for its simplicity and effectiveness for a wide variety of applications and network configurations. For this reason, we use HEFT as the teacher algorithm for GCNScheduler in this paper.

B. GCNScheduler

GCNScheduler applies recent advancements in machine learning for graphs to the task scheduling problem [11]. Task scheduling presents a unique challenge for graph-based learning because it involves two different graphs - the task graph (with task and dependency costs) and the network (with node compute capabilities and communication rates). The primary innovation of GCNScheduler is the combination of the two graphs into one graph with the same structure as the task graph, but with node and edge features that capture the network configuration. Our previous work shows that that GCNScheduler can produce schedules in a fraction of the time of HEFT [11]. Here we summarize it briefly.

Graph Neural Networks The idea behind GCNs is to interpret graph edges as a *message passing* channel between nodes [16]. In particular, every node is initialized with an embedding (its feature vector). For each GCN layer (the number of layers is a hyper-parameter), a neural network is applied to an aggregation of neighbor embeddings. EDGNN [9] was proposed to capture the nonreciprocal relationship between nodes in directed graphs by treating incoming and outgoing edges differently. The embedding for a node u in the input graph is computed as follows:

$$\mathbf{h}_{n,u}^{(t)} = \sigma\left(\mathbf{W}_1^{(t)}\mathbf{h}_{n,u}^{(t-1)} + \mathbf{W}_2^{(t)}\sum_{\text{neighbor } v}\mathbf{h}_{n,v}^{(t-1)} + \mathbf{W}_3^{(t)}\sum_{v:(v,u)\in E}\mathbf{h}_{e,(v,u)}^{(t-1)} + \mathbf{W}_4^{(t)}\sum_{v:(u,v)\in E}\mathbf{h}_{e,(u,v)}^{(t-1)}\right), \tag{1}$$

where $\mathbf{W}_1^{(t)}$, $\mathbf{W}_2^{(t)}$, $\mathbf{W}_3^{(t)}$, and $\mathbf{W}_4^{(t)}$ represent the weight matrices of layer t for embedding of the node itself, neighboring nodes, incoming edges, and outgoing edges, respectively. Moreover, $\mathbf{h}_{n,v}^{(t)}$ and $\mathbf{h}_{e,(u,v)}^{(t)}$ denote embeddings of node v and of the edge from node u to node v at layer t, respectively. Finally, σ is a non-linear activation function (i.e. ReLU).

Input Graph For a single network $N = (R, C)$ and task graph $G = (V, E)$, GCNScheduler uses the task graph nodes and edges to construct the input graph, i.e. $G_{input} := (V, E)$. Since the makespan (along with other objective metrics) is a function of the required computational time of tasks across machines and the required communication times to transfer task outputs to successor tasks across pairs of machines, GCNScheduler incorporates this information into node and edge features. The node feature vector for a task u is defined as:

$$\mathbf{x}_{\mathbf{n},u} := \left(\frac{\texttt{cost}(u)}{\texttt{comp}(1)}, \frac{\texttt{cost}(u)}{\texttt{comp}(2)}, \cdots, \frac{\texttt{cost}(u)}{\texttt{comp}(|R|)}\right)^T$$

and the edge feature vector for an edge $(u, v) \in E$ is defined as:

$$\mathbf{x}_{\mathbf{e},(u,v)} := \left(\frac{\texttt{data}(u,v)}{\texttt{comm}(1,1)}, \frac{\texttt{data}(u,v)}{\texttt{comm}(1,2)}, \cdots, \frac{\texttt{data}(u,v)}{\texttt{comm}(|R|,|R|)}\right)^T$$

The intuition behind $\mathbf{x}_{\mathbf{n},u}$ is that these features represent the required computational time of task u for each compute node. Similarly, the intuition behind $\mathbf{x}_{\mathbf{e},(u,v)}$ is that these features represent the required time for transferring the result of executing task u to the successor task v for each pair of compute nodes.

Recall that our goal is to train GCNScheduler so that it is generally effective for a given class of networks and/or task graphs. Thus, our training data (and therefore input graph) must include labeled data for many task graph/network pairs. Let $G_{input}(N, G)$ represent the input graph for a single network N and task graph G as described above. Then for a given set of networks $\mathcal{N}$ and set of task graphs $\mathcal{G}$, the input graph for GCNScheduler during training is $G_{input} := \bigcup_{(N,G)\in\mathcal{N}\times\mathcal{G}} G_{input}(N, G)$. Essentially, the input graph is a "forest" of task graphs with network configurations encoded into node and edge feature vectors.

Imitation Learning For the training data, GCNScheduler requires that each node in G_{input} be labeled using some predetermined "teacher" algorithm. For this paper, we use HEFT as the teacher algorithm. So, for every task graph $G = (V, E)$ and network $N = (R, C)$ pair in the training data, we label each node $u \in V$ with the node $r \in R$ which HEFT assigns the task to execute on.

V. Simulation

In this section we provide implementation details of our simulation environment and experimental results. More information and the source code is available online [1].

A. Implementation Details

Task graph The task graph we use features a common pattern for distributed applications modeled as task graphs with four parallel chains of tasks that can be executed in parallel (Fig. 2). We assume the data cost of each edge is 1 for all dependencies. In order to demonstrate that GCNScheduler generalizes for a class of task graphs, we assume tasks costs are drawn from a truncated normal distribution with mean 1, standard-deviation $1/2$, and lower/upper bounds of 0 and 2 respectively. For our dataset, we generated 50 task graphs (30 for training, 10 for validation, and 10 for testing).

Simulated IoT Network For the purpose of comparison between GCNScheduler and HEFT makespans over time, we

[1] https://github.com/ANRGUSC/gsdn

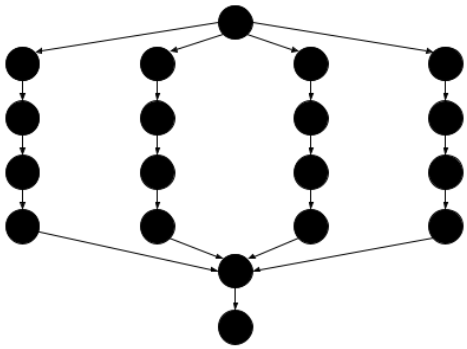

Fig. 2. Task graph used for our experiments with a single source nodes and four parallel four-node chains of tasks. Dependency data sizes are assumed to be 1 while task cost is drawn from a truncated normal distribution between 0 and 2. Its structure is inspired by the epigenomics scientific workflow from [10], it captures the essence of many broader applications where the processing involves multiple parallel chains of tasks.

consider a relatively small network of 10 robots equally spaced along the perimeter of a randomly generated polygon. We generate polygons by first drawing the number of vertices k from $\{3, 4, \ldots, 10\}$ uniformly at random and then using the well-known 2-opt algorithm [21] to generate a k-vertex polygon in $O(k^3)$ time. Each robot patrols counter-clockwise along the perimeter of the polygon at the same speed. For consistency across experiments, we scale polygons so that it takes exactly 500 units of time for a robot to make one trip around the polygon. Each robot is assumed to have equal compute speed of 1. Two robots can communicate directly whenever they are within $s := 50$ units of distance from each other (this number was chosen so that adjacent robots are always connected at the maximum distance). Two robots that are closer than this distance (due to the shape of the polygon) have a communication rate of

$$\frac{1-b}{1+e^{c(2d/s-1)}} + b$$

where d is the distance between the robots, $b := 0.2$ (so that the communication rate between two directly connected nodes is between 0.2 and 1), and $c := 5$ (a constant that affects the shape of the inverse sigmoid). Observe that, by construction, the network is always connected but not necessarily fully connected. Since HEFT requires all pairwise communication rates, we augment the networks to consider multi-hop communication rate between nodes that are further than distance s apart. Let $r_1, r_2 \in R$ be two robots further than s away from each other. Then their communication rate is given by

$$\frac{1}{\sum_{i=1}^{l} 1/\texttt{comm}(p_i, p_{i-1})}$$

where $p = (p_0, p_1, \ldots, p_l)$ is a shortest path (with respect to inverse communication rate) between r_1 and r_2 such that $p_0 = r_0$ and $p_l = r_1$. This results in highly dynamic communication rates, as shown in Fig. 3.

Training For our dataset, we generated 10 random polygons and, for each of these, construct the 10 corresponding networks after robots patrol for $0, 50, 100, \ldots, 450$ units of time. This results in 100 networks in our dataset (60 for training, 20 for validation, and 20 for testing) and a total of 2200 task graph/network pairs (1800 for training, 200 for validation, and 200 for testing). Since each task graph has 19 tasks, the resulting input graph has 41800 nodes (34200 for training, 3800 for validation, and 3800 for testing). We trained GCNScheduler with two 64-unit hidden layers, with 0 dropout using the ReLU activation function, a learning rate of 0.001, a weight-decay of 0.005 in batches of 128 nodes for 200 epochs and achieved a testing accuracy of 56%.

B. Simulation Results

After training GCNScheduler, we generated new polygons and task graphs (drawn from the same distributions as described above) and simulated the robots patrolling for 1500 units of time so that each robot traverses the entire perimeter of the random polygon three times. For each task graph/network pair, we run four simulations in parallel to compare the following scheduling algorithms:

1) HEFT: The standard HEFT algorithm
2) GCNScheduler: The trained network used in prediction mode
3) Random: Each task is assigned to execute on a random node
4) Static: HEFT runs once on the initial network and the schedule is not updated

The random and static schedulers are of interest because as the task graph and/or network size is increased, recomputing a schedule becomes intractable. Each algorithm recomputes a schedule when the last execution has completed, as described in Algorithm 1.

Algorithm 1 Simulation with stop time `stop` and timestep dt

```
t ← 0
sched ← ∅
polygon ← random polygon
G ← random task graph
while t < stop do
    N ← network induced by robot positions at time t
    if sched = ∅ or finished executing then
        sched ← SCHEDULER(G, N)
    end if
    execute G according to sched on N for dt time
    t ← t + dt
end while
```

We ran 50 simulations and collected information on robot communication rates over time, the number of full application executions in each simulation, and the makespans of each of these executions. While the average communication rate over all robots is relatively stable over time (Fig. 3), the individual communication rate for a single robot is highly volatile over the course of its patrol around the polygon (Fig. 4). Therefore, we expect that a scheduler which can adapt to changing network conditions (HEFT and GCNScheduler) will be relatively

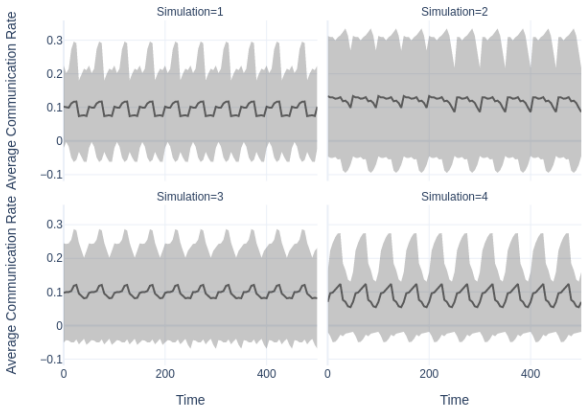

Fig. 3. Average communication rate (with one-standard-deviation error band) of all robots over the time it takes to make one trip around the perimeter of the polygon.

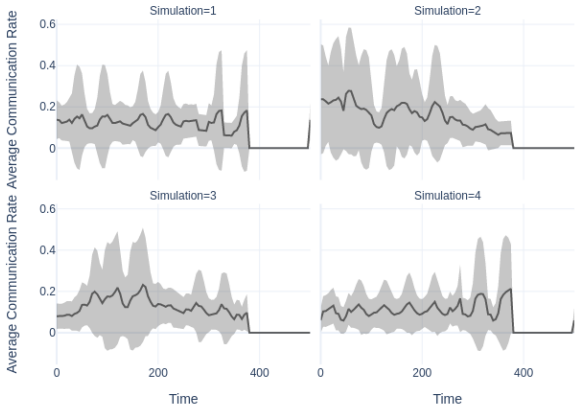

Fig. 4. Average communication rate (with one-standard-deviation error band) of one robot as it makes one trip around the perimeter of the polygon.

stable while a scheduler that does not (Static and Random schedules) will experience large spikes in makespans as nodes communication rates degrade.

Fig. 5 shows the average makespan of each scheduler over the average makespan of the HEFT scheduler, or *makespan ratio*. This metric is an indicator of how much better or worse than HEFT a scheduler is at minimizing makespan (< 1 indicates better than HEFT; > 1 indicates worse). Over all 50 simulations, GCNScheduler has the lowest average makespan ratio and smallest variance. Since schedules are produced back-to-back, the makespan has a direct effect on the number of executions (or the number of schedules produced) for each scheduler. Fig. 6 shows that while HEFT is able to complete the most executions in one trip around the polygon, GCNScheduler outperforms the static and random schedulers and achieves the smallest variance. The key takeaway is that GCNScheduler is able to adapt to the high volatility of the

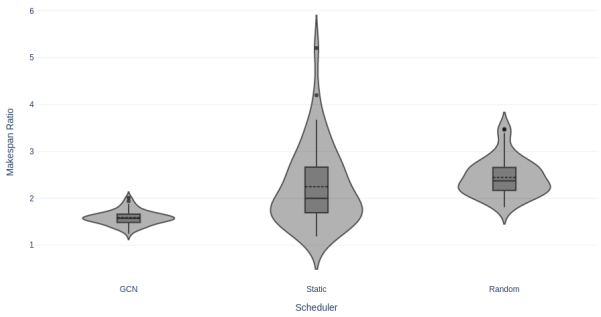

Fig. 5. Violin plot for 50 simulations of makespan ratios for each scheduler. For each violin, the dashed line marks the mean, the solid line marks the median, and the top and bottom of the boxes mark the first and third quartiles, respectively. GCNScheduler has the lowest average makespan ratio of 1.58 with a standard deviation of 0.154, the static scheduler an average makespan ratio of 2.246 with a standard deviation of 0.831, and the random scheduler an average makespan ratio of 2.442 with a standard deviation of 0.396.

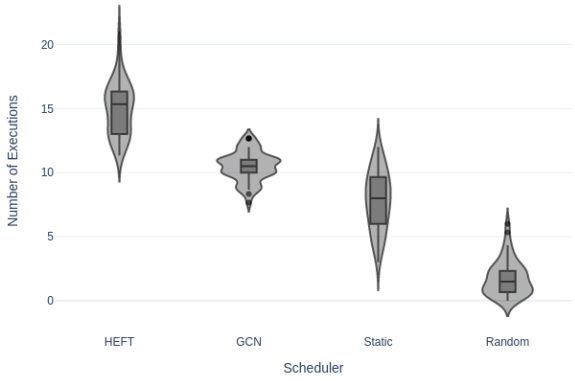

Fig. 6. Violin plot for 50 simulations of the number of task graph executions for each scheduler. HEFT supports 15 executions on average with a standard deviation of 2.087, GCNScheduler supports 10.48 executions on average with a standard deviation of 1.163, the static scheduler supports 7.82 executions on average with a standard deviation of 2.247, and the random scheduler supports 1.78 executions on average with a standard deviation of 1.379.

network conditions because of its fast computation times.

VI. Conclusions and Future Work

The results from our experiments reveal many interesting directions for future research. First, one issue with GCNScheduler is that it cannot learn the ordering that HEFT prescribes when scheduling since it only labels tasks with the compute nodes they should execute on. This may be part of the reason why the accuracy is relatively low. It may be interesting to explore whether different loss functions (we use cross-entropy loss) or neighborhood embedding aggregation functions (we use averaging) can account for this.

In order to make the problem interesting, we chose a task graph for which HEFT is able to produce "interesting" schedules (i.e. not simply scheduling everything on a single node). In order to create better models, though, we need to better understand the performance of HEFT (or any other scheduler) on different classes of task graphs and networks. It seems reasonable, for example, that one scheduler works very well on one class of networks while another scheduler works well on a different class. GCNScheduler could even be trained using a hybrid teacher algorithm which takes the best of multiple traditional schedules.

We have experimented with GCNScheduler on a network of mobile robots (Fig. 1) and preliminary findings indicate that it adapts when a single robot becomes disconnected from the network [5]. These findings were limited to four network nodes, while the simulation results presented here indicate that GCNScheduler shows more impressive performance on larger networks. Experimenting with larger, real-world robot networks is another direction for future work.

In this paper, we have demonstrated that a Graph Convolutional Netowrk-based Scheduler can learn to produce schedules with decent makespan and can compute schedules fast enough to adapt to dynamic networks, making it applicable for the Internet of Robotic Things.

VII. Acknowledgements

This work was supported in part by Army Research Laboratory under Cooperative Agreement W911NF-17-2-0196.

References

[1] S. K. Addya, A. K. Turuk, B. Sahoo, M. Sarkar, and S. K. Biswash. Simulated annealing based vm placement strategy to maximize the profit for cloud service providers. *Engineering science and technology, an international journal*, 20(4):1249–1259, 2017.

[2] M. R. Alizadeh, V. Khajehvand, A. M. Rahmani, and E. Akbari. Task scheduling approaches in fog computing: A systematic review. *International Journal of Communication Systems*, 33(16):e4583, 2020.

[3] Y. Azar and A. Epstein. Convex programming for scheduling unrelated parallel machines. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 331–337, 2005.

[4] R. F. de Mello, J. A. A. Filho, L. J. Senger, and L. T. Yang. Grid job scheduling using route with genetic algorithm support. *Telecommunication Systems*, 38:147–160, 2008.

[5] D. D'Souza, M. Kiamari, L. Clark, J. Coleman, and B. Krishnamachari. Graph convolutional network-based scheduler for distributing computation in the internet of robotic things. The 2nd Student Design Competition on Networked Computing on the Edge. url=https://github.com/ANRGUSC/gcnschedule-turtlenet, 2022.

[6] M. Z. Hasan and H. Al-Rizzo. Task scheduling in internet of things cloud environment using a robust particle swarm optimization. *Concurrency and Computation: Practice and Experience*, 32(2):e5442, 2020.

[7] M. A. Iverson and F. Özgüner. Hierarchical, competitive scheduling of multiple dags in a dynamic heterogeneous environment. *Distributed Syst. Eng.*, 6:112–, 1999.

[8] H. Izadkhah. Learning based genetic algorithm for task graph scheduling. *Applied Computational Intelligence and Soft Computing*, 2019, 2019.

[9] G. Jaume, A.-p. Nguyen, M. R. Martínez, J.-P. Thiran, and M. Gabrani. edgnn: a simple and powerful gnn for directed labeled graphs. *arXiv preprint arXiv:1904.08745*, 2019.

[10] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. Characterizing and profiling scientific workflows. *Future generation computer systems*, 29(3):682–692, 2013.

[11] M. Kiamari and B. Krishnamachari. GCNScheduler: Scheduling distributed computing applications using graph convolutional networks. *arXiv preprint arXiv:2110.11552*, 2021.

[12] M. Kumar and S. C. Sharma. Dynamic load balancing algorithm for balancing the workload among virtual machine in cloud computing. *Procedia computer science*, 115:322–329, 2017.

[13] S. Liu, S. Yao, X. Fu, H. Shao, R. Tabish, S. Yu, A. Bansal, H. Yun, L. Sha, and T. Abdelzaher. Real-time task scheduling for machine perception in in intelligent cyber-physical systems. *IEEE Transactions on Computers*, 2021.

[14] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*, pages 270–288. 2019.

[15] S. Russell and T. Abdelzaher. The internet of battlefield things: the next generation of command, control, communications and intelligence (c3i) decision-making. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*, pages 737–742. IEEE, 2018.

[16] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[17] H. Y. Shishido, J. C. Estrella, C. F. M. Toledo, and M. S. Arantes. Genetic-based algorithms applied to a workflow scheduling algorithm with security and deadline constraints in clouds. *Computers & Electrical Engineering*, 69:378–394, 2018.

[18] M. Skutella. Convex quadratic and semidefinite programming relaxations in scheduling. *Journal of the ACM (JACM)*, 48(2):206–242, 2001.

[19] R. Sudarsan and C. J. Ribbens. Combining performance and priority for scheduling resizable parallel applications. *Journal of Parallel and Distributed Computing*, 87:55–66, 2016.

[20] P. Sun, Z. Guo, J. Wang, J. Li, J. Lan, and Y. Hu. Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 3314–3320, 2021.

[21] A. Thomas and H. Martin. Heuristics for the generation of random polygons. In *Proceedings of the 8th Canadian Conference on Computational Geometry (CCCG'96)*, pages 38–44. Citeseer, 1996.

[22] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[23] J. D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[24] O. Votava, P. Macejko, and J. Janecek. Dynamic local scheduling of multiple dags in distributed heterogeneous systems. In *DATESO*, 2015.