DRAS: Deep Reinforcement Learning for Cluster Scheduling in High Performance Computing

Yuping Fan, Boyang Li, Dustin Favorite, Naunidh Singh, Taylor Childers, Paul Rich, William Allcock, Michael E. Papka, and Zhiling Lan

Abstract—Cluster schedulers are crucial in high-performance computing (HPC). They determine when and which user jobs should be allocated to available system resources. Existing cluster scheduling heuristics are developed by human experts based on their experience with specific HPC systems and workloads. However, the increasing complexity of computing systems and the highly dynamic nature of application workloads have placed tremendous burden on manually designed and tuned scheduling heuristics. More aggressive optimization and automation are needed for cluster scheduling in HPC. In this work, we present an automated HPC scheduling agent named DRAS (Deep Reinforcement Agent for Scheduling) by leveraging deep reinforcement learning. DRAS is built on a hierarchical neural network incorporating special HPC scheduling features such as resource reservation and backfilling. An efficient training strategy is presented to enable DRAS to rapidly learn the target environment. Once being provided a specific scheduling objective given by the system manager, DRAS automatically learns to improve its policy through interaction with the scheduling environment and dynamically adjusts its policy as workload changes. We implement DRAS into a HPC scheduling platform called CQGym. CQGym provides a common platform allowing users to flexibly evaluate DRAS and other scheduling methods such as heuristic and optimization methods. The experiments using CQGym with different production workloads demonstrate that DRAS outperforms the existing heuristic and optimization approaches by up to 50%.

Index Terms—High-performance computing, cluster scheduling, deep reinforcement learning, job starvation, backfilling, resource reservation, OpenAI Gym

1 Introduction

Cluster schedulers play a critical role in high-performance computing (HPC). They enforce site policies through deciding when and which user jobs are allocated to system resources. Common scheduling goals include high system utilization, good user satisfaction and job prioritization. *Heuristics* are the prevailing approaches in HPC cluster scheduling. For example, first come, first served (FCFS) with EASY backfilling is a well-known scheduling policy deployed on production HPC systems [1]. Bin packing is another well-known heuristic approach aiming for high utilization. Heuristics are easy to implement and fast by trading optimality for speed. In addition, *optimization* is also extensively studied in the literature for cluster scheduling [2], [3], [4]. Optimization methods focus on optimizing immediate scheduling objective(s) without regard to long-term performance. Moreover, both heuristics and optimization approaches are static, and neither of them is capable of adapting its scheduling policy to dynamic changes in the environment. In case of sudden variation in workloads, system administrators have to manually tune the algorithms and parameters in methods to mitigate performance degradation. As HPC systems become increasingly complex combined with highly diverse application workloads, such a manual process becomes challenging, time-consuming, and error-prone. We believe that more aggressive optimization and automation, beyond the existing heuristics and optimization methods, is essential for HPC cluster scheduling.

In recent years, reinforcement learning (RL) combined with deep neural networks has been successfully employed in various fields for dynamic decision making, such as self-driving cars [8], autonomous robots [9], and game playing [10], [11]. RL refers to an area of machine learning that automatically learns to maximize cumulative reward through interaction with the environment [12]. Deep Q-learning (DQL) and policy gradient (PG) are two widely adopted RL baselines. More advanced RL algorithms, e.g., A2C (Advantage Actor Critic) [13] and PPO (Proximal Policy Optimization) [14], have emerged in recent years to address some key challenges, such as learning efficiency and performance robustness [15], in real applications.

- *Yuping Fan, Boyang Li, Dustin Favorite, Naunidh Singh, and Zhiling Lan are with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616 USA. E-mail: {yfan22, bli70, dfavorite, nsingh28}@hawk.iit.edu, lan@iit.edu.*
- *Taylor Childers, Paul Rich, and William Allcock are with Argonne National Laboratory, Lemont, IL 60439 USA. E-mail: {jchilders, richp, allcock}@anl.gov.*
- *Michael E. Papka is with Argonne National Laboratory, Lemont, IL 60439 USA, and also with the University of Illinois, Chicago, IL 60607 USA. E-mail: papka@anl.gov.*

Manuscript received 8 November 2021; revised 24 July 2022; accepted 29 August 2022. Date of publication 16 September 2022; date of current version 3 October 2022.
This work was supported in part by US National Science Foundation under Grants CNS-1717763, CCF-2109316 in part by the and CCF-2119294. Argonne Leadership Computing Facility is a U.S. Department of Energy Office of Science User Facility operated under Grant DE-AC02-06CH11357, and in part by the National Energy Research Scientific Computing Center (NERSC) is a U.S. Department of Energy Office of Science User Facility operated under under Grant DE-AC02-05CH11231.
(Corresponding author: Yuping Fan.)
Recommended for acceptance by A. Bhatele.
Digital Object Identifier no. 10.1109/TPDS.2022.3205325

Several recent studies have shown that RL driven scheduling is a promising approach for cluster scheduling [6], [16], [17], [18], [19], [20]. Unfortunately, these studies do not take into account two special features of cluster scheduling in HPC, that is, *resource reservation* to prevent job starvation and *backfilling* to reduce resource fragmentation. In order to effectively use RL algorithms for HPC scheduling, specialized strategies are needed.

In this study, we present an automated HPC scheduling agent named *DRAS (Deep Reinforcement Agent for Scheduling)* tailored for HPC workloads [7]. The goal of the agent is twofold: (1) to improve HPC scheduling performance beyond the existing approaches, and (2) to automatically adjust scheduling policies in case of workload changes. In the design of DRAS, we incorporate the HPC domain specific knowledge into the formulation of deep reinforcement learning and introduce *a hierarchical neural network structure*, where the level-1 network selects jobs for immediate or reserved execution and the level-2 network concentrates on choosing proper backfilled jobs for more scheduling optimization. The customized hierarchical structure is applicable to any generalized RL algorithms with minimal changes. In order to optimize and automate the process, all the scheduling decisions including immediate job selection, job reservation, and backfilling are made by DRAS without human involvement. Moreover, we develop a three-phase training process using historical job logs. Our training strategy allows DRAS to gradually explore simple average situations to more challenging rare situations, hence rapidly converging to high-quality models.

To bridge RL and HPC scheduling, we build an open-source platform named CQGym [21]. CQGym provides a common platform allowing users to flexibly evaluate DRAS and other scheduling methods such as heuristic and optimization methods. GQGym consists of three main components: scheduling environment, Gym interface [22], and scheduling agent. The scheduling environment is an event-driven HPC job scheduling simulator that executes scheduling decisions made by the scheduling agents. It can adopt any event-driven HPC scheduling simulator and our default implementation is CQSim [23], which has been used in HPC scheduling research for decades [2], [24], [25]. Gym provides a standard interface between the scheduling environment and the scheduling agent. Both RL algorithms and heuristic policies can be easily implemented as a scheduling agent and communicate with the environment through Gym. By decoupling scheduling agents from the environment, we can focus on implementing scheduling policies without considering the complexity of the environment. In addition, the standard decision-making process ensures a fair comparison between different scheduling policies.

In this study, we implement four DRAS agents by customizing four popular RL algorithms which are denoted as DRAS-PG, DRAS-DQL, DRAS-A2C, DRAS-PPO. These DRAS agents are compared with the existing scheduling methods in CQGym by using the job traces collected from two production supercomputers representing capability computing and capacity computing. The results indicate our DRAS agents outperform the existing scheduling methods using heuristic or optimization methods. Each DRAS agent is capable of automatically learning to improve its policy through interaction with the scheduling environment and dynamically adjusts its policy as workload changes. In the comparison between DRAS agents, PPO delivers the best overall performance. Specifically, this paper makes four major contributions:

1) We design four new DRAS agents, i.e., DRAS-DQL, DRAS-PG, DRAS-A2C, and DRAS-PPO, which leverage the advance in deep reinforcement learning and incorporate the key features of HPC scheduling in the form of a hierarchical neural network model.
2) We develop a three-phase training process that allows DRAS to automatically learn the scheduling environment (i.e., the system and its workloads) and to rapidly converge to an optimal policy.
3) We develop a common and extensible HPC job scheduling platform CQGym to bridge the HPC scheduling environment and various scheduling agents. It facilitates quantitative evaluation of various scheduling methods including RL-based agents such as DRAS, heuristics, and optimization methods.
4) We evaluate four DRAS agents and five traditional scheduling policies by using CQGym. Our trace-based simulation demonstrates DRAS outperforms existing scheduling methods by up to 50%. Compared to the heuristic and optimization approaches, DRAS offers two benefits: better long-term scheduling performance and adaptation to dynamic workload changes without human intervention.

2 Background and Challenges

2.1 Cluster Scheduling in HPC

HPC cluster scheduling, also known as batch scheduling, is responsible for assigning jobs to resources (e.g, compute nodes) according to site policies and resource availability [1], [26], [27]. Well-known schedulers include Slurm, Moab/TORQUE, PBS, and Cobalt [28], [29], [30], [31]. Let's consider a cluster with N nodes. Users submit their jobs to the system through the scheduler. When submitting a job, a user is required to provide job size (i.e., number of compute nodes needed for a job) and job runtime estimate (i.e., estimated time needed for a job). Typical HPC jobs are *rigid*, meaning job size is fixed throughout its execution. Job runtime estimate is the upper bound for the job such that it will be killed by the scheduler if the actual job runtime exceeds this runtime estimate [32]. At each scheduling instance, the scheduler orders jobs in the queue according to the site policy and executes jobs from the head of the queue.

Existing HPC scheduling policies can be broadly classified into two groups: *heuristics* and *optimization* methods. First Come First Serve (FCFS) with EASY backfilling is the most widely used heuristics, which sorts jobs in the wait queue according to their arrival times and executes jobs from the head of the queue [1]. If the available resources are not sufficient for the first job in the queue, the scheduler will reserve the resources for this job. *Backfilling* is often used in conjunction with reservation to enhance system utilization. It allows subsequent jobs in the wait queue to move ahead under the condition that they do not delay the existing reservations [1]. Optimization methods select a set

of jobs from the queue with an objective to optimize certain scheduling metrics, such as minimizing average job wait time and maximizing system utilization.

Several recent studies have explored reinforcement learning for cluster scheduling. DeepRM [16] is the first study demonstrating the potential of using reinforcement learning for learning customized scheduling policies from experience. It uses the policy gradient method to train the RL agent to minimize average job complete time. Its neural network input is represented as the state of system and jobs in two dimensions, i.e., resource and time. DeepRM's state representation has been adopted by other studies [18], [20], [33]. A major limitation of DeepRM's state representation is the use of a bounded time horizon for scheduling environments and job runtimes, whereas real HPC cluster scheduling problems have infinite time horizons and job runtimes vary from seconds to several days. Therefore, DeepRM is infeasible to schedule realistic HPC cluster workloads with continuous job arrivals. Unlike DeepRM, RLScheduler [17] attempts to develop a general reinforcement learning model that is trained with one system log and then is used on other systems with different characteristics (e.g., system size, workload patterns, etc.). While such a generic model is appealing, Table V-VII in RLScheduler paper [17] shows that it might lead to less satisfactory scheduling performance than heuristic methods, e.g., FCFS and Short Job First (SJF). In Section 6.6, we compare the performance of RLScheduler-learned models with our customized model, which shows that our customized models deliver better performance.

The work most closely related to ours is Decima, which explores reinforcement learning to allocate data processing jobs [6]. Each job consists of dependent tasks and is represented as directed acyclic graphs (DAGs). Decima integrates a graph neural network to extract job DAGs and cluster status as embedding vectors. It then feeds the embedding vectors to a policy gradient network for decision making. The decision consists of two parts: to select tasks for immediate execution and to determine task parallelism. However, HPC scheduling targets different goals than DAG job scheduling. First, DAG jobs can be decomposed into malleable tasks, whereas HPC is dominated by rigid jobs that cannot be decomposed. Second, HPC scheduling needs resource reservation support to prevent large jobs from starvation, while data processing jobs are decomposable and therefore are not affected by job starvation problems. Hence, Decima does not equip with resource reservation strategies. In short, Table 1 summarized and compared existing cluster scheduling methods, along with their features.

2.2 Overview of Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning technique that studies how agents situated in stochastic environments can learn optimal policies through interaction with their environment [34]. The agent's environment is described by an abstraction called Markov Decision Process (MDP) with four basic components: state space S, action space A, reward R, and state transition probability P. In Markov decision processes, a learning agent interacts with a dynamic environment in discrete timesteps. At each time step t, the agent observes the state $s_t \in S$ and takes an action $a_t \in A(s_t)$. Upon taking the action, the environment transits to a new state s_{t+1} with the transition probability $P(s_{t+1}|s_t, a_t)$ and provides a reward r_t to the agent as feedback of the action. The process continues until the agent reaches a terminal state. The goal of the agent is to find a policy $\pi(s)$, mapping a state to an action (deterministic) or a probability distribution over actions (stochastic), which maximizes *the long-term (discounted) cumulative reward* $R(t) = \sum_{t'=t}^{T} \gamma^{t'} r_{t'}$. A discount factor γ is between 0 and 1. The smaller of γ, the less importance of future rewards.

In practice, the state and action space is often too large to be stored in a lookup table. It is common to use function approximators with a manageable number of adjustable parameters θ, to represent the components of agents. Using a deep neural network with reinforcement learning is often called *deep reinforcement learning* [35]. The highly representational power of deep neural networks enables reinforcement learning to solve complex decision-making problems, such as playing Atari and Go games [10], [11].

Two main approaches to represent agents with model-free reinforcement learning are Q-learning and policy gradient. In Q-learning algorithms, an agent chooses an action at a given state that maximizes Q-value, i.e., the cumulative reward over all successive steps. Q-table is a lookup table containing Q-value for all the state-action pairs. To address an overwhelming number of state-action pairs, neural networks are often used to approximate Q-table and the methods are generally called deep Q-learning (DQL). DQL learns by approximating the optimal action-value function $Q^*_\theta(s, a)$. Policy gradient methods directly parameterize the policy $\pi_\theta(s)$ and optimize the parameters θ in the neural network by gradient descent. Policy gradient methods are generally believed to be applicable for a wider range of problems and converge faster, but tend to converge to a local optimum. On the other hand, Q-learning methods are more difficult to converge, but once they converge, they tend to have more stable performance than policy gradient [36].

Actor-critic approaches combine the strengths of Q-learning and policy gradient via a hybrid neural network consisting of two parts: actor and critic. The actor network is the policy distribution $\pi_\theta(s)$ to be optimized; the critic network is value function $V(s)$ to be learned. The estimated value function provides a baseline for policy updates, and thus reduces variance estimates in actor network. Advantage Actor-Critic (A2C) and PPO (Proximal Policy Optimization) are the most popular actor-critic variants. Advantage Actor-Critic models define advantage term, i.e., $A(s) = R_t - V(s)$. Subtracting the learned value function reduces the reward variance and keeps advantages unbiased. A2C [37] and A3C [13] are two variants of advantage actor critic models. A3C is the asynchronous version allowing multiple workers training asynchronous, while A2C is the synchronous version training on a single worker. Research found that A2C yields comparable performance to A3C while being more cost-effective [38]. PPO is the more advanced actor-critic model published by OpenAI in 2017 [14], which leverages clipping or penalty surrogate to prevent large policy changes in a single step. Other models, such as Trust region policy optimization (TRPO), also have constraint on policy updates in a single step. Studies show that PPO is simpler with fewer hyperparameters, while achieving comparable performance

TABLE 1
Comparison of Cluster Scheduling Methods

Features \ Methods	FCFS [1]	BinPacking [5]	Optimization [2], [3], [4]	Decima [6]	DRAS [7]
Adaption to workload changes	✗	✗	✗	✓	✓
Automatic policy tuning	✗	✗	✗	✓	✓
Long-term scheduling performance	✗	✗	✗	✓	✓
Starvation avoidance	✓	✗	✗	✗	✓
Require training	✗	✗	✗	✓	✓
Implementation effort	Easy	Easy	Median	Hard	Hard
Key objective	Fairness	Resource utilization	Customizable	Customizable	Customizable

[14]. Given PPO's superior performance and simple implementation, it has become a commonly used reinforcement learning baseline.

2.3 Technical Challenges

Designing deep reinforcement learning driven scheduling for HPC is challenging. Key obstacles as listed below.

Avoiding Job Starvation. HPC jobs have drastically different characteristics: user jobs may range from a single-node job to a whole-system job, and job runtimes may vary from seconds to hours or even days. This feature presents a unique challenge to HPC systems: jobs, especially large-sized jobs, tend to be starved, if small-sized jobs keep arriving and skip over large jobs due to insufficient available resources. Directly applying existing RL-based scheduling methods can lead to severe job starvation. We have tested a state-of-the-art policy gradient method with a real workload trace. Our results show that large jobs, e.g., 4k-node jobs, were held in the queue for 170 days. Typically, large jobs have high priority at HPC sites, especially capability computing facilities. The long wait times discourage users from submitting large jobs.

Incorporating Backfilling. Backfilling is a key strategy to reduce resource fragmentation in HPC. Currently, the well-known EASY backfilling strategy uses the simple first-fit method to select jobs for backfilling, i.e., choosing the first job which can fit in the backfill hole. We suggest that similar to the selection of jobs for scheduling, the selection of jobs for backfilling has many possible options, hence having the potential for more aggressive optimization.

Scalable State and Action Representation. To transform a scheduling problem to a reinforcement learning problem, we must first capture the dynamic environment, e.g., status of thousands of nodes and hundreds of waiting jobs, to a state vector as an input to the neural network. Additionally, it is vitally important to map the extremely large action space to an output of the neural network in a manageable size. The action space grows exponentially with the number of jobs in the queue. Working directly with large action space can be computationally demanding.

Effective Agent Training. A RL agent learns to improve its policy by experiencing diverse situations. Effective training should be capable of efficiently and rapidly building a converged model based on sample data in order to make decisions without being explicitly programmed to do so. It is also challenging to select training data to reliably cover as much of the state space as possible and generalize to new or unseen situations.

Bridging RL and HPC Scheduling. The vigorous debates on whether to choose classical heuristic policies or automatically learned policies call for a fair comparison between them. However, scheduling processes vary significantly between traditional and auto-learn policies. Even different reinforcement learning policies can vary significantly on how to learn and make decisions. Plus, novel reinforcement learning algorithms are rapidly emerging. To fairly compare traditional and RL-based policies, it is vitally important to build a common platform for scheduling policies to easily communicate with the scheduling environment.

3 DRAS: HPC Scheduling Agent

To address the aforementioned challenges, we present DRAS, our scheduling engine tailored for HPC workload and empowered by deep reinforcement learning. DRAS, illustrated in Fig. 1, represents the scheduler as an agent to make decisions on *when and which* jobs should be allocated to computer nodes with the objective to optimize scheduling performance. The environment interface constantly sends scheduling requests to the DRAS agent. Upon receiving a request at time t, DRAS first encodes the job queue and system state into a vector s_t, and passes the vector to the neural network (Section 3.1). Next, DRAS uses a hierarchical neural network for decision making (Section 3.2). The agent takes an action by selecting jobs from the wait queue according to the output of the neural network and then receives a reward signal from the environment. The goal of DRAS is to choose actions (i.e., to select jobs) over time so as to maximize the cumulative reward. DRAS trains its neural network through simulation with massive datasets composed of real, sampled, and synthetic workload traces (Section 3.3). Once the model is converged, we deploy the DRAS agents into operation. The DRAS agents automatically adjust their neural network parameters during operation to handle workload changes.

3.1 State, Reward and Action Representation

The DRAS agent receives three observations from the environment: (1) job wait queue, (2) cluster node status, and (3) reward, a scalar indicating the quality of the action.

State. We encode each waiting job as a vector of [2,2], containing four pieces of information, including job size, job estimated runtime, priority (1 means high priority; 0 means low priority), and job queued time. We encode each node as a vector of [1,2] with two pieces of information. The first cell is a binary representing node availability (1 means available; 0 means not available). If the node is occupied, we use

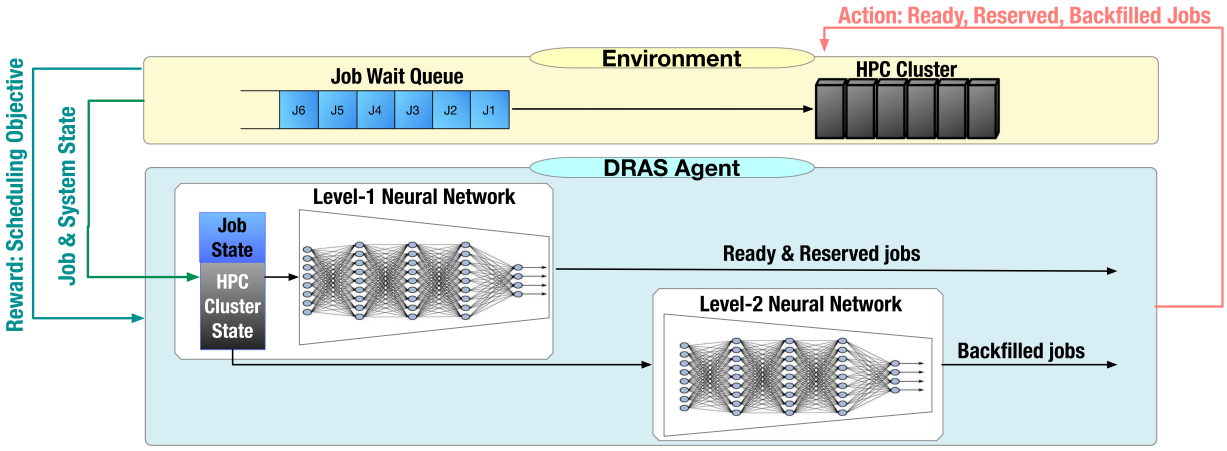


Fig. 1. DRAS overview. DRAS agent (at the bottom) represents the scheduling agent; the environment (at the top) comprises the rest of the system, including job wait queue and HPC cluster. The DRAS agent first observes the environment state, including job state and system state, and encodes the state into a vector. The agent's neural network takes the vector as input and outputs a scheduling action. The environment executes the action and provides a reward indicating the quality of the action. The agent uses reward to improve its policy automatically.

the user-supplied runtime estimate and job start time to calculate the node estimated available time. The second cell represents the time difference between the node estimated available time and the current time. If the node is available, we set the second cell to zero. We concatenate job information and node information into a fixed-size vector as the input to the network.

Reward. Reward functions reflect scheduling objectives. It is hard to offer a one-size-fits-all reward function due to diverse site objectives. HPC systems can be broadly classified as capability computing or capacity computing. Capability computing facilities are commonly interested in prioritizing capability jobs (i.e., large jobs) [26] and optimizing resource utilization. An example reward of capability computing could be as follows:

$$w_1 \times \frac{\overline{t}}{t_{max}} + w_2 \times \frac{\overline{n}}{N} + w_3 \times \frac{N_{used}}{N} \tag{1}$$

where $\overline{t}$ denotes the average wait time of selected jobs; t_{max} is the maximum wait time of jobs in the queue. Similarly, $\overline{n}$ is the average job size of the selected jobs; N is the total number of nodes in the system; N_{used} is the number of occupied nodes. In other words, this reward function intends to balance three factors: to prevent job starvation, to promote capability jobs, and to improve system utilization. The weights can be tuned by system administrators based on the site priority. For example, the higher w_1 value could meet a more stringent requirement on job starvation.

Capacity computing facilities typically focus on fast turnaround time and short wait time [39]. For capacity computing facilities, we may define the reward function as:

$$\frac{\sum_{j \in J} -1/t_j}{c} \tag{2}$$

where J is the set of jobs in the queue and c is the number of waiting jobs at the current timestep. This reward function aims to minimize the average job wait time.

Action. DRAS processes the input vector and outputs a vector as the scheduling action. The output vector specifies which jobs are selected for job execution (i.e., immediate execution, reserved execution, and backfilled execution). Intuitively, at each scheduling instance, the scheduler selects multiple jobs simultaneously. This leads to an explosive number of actions and is infeasible to be trained efficiently. Instead, DRAS decomposes one scheduling decision (i.e., selects several jobs in one shot) into a series of job selections, i.e., selecting one job at each time.

3.2 Two-Level Neural Network

A key challenge when applying reinforcement learning to HPC cluster scheduling is to prevent job starvation. State-of-the-art RL methods focus on scheduling jobs for immediate execution and lack reservation strategy, hence leading to job starvation. To overcome this obstacle, we build *a hierarchical neural network structure*, in which the level-1 network is to select jobs for immediate or reserved execution and the level-2 network is to identify jobs for backfilling.

More specifically, at a given scheduling instance, the scheduler first enforces a window at the front of the job wait queue. The window alleviates job starvation problems by providing higher priorities to older jobs. The level-1 network selects a job from the window. If the number of available nodes is more than or equal to the job size, the agent marks the job as *ready job* and sends it for immediate execution on the system. This process repeats until the job selected from the window has a size greater than the number of available nodes. The agent marks the job as a *reserved job* and reserves a set of nodes for its execution on the system at the earliest available time. At this point, the agent moves to the level-2 network. Unlike the first-fit strategy used in the traditional backfilling method, we use the neural network to make backfilling decisions so as to minimize resource waste. Toward this end, we fill the window with job candidates, i.e., the jobs that can be fit into the holes in the system before the reserved time. The agent selects one job at a time for the system to backfill. The process at the level-2 network repeats until no more job candidates for backfilling.

In a nutshell, the decision making of DRAS is to select jobs and execute them in three modes:

1) *ready job*: the jobs are selected to run immediately.
2) *reserved job*: the jobs are selected to start at the earliest reserved time.
3) *backfilled job*: the jobs are selected to fill the holes before the reserved time.

The same neural network is used for both level-1 and level-2 networks. The entire 2-level neural network is trained jointly using deep reinforcement learning to optimize scheduling performance. Each network consists of *five layers*: input layer, convolution layer, two fully-connected layers, and output layer. The input layer is connected to a convolution layer with a 1×2 filter to extract job or node status information in each row. The convolution layer is connected to two fully-connected layers activated by a leaky rectifier [40]. The second fully-connected layer is connected to the output layer. We denote all of the parameters in the neural network jointly as θ.

The two-level neural network structure is generally applicable to various RL algorithms. In this study, we develop four RL agents leveraging different RL algorithms: *DRAS-DQL*, *DRAS-PG*, *DRAS-A2C* and *DRAS-PPO*. DQL denotes deep Q-learning; PG denotes policy gradient; A2C denotes advantage actor critic, PPO denotes proximal policy optimization. The selection of DQL, PG, A2C and PPO is for us to systematically evaluate these popular reinforcement learning methods under a unified environment.

DRAS-DQL uses the neural network to approximate Q-value as $Q_\theta(s_k, a_k)$ (i.e., the expected cumulative reward of taking action a_k in state s_k). DRAS-DQL network processes one job at a time and produces the expected Q-value for this job. We use the same network to approximate Q-value for all the jobs in the window W. The input of the DRAS-DQL neural network is a 2D vector of $[2 + N, 2]$, containing one job information and N nodes information. The output is a single neuron corresponding to the expected Q-value of the job. After processing all the jobs in the window, normally, the agent selects the job with the highest Q-value.

In order to explore various actions, the agent randomly chooses a job instead of the job with the highest Q-value with probability ϵ. In practice, ϵ is very high at the beginning of the training to ensure that the agent explores various state-action pairs and it decays over time as the agent becomes more experienced. In our study, we set $\epsilon = 1.0$ at the beginning of the training and it decays at the rate of $\alpha = 0.995$. In training, the parameters θ in DRAS-DQL network is updated by:

$$\theta \leftarrow \theta - \alpha \sum_{k=1}^{K} \bigtriangledown_\theta Q(s_k, a_k)(\underbrace{r_k + \max_a Q(s_{k+1}, a)}_{\text{new value}} - \underbrace{Q(s_k, a_k)}_{\text{old value}}) \tag{3}$$

Here, the old value $Q(s_k, a_k)$ is the expected Q-value of taking action a_k at state s_k. After taking action a_k, we can compute the more accurate expected Q-value (i.e., the new value) by adding the immediate reward r_k and the expected cumulative future reward. DRAS-DQL networks learn through minimizing the loss between the new value and the old value.

DRAS-PG uses the neural network to parameterize scheduling policy as $\pi_\theta(s_k, a_k)$ (i.e., the probability of taking action a_k in state s_k). The input of DRAS-PG is a 2D vector of $[2 \times W + N, 2]$, where W is the window size and N is the total number of nodes in the system. The output of the neural network contains W neurons, each denoting the probability of selecting a job out of the W jobs. A scheduling action is stochastically drawn from the W jobs following their probability distributions. We employ the softmax [40] as the activation function to ensure the sum of output values equals to 1.0. If the number of wait jobs is less than the window size W, we mask the invalid actions in the output by rescaling all valid actions. In terms of learning, DRAS-PG method updates the neural network parameters θ by:

$$\theta \leftarrow \theta + \alpha \sum_{k=1}^{K} \bigtriangledown_\theta log\pi_\theta(s_k, a_k) \left(\sum_{k'=k}^{K} r_{k'} - b_k \right) \tag{4}$$

Here, K denotes the total number of actions taken in the parameter update, α is the learning rate, and b_k is the baseline used to reduce the variance of policy gradient. We set b_k to the cumulative reward from step k onwards averaging over all past parameter updates. Gradient descent was performed using the Adam optimizer [41].

DRAS-A2C was introduced to reduce baseline variance in the policy gradient method. In Equation (4), the baseline is the average values of the historical steps. Actor critic model tries to provide a more accurate baseline and thus further reduce variance. The output of DRAS-A2C splits into two parts: actor and critic. The actor outputs the policy π_θ for a given state s_k, which is the same as the output of DRAS-PG. The critic output is the value function $V(s_k)$. In DRAS-A2C, we subtract cumulative reward with value $V(s_k)$ and we call this value the advantage value $A(s_k, a_k) = \sum_{k'=k}^{K} r_{k'} - V(s_k)$. Intuitively, the advantage value presents how much better it is to take a specific action compared to the average action at the given state s_k. In addition, entropy regularization term is added to enforce exploration during learning [42]. Entropy measures the "randomness" of the policy: if the policy is fully deterministic (the same action is systematically selected), the entropy is zero as it carries no information; if the policy is completely random, the entropy is maximal. High entropy value encourages exploration by forcing the policy to become more random. In summary, DRAS-A2C method updates the neural network parameters θ by:

$$\theta \leftarrow \theta + \alpha \sum_{k=1}^{K} [\bigtriangledown_\theta log\pi_\theta(s_k, a_k) \left(\sum_{k'=k}^{K} r_{k'} - V(s_k) \right) + \beta \bigtriangledown_\theta H(\pi_\theta(s_k))]$$

Here, the value $V(s_k)$ replaces the baseline b_k in Equation (4) to provide more accurate value estimation. $H(\pi_\theta(s_k))$ is the entropy of the policy for a state s_k, which can be computed as: $H(\pi_\theta(s_k)) = -\sum_a \pi_\theta(s_k, a) log\pi_\theta(s_k, a)$.

DRAS-PPO was motivated by the fact that large improvement steps on a policy might accidentally cause performance collapse. To address this challenge, PPO provides two variants, i.e., penalty and clip, to limit large policy changes. Studies show that clipping is more effective and simpler than penalty [14]. Therefore, DRAS-PPO adopts clipping technique. In DRAS-PPO, we first define the probability ratio $r(\theta)$ between old policy $\pi_{\theta_{old}}(s_k, a_k)$ and new policy $\pi_\theta(s_k, a_k)$: $r(\theta) = \frac{\pi_{\theta_{old}}(s_k, a_k)}{\pi_\theta(s_k, a_k)}$. This ratio measures the

difference between the old and the new policies. If $r(\theta) > 1$ the action will be selected more in the new policy; if $r(\theta) < 1$ the action will be selected less in the new policy. In order to prevent performance collapse, PPO constructs a new objective function to clip the estimated advantage function if the new policy is far away from the old policy, i.e., $r(\theta) > 1 + \epsilon$, where ϵ is a clipping hypermeter. By adding the clipping, the new objective function becomes:

$$L^{CLIP}(\theta) = E[min(r(\theta)A_{\theta_{old}}(s,a), \\ clip(r(\theta), 1-\epsilon, 1+\epsilon)A_{\theta_{old}}(s,a))]$$

The clipping function discourages large policy change if it is outside predefined comfortable zones. Similar to DRAS-A2C, for the critic network, we seek to minimize the difference between the estimated value and the actual value.

Both A2C and PPO are hybrid algorithms that combine value based learning (DQL) and policy gradient (PG). According to the literature [14], such a hybrid approach consistently outperforms DQL and PG on various applications. Additionally, they are more robust and achieve better sample efficiency. In Sections 5.3 and 6.1, we also demonstrate PPO and A2C yield better performance than DQL and PG in solving the cluster scheduling problem.

3.3 Training Strategy

At the beginning of the training, we initialize DRAS's neural network parameters θ to random numbers. We train the neural network in episodes and the network parameters θ are updated with episodic training until convergence. For each episode, the environment is first set to its initial state (i.e., all nodes are idle and no jobs run on the system). We train DRAS via trace-based simulation, in which job events occur at a specific instant in time according to the job traces. DRAS observes the scheduling state, makes scheduling decisions according to its neural network, and collects scheduling rewards. For every ten scheduling instances, DRAS updates its parameters θ based on the collected observations and then clears the memory for the next update. An episode terminates when all jobs in the jobset (defined as a collection of jobs grouped together for training an episode) have been scheduled. We monitor the progress of the training by taking a snapshot of the model after each episode. The next episode uses a new jobset to refine the previous model.

The jobsets used in training determine the convergence and quality of the DRAS model. To learn a converged model, we follow the principle of gradual improvement: *DRAS starts with simple average cases and gradually improves its capability with unseen rare cases*. Specifically, we train DRAS by using a three-phase training process and three types of jobsets are used to train DRAS in order: (1) a set of sampled jobs randomly selected from real job traces, (2) a period of real job traces, and (3) a set of synthetic jobs generated according to job patterns on the target system. The sampled jobsets have controlled job arrival rates providing the easiest learning environment. Once DRAS can make good scheduling decisions under the controlled environment, training on the real job traces with various job arrival patterns allows DRAS to learn more challenging situations. The final phase is to train DRAS with synthetic jobsets, which enables DRAS to experience a variety of potential states that might not be seen in the first two types of jobsets. We will show this three-phase training process leads to a fast convergence (Section 5.3).

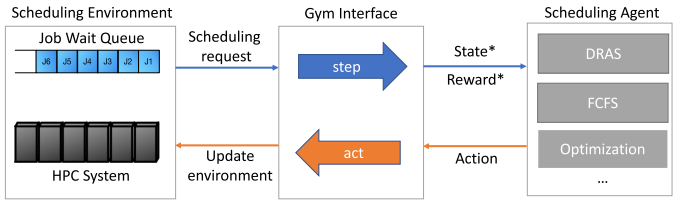


Fig. 2. Overview of CQGym. Here * denotes the information is passed from Gym interface to scheduling agent, while the information might not be used by the agent. For example, Gym interface sends state (job and system information) and reward to the scheduling agent, and the agent can select relevant information for its decision making, e.g., FCFC only needs to know job submission time.

The RL algorithms in DRAS can be implemented using various machine learning libraries, including Tensorflow [43] and PyTorch [44]. We implement DRAS-DQL and DRAS-PG in Tensorflow; we implement DRAS-A2C and DRAS-PPO in PyTorch. The source code is available as open-source on GitHub [21].

4 CQGym

Now we present CQGym, a platform to comprehensively evaluate different HPC scheduling policies under the same setting. CQGym consists of three main components, i.e., scheduling environment, Gym interface, and scheduling agent. Scheduling environment is an event-driven scheduling simulator that simulates job events, such as job submission, start, and end. The scheduling agent processes scheduling requests from the environment. The scheduling environment and agent are running on two separate threads. Gym provides a standard interface to bridge the environment and agent enabling their communication and coordination.

Fig. 2 provides an overview of CQGym. CQGym is implemented as a producer-consumer problem, where the scheduling environment behaves as the consumer and the scheduling agent acts as a producer. The scheduling environment and the agent run on two separate threads. They interact through Gym interface using shared variables.

Environment simulates the actual scheduling environment. Since Gym interface decouples the agent from the environment, we can adopt any HPC scheduling simulator [23], [45]. The default environment in CQGym is CQSim, which is a trace-based HPC job scheduling simulator that has been used for more than a decade [2]. A real system takes jobs from user submission, while a simulator takes jobs by reading the job arrival information in the trace. A simulator emulates execution by advancing the simulation clock according to the job runtime information in the trace. Changes in job wait queue or system trigger the simulator to send scheduling requests to the scheduling agent. The typical triggers are new job submitted to job queue and running job leaving the system.

Gym provides a standard interface between scheduling environment and scheduling agent. Our Gym interface follows the standard OpenAI Gym [22]. Gym is a widely

TABLE 2
Theta and Cori Workloads

LightCyan	Theta	Cori
Location	ALCF	NERSC
Scheduler	Cobalt	Slurm
System Types	Capability computing	Capacity computing
Compute Nodes	4,392 (4,392 KNL)	12,076 (2,388 Haswell; 9,688 KNL)
Trace Period	Jan. 2018 - Dec. 2019	Apr. 2018 - Jul. 2018
Number of Jobs	121,837	2,607,054
Max Job Length	1 day	7 days

adopted environment to compare the performance of different RL algorithms. OpenAI actively implements new RL algorithms to evaluate on Gym [46]. We follow the Gym standard in order to easily embrace new RL algorithms for HPC scheduling. Our Gym interface implements two functions, i.e., step and act. These functions provide standard inputs and outputs for both the environment and the agent. Therefore, the agent does not need to know the implementation details about the environment. Upon receiving a scheduling request from the environment, the step function pauses the simulation, collects job queue states, system states, and reward, then transforms the information in a standard format, and invokes the agent to make scheduling decisions based on the information. Then Gym waits on the agent for a response. Upon obtaining the scheduling decision from the agent, Gym interface instructs the environment to execute the action and resume the simulation. A salient feature is the Gym interface supports not only traditional scheduling policies, but also RL-based scheduling policies.

Agent makes scheduling decisions. In this study, we have implemented several scheduling agents in CQGym: the classic heuristic and optimization methods and four DRAS agents presented in Section 3.2. In order to embrace new RL algorithms benchmarked on Gym [46], we modularize DRAS into several components and expose the components, e.g., neural network structure, that need customization for HPC scheduling. In addition, DRAS provides a switch to turn on/off training. Once we train a RL agent with substantial experience and obtain a converged RL model, we can turn off training to make faster decisions in production.

5 Experimental Setup

5.1 Comparison Methods

- *FCFS/B* represents FCFS with EASY backfilling, which is the default scheduling policy deployed on many production supercomputers [28]. FCFS/B prioritizes jobs based on their arrival times and EASY backfilling is used to reduce resource fragmentation [1].
- *BinPacking* is widely used heuristic method for scheduling in datacenters [5]. It iteratively allocates the largest runnable jobs (i.e., job size is less than or equal to the number of available nodes in the system) until the system cannot accommodate any further jobs.
- *Random* randomly selects runnable jobs from the queue to execute until no more jobs in the queue can fit into the system. Since DRAS performs similarly to Random at the beginning of training by randomly exploring action space, if DRAS's performance is better than Random, it demonstrates that DRAS gradually learns to improve its scheduling action.
- *Optimization* denotes a suite of scheduling methods that formulate cluster scheduling as an optimization problem [3], [47], [48]. In our experiments, the optimization problem is formulated as a 0-1 knapsack problem which is solved using dynamic programming. For a fair comparison, we use the same scheduling objectives (i.e., Equations (1) and (2)) for Optimization and for DRAS. For example, we can formulate the capability computing optimization problem as:

$$\max \left(w_1 \times \frac{\overline{t}}{t_{max}} + w_2 \times \frac{\overline{n}}{N} + w_3 \times \frac{N_{used}}{N} \right)$$
$$s.t. \quad N_{selected} \leq N_{avaiable}$$

  Per scheduling instance, optimization method selects several waiting jobs to execute to maximize the objective with the constraint that the resource needed from the selected jobs does not exceed available resource in the system.
- *Decima-PG* denotes a modified version of Decima [6]. Since Decima is not designed for scheduling HPC jobs, we modify Decima by skipping the graph neural network and adopting our state representation presented in Section 3.1. Decima-PG acts as the baseline to demonstrate the benefits of the hierarchical design of DRAS.
- *DRAS* denotes our HPC custom designs scheduling. In this study, we study four different RL agents, namely DQL, PG, A2C, and PPO.

Among these methods, FCFS/B and DRAS are equipped with reservation and backfilling strategies. Optimization does not have backfilling and reservation capability. It is challenging to incorporate reservation and backfilling into optimization formulation, i.e., constraints, therefore they are not considered in existing optimization studies.

5.2 Workload Traces

In our study, two real workload traces are used. Table 2 summarizes the two traces collected from production systems, and Fig. 3 gives an overview of job size distributions. We select these traces as they represent different workload profiles: (1) capability computing focusing on solving large-sized problems, (2) capacity computing solving a mix of small-sized and large-sized problems. The first workload is

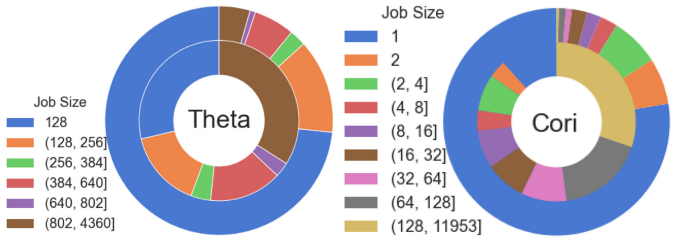


Fig. 3. Job characterization of Theta at ALCF and Cori at NERSC. The outer circle shows the number of jobs in each job size category. The inner circle presents the total core hours consumed by each job size category.

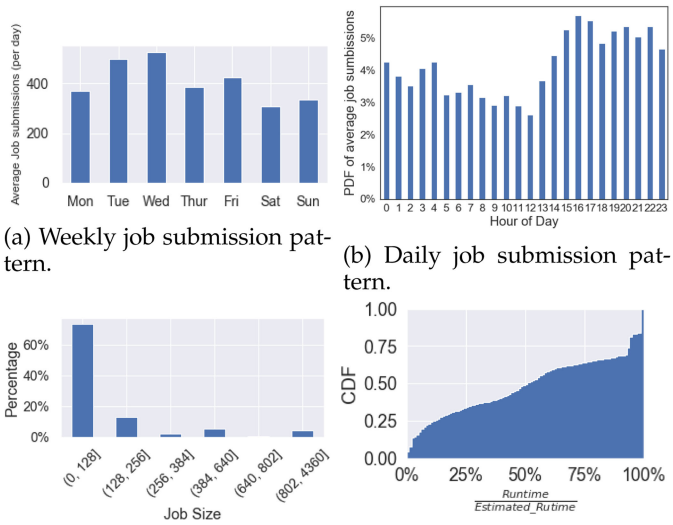
a two-year job log from Theta [49], the production HPC system located at ALCF. Theta is a capability computing system. The smallest job allowed on Theta is 128-node [50]. Only 2.25% of jobs have dependency. For jobs with dependency, the scheduler hides them from scheduling until all their parents have been executed. On Theta, there are 32 nodes dedicated to run debugging jobs and the rest of 4,360 nodes are dedicated to user jobs. In our experiments, we set the system size to be 4,360 and filter out all debugging jobs in the trace. *We use the first 2-month data for training, the next month data for validating model convergence, and the rest 21-month data for testing.*

The second trace is a four-month job log from Cori [51]. Cori is a capacity computing system deployed at NERSC. A majority of its jobs consume one or several nodes (Fig. 3). The longest job executed for seven days. *We use the first 2-week data for training, the next 1-week data for validating model convergence, and the last 15-week data for testing.*

5.3 DRAS Training

The details of RL architectures for these systems are listed in Table 3. Take *DRAS-PG* on Theta as an example. The input of the neural network is a vector of $[4460, 2]$. We use a convolutional layer with 4460 neurons and two fully-connected layers with 4000 and 1000 neurons respectively. The output layer contains 50 neurons representing jobs in the window. In total, the neural network has 21,890,053 trainable parameters. We shall note that the neural network is configured based on the machine size, learning capability, and computation overhead.

For Theta (capability computing), we define its reward as Equation (1). We set the weights $w_1 = w_2 = w_3 = 1/3$. For Cori (capacity computing), we set the reward as Equation (2). The learning rate α is set to 0.001.



(a) Weekly job submission pattern.

(b) Daily job submission pattern.

(c) Job size distribution.

(d) Accuracy of job runtime estimates supplied by users.

Fig. 4. Job patterns of Theta training dataset.

We use 100 jobsets composed of 720,000 jobs for DRAS training on Theta. We collect 9 sampled jobsets by randomly selecting jobs from the original training trace and modeling job arrival times as Poisson distribution following the average inter-arrival time of the original trace. We split the original Theta training dataset into nine one-week jobsets. We generate 82 synthetic jobsets that mimic Theta workload patterns in terms of hourly and daily job arrivals, and distributions of job sizes and runtimes (Fig. 4).

We validate the trained DRAS agent with an unseen validation dataset (i.e., March of 2018). Fig. 5 compares the convergence rates by training DRAS in different jobset orderings. We make two key observations. First, training only with real jobsets (the first 9 episodes of the orange line) cannot obtain a converged model. To achieve a converged model, more jobsets are needed to train our agents. Second, training order plays an important role in performance. Training in the order of sampled, real and synthetic jobsets achieves the best result. While training with real jobsets first can also obtain a converged model, the performance is not as good as the case of training with sampled jobsets first. Training with synthetic jobsets first results in slow convergence. In summary, *in order to generate a converged and high-quality model, DRAS needs to first learn from simple averaged cases (sampled jobsets) and then gradually move to more complicated special cases (real and synthetic jobsets).*

TABLE 3
DRAS Network Configurations for Theta and Cori

	Theta				Cori			
	DRAS-DQL	DRAS-PG	DRAS-A2C	DRAS-PPO	DRAS-DQL	DRAS-PG	DRAS-A2C	DRAS-PPO
Input	$[4362, 2]$	$[4460, 2]$	$[4460, 2]$	$[4460, 2]$	$[12078, 2]$	$[12176, 2]$	$[12176, 2]$	$[12176, 2]$
Convolutional Layer	4368	4460	4460	4460	12078	12176	12176	12176
Fully Connected Layer 1	4000				10000			
Fully Connected Layer 2	1000				4000			
Output	1	50	51	51	1	50	51	51
Trainable Parameters	21,449,004	21,890,053	21,891,054	21,891,054	161,764,004	161,960,053	161,964,054	161,964,054

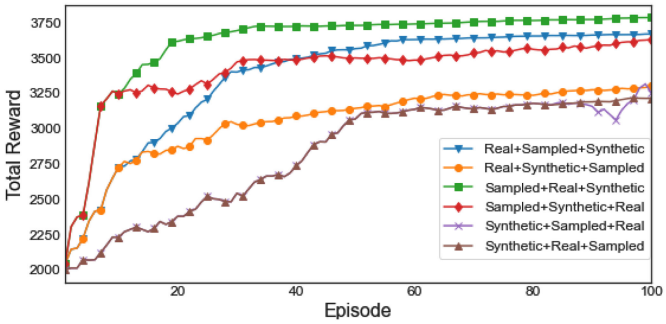


Fig. 5. Comparison of quality and convergence of DRAS-PG by training it in different jobset orders (Section 3.3).

Fig. 6 shows the learning curves of different scheduling methods. Our three-phase training process allows DRAS to quickly learn and surpass other competing methods and converge to optimal solutions. Based on the results, we use the model trained after the 50th episode for testing in Section 6. Another observation is that DRAS-PPO and DRAS-A2C converge faster and obtain higher total reward than DRAS-PG and DRAS-DQL. DRAS-PPO obtains 3800 rewards at the 100th episode compared with 3700 rewards of DRAS-PG and DRAS-DQL. In addition, DRAS-PPO and DRAS-A2C converge on the 4th sampled log and the 6th real log. On the other hand, DRAS-DQL and DRAS-PG take more episodes to reach similar rewards. The result indicates that DRAS-PPO and DRAS-A2C achieve better sample efficiency, requiring fewer historical jobs to train high-quality models.

We perform a similar training and validation process on Cori. We train DRAS using 100 jobsets (20,000,000 jobs) composed of sampled traces, real traces, and synthetic traces. Both *DRAS* methods converge at the 40th episode. Hence, we use the model trained after the 40th episode for testing.

5.4 Evaluation Metrics

There are two classes of metrics for evaluating cluster scheduling: user-level metrics and system-level metrics. In our experiments, we measure four well-established metrics:

- *Job wait time* is a user-level metric. It measures the interval between job submission to job start time. In our experiments, we analyze average job wait time, maximum job wait time, as well as the distribution of job wait times.

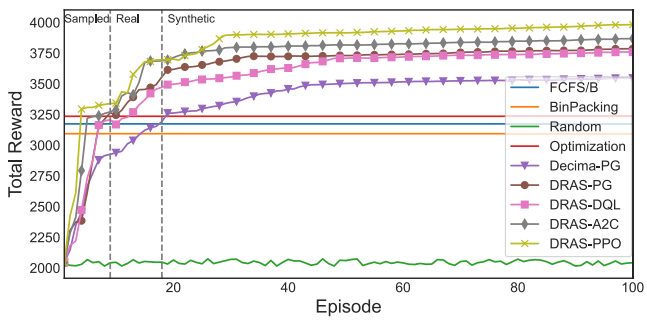


Fig. 6. The total reward collected by the different scheduling methods on Theta validation dataset.

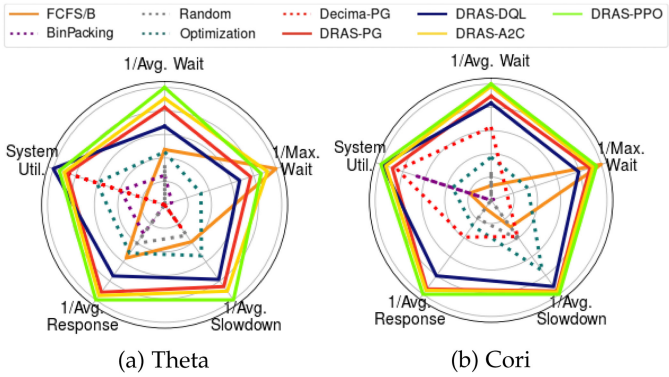


Fig. 7. Scheduling performance comparison using Kiviat graphs: Theta (left) and Cori (right). Solid lines represent the methods with backfilling and reservation strategies, while dashed lines represent the methods without these strategies. All metrics are normalized to the range of 0 to 1. 1 means a method achieves the best performance among all methods and 0 means a method obtains the worst performance. The larger the area is, the better the overall performance is.

- *Job response time* is a user-level metric which measures the interval between job submission to completion.
- *Job slowdown* is another user-level metric. It measures the ratio of the job response time to its actual runtime.
- *System utilization* is a system-level metric. It measures the ratio of the used node-hours for useful job execution to the total elapsed node-hours.

6 Case Study

We present the case study using CQGym to compare four DRAS agents and five existing scheduling methods on test data (i.e., 21-month Theta log and 15-week Cori log). Our experiments are designed to answer the following questions:

1) Does DRAS outperform existing, non-learning-based scheduling policies? (Section 6.1)
2) Can DRAS prevent jobs from starvation? (Section 6.2)
3) If DRAS outperforms the traditional scheduling methods, where does the gain come from? (Section 6.3)
4) Which DRAS agent performs the best? (Section 6.4)
5) Can DRAS adapt to workload changes? (Section 6.5)
6) Is generic or customized RL model better? (Section 6.6)?
7) Do DRAS agents cause scheduling delays? (Section 6.7)

6.1 Scheduling Performance

The quality of a scheduling method needs to be evaluated by multiple metrics, including both system-level and user-level metrics. Fig. 7 presents the overall scheduling performance obtained by different scheduling methods. *DRAS* yields the best result. Although *FCFS/B* has the lowest maximum wait time, it has poor performance on the rest of the metrics. All *DRAS* methods outperform *Optimization*, suggesting that *DRAS* agents learn to select jobs that not only maximize the immediate reward, but also potentially improve performance in the future through maximizing cumulative reward. *Decima-PG* achieves good performance

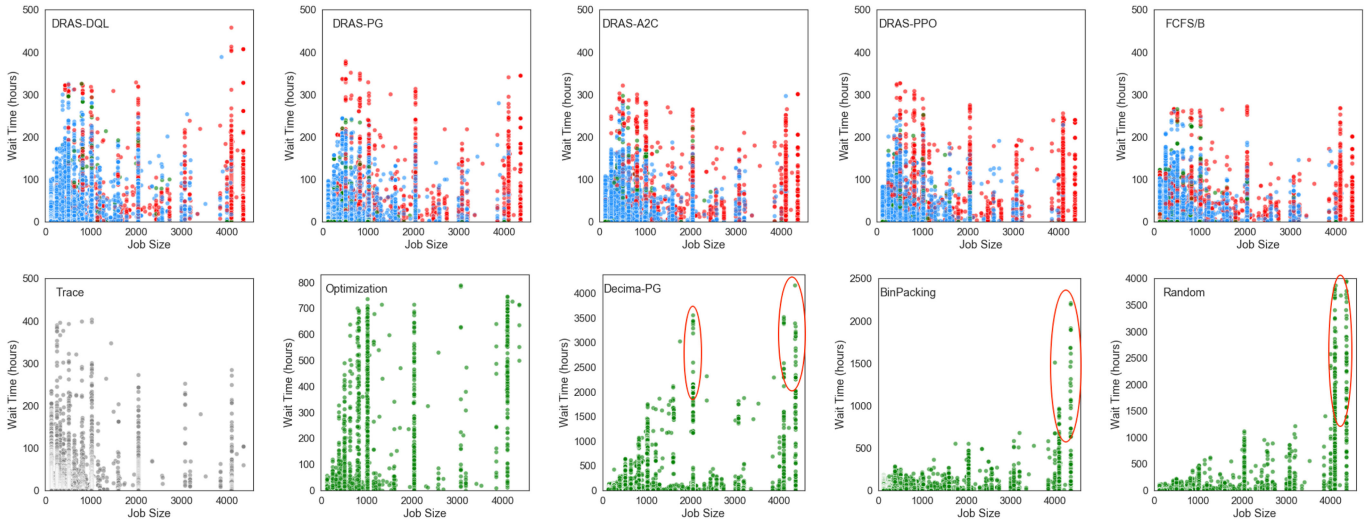


Fig. 8. Job wait time distributions with respect to job size and job type on Theta. Note that the y-axis scale for *Decima-PG*, *BinPacking* and *Random* is much larger than those for others. *Trace* presents the job wait times extracted from the original log, which can be used as the baseline. Since we do not have the job type information, all jobs are marked in grey. Eclipses in the plots indicate *Decima-PG*, *BinPacking*, and *Random* lead to severe job starvation.

on system utilization, but it fails to improve user-level metrics. *BinPacking* and *Random* have the worst performance, because they greedily select jobs one by one which ignores the best job combinations. Recall that *DRAS* applies a similar strategy as *Random* at the beginning of the training, by randomly exploring various actions, the better performance of *DRAS* indicates that our RL models developed good policies through learning.

Some methods have inconsistent performance on the user-level metrics. For example, FCFS/B achieves the lowest maximum job wait time; however, it suffers from high average job wait time. We present the in-depth analysis of job wait time on Theta in the following subsections.

6.2 Job Starvation Analysis

Fig. 8 shows job wait times under different job sizes and categories. We make three key observations from this figure. First, *DRAS* and *FCFS/B* prevent jobs from starvation, while *Decima-PG*, *BinPacking* and *Random* suffer severe job starvation. The maximum job wait times of DRAS methods are in the range of 14 to 20 days, which are only slightly higher than the maximum job wait time of *FCFS/B* (13 days) and are similar to *Trace*. Although *Optimization* and *DRAS* aim at the same scheduling objectives, the maximum wait time of *Optimization* is twice as long as that of *DRAS*. The maximum job wait times of *Decima-PG*, *BinPacking* and *Random* are 170 days, 95 days, and 170 days, indicating they are not suitable for HPC cluster scheduling. Second, in *Decima-PG*, *BinPacking*, and *Random*, large-sized jobs wait noticeably more time than small-sized jobs. They inherently give higher priority to small-sized jobs at the expense of large-sized jobs, because they lack reservation strategies to reserve resources for large-sized jobs. The bias toward small-sized jobs is not ideal for HPC scheduling, especially for capability systems. In contrast, the methods with reservation strategy, i.e., *FCFS/B* and *DRAS*, do not have a significant difference between small jobs' wait times and large jobs' wait times. This demonstrates that *DRAS* and *FCFS/B* are relatively fair scheduling policies. Third, if we take a look at the methods using reservation and backfilling strategies (i.e., *FCFS/B* and *DRAS*), we notice that almost all large jobs are executed through reservation, while the majority of small jobs are executed through backfilling. In short, these results demonstrate that DRAS is capable of preventing job starvation mainly due to the incorporation of job reservation and backfilling in our DRAS design.

6.3 Source of DRAS Performance Gain

Table 4 presents job distributions by using different scheduling methods. We notice that although *DRAS* backfills a majority of the jobs, most node hours are consumed by reserved jobs. If we read Table 4 along with Fig. 8, we observe that there are a few jobs with wait time of over 300 hours and these jobs are mainly allocated through reservation by DRAS. Without reservation, these jobs would wait for 2X-10X more time as happened in *Decima-PG*, *Optimization*, *BinPacking*, and *Random*. Put together, these results reveal that all DRAS agents learn to achieve the goals by prioritizing jobs and preventing job starvation through its reservation mechanism embedded in its two-level network design.

TABLE 4
Job Distributions in Different Execution Models (Defined in Section 3.2) on Theta

	Backfilled		Ready		Reserved	
	jobs	core hours	jobs	core hours	jobs	core hours
Optimization	0%	0%	100%	100%	0%	0%
Decima-PG	0%	0%	100%	100%	0%	0%
BinPacking	0%	0%	100%	100%	0%	0%
Random	0%	0%	100%	100%	0%	0%
FCFS/B	79.25%	30.45%	9.88%	16.99%	10.87%	52.56%
DRAS-DQL	84.83%	34.17%	6.84%	10.91%	15.17%	54.92%
DRAS-PG	83.76%	33.67%	8.63%	11.29%	7.61%	55.04%
DRAS-A2C	80.36%	38.48%	10.60%	13.95%	9.03%	47.56%
DRAS-PPO	79.73%	38.57%	10.96%	13.39%	9.30%	48.03%

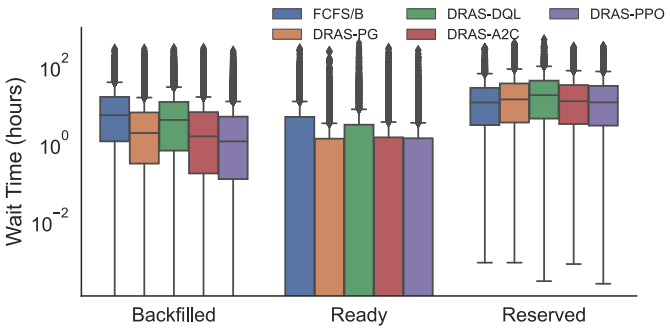


Fig. 9. Box plot of job wait times, grouping by job execution modes. Note that the y axis presents in log scale. As compared to *FCFS/B*, *DRAS* learns to intelligently select jobs for immediate execution, reservation, or backfilling so as to maximize the overall scheduling performance.

Although both *FCFS/B* and *DRAS* apply backfilling strategies, *DRAS* performs significantly better in terms of average job wait time (Fig. 7). In Fig. 9, we notice that DRAS largely reduces the wait time of ready and backfilled jobs at the expense of a slightly longer wait time for reserved jobs. *FCFS/B* schedules jobs in their arrival order, while *DRAS* selects jobs from the queue that aim to balance three objectives (i.e., minimizing average job wait time, prioritizing large jobs, and maximizing system utilization). Therefore, *DRAS* learns to pick backfilled and ready jobs that lead to lower average job wait time and select jobs queued for long times to avoid job starvation. The better performance of *DRAS* demonstrates that DRAS learns to intelligently select jobs for resource allocation so as to maximize the long-term scheduling performance.

6.4 Comparison of DRAS Agents

We make several observations by comparing the performance of the four DRAS agents, i.e., DRAS-DQL, DRAS-PG, DRAS-A2C, and DRAS-PPO. First, DRAS-PPO achieves the best overall performance on both Theta and Cori traces as shown in Fig. 7. DRAS-A2C obtains the second-best performance, which is very close to DRAS-PPO. DRAS-DQL is the worst among the four DRAS agents. The result indicates that the advanced actor-critic model and clipping technique can reduce the variance in predicting the baseline performance and therefore choose jobs with higher rewards. Second, as shown in Fig. 8, DRAS-PPO is the most effective in preventing large jobs from starvation. For DRAS-PPO, the maximum wait time of large jobs is less than 300 hours, which is lower than that of small jobs (325 hours). This result demonstrates that DRAS-PPO learns to penalize those large jobs waiting in the queue for a long time. For DRAS-PG and DRAS-A2C, no significant difference is observed in the maximum wait times of small jobs and large jobs. DRAS-DQL is less effective in terms of job starvation prevention. For DRAS-DQL, the maximum wait time of large jobs is 460 hours, while the maximum wait time of small jobs is 320 hours. Third, as shown in Table 4, DRAS-PPO and DRAS-A2C have higher percentage of ready jobs and lower percentage of reserved core hours compared with DRAS-PG and DRAS-DQL. In addition, in Fig. 9, the average wait times of reserved jobs in DRAS-PPO and DRAS-A2C are lower than that of DRAS-PG and DRAS-DQL. This suggests that DRAS-PG and DRAS-DQL are more intelligent in selecting ready jobs in the first-level neural network and therefore leave fewer job core hours that need reservation. In summary, DRAS-PPO and DRAS-A2C leverage advanced techniques to reduce reward variance and obtain better models for deployment.

6.5 Adaptation to Workload Change

As shown in the top subfigure in Fig. 10, the system loads are dynamically changing over time. We observe several

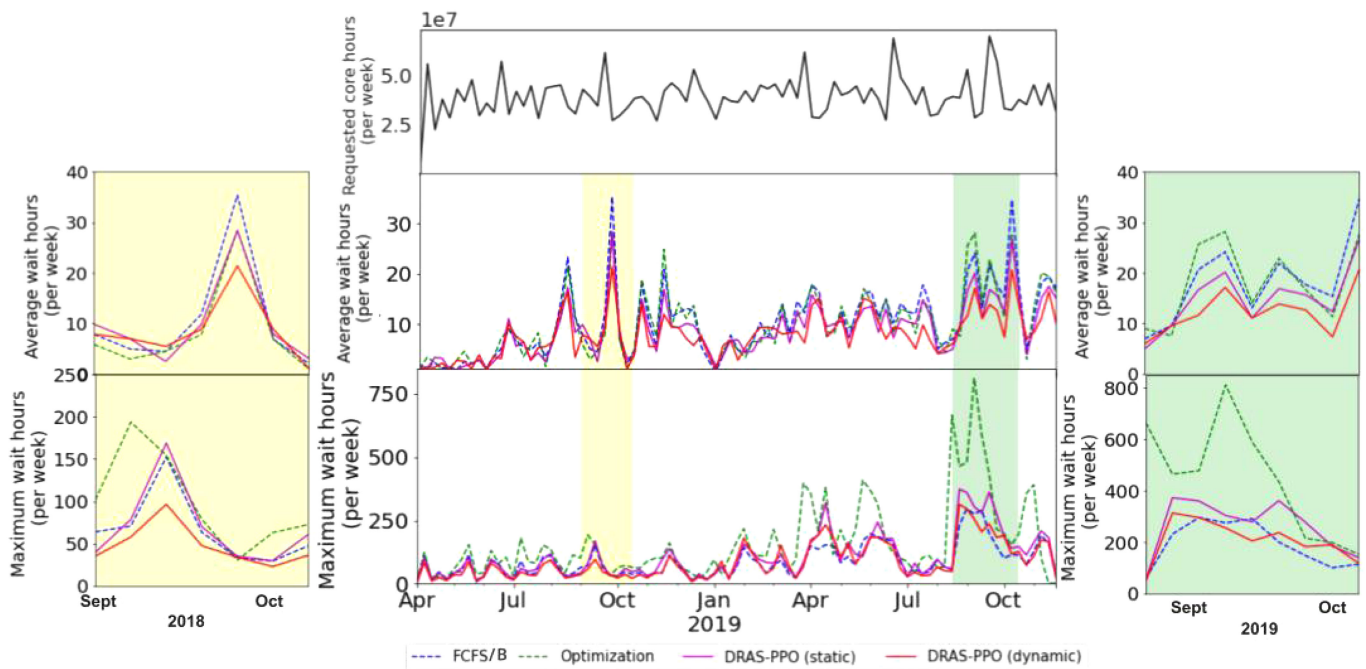


Fig. 10. Comparing scheduling performance of FCFS/B, Optimization and DRAS-PPO, under heavy workload. When the system load is high, the dynamic DRAS-PPO adjusts it network parameters to reduce average job wait time and maximum job wait time. Two periods of busy times are zoomed in at right and left.

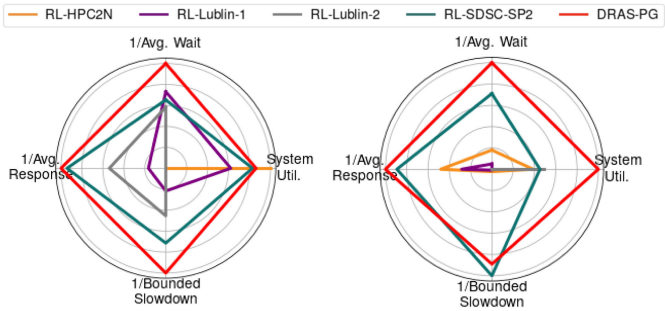


Fig. 11. DRAS_PG versus RLScheduler-learned models.

dramatic demand surges during this one-and-half year period. Unlike the existing static scheduling policies, DRAS is capable of dynamically adjusting its policy after deployment by continuous training. In Fig. 10, we compare the scheduling performance of two selected static methods, i.e., FCFS/B and Optimization, with DRAS-PPO. We select these methods for comparison for the following reasons: FCFS/B is the baseline algorithm; Optimization achieves the best performances among all methods without backfilling and reservation strategies; DRAS-PPO is the DRAS agent with the best performance. In order to show the policy adjustment capability, we provide two versions of DRAS-PPO: static DRAS-PPO that freezes its neural network parameters during testing, and dynamic DRAS-PPO that dynamically adjusts its neural network during testing. The neural network update frequency during testing is much lower than that during training, because we need a stable model and only adjust models when workload dramatically changes.

The bottom two subfigures in Fig. 10 show the average and maximum job wait time during the testing period. During busy weeks, FCFS/B has the worst performance on average wait times, while Optimization has the longest maximum job wait time, especially in September 2019. Our further investigation in September 2019 trace reveals that users submitted 6 whole-system jobs each running 24 hours. Normally, there are almost 2 such jobs submitted per month. It is challenging to schedule such large and long jobs: running them early could block all other jobs causing long average wait times; running them late leads to long maximum wait times. Minimizing average wait time and minimizing maximum wait time are two conflicting goals. Optimization chooses to delay large and long jobs causing long maximum wait time. This is because Optimization does not have reservation and backfilling strategies to prevent large jobs from starvation. This demonstrates that reservation and backfilling strategies are especially effective in preventing large jobs from starvation under heavy workloads. It is clear that DRAS-PPO is capable of balancing these two conflicting goals by reducing average wait time with only a slight increase in maximum job wait time compared with FCFS/B. The dynamic DRAS-PPO performs even better than the static version under heavy workloads. This demonstrates that dynamic adjusting neural networks helps the DRAS agent adapt to dramatic workload changes.

6.6 Generic versus Customized RL Agent

RLScheduler [17] trained generic PG-based models based on several workloads: HPC2N, Lublin-1, Lublin-2, and SDSC-SP2. The learned model can apply on other unseen workloads. In contrast to RLScheduler's generic approach, we adopt a customized approach, which learn the traits of a specific system and workload through its historical traces. We compare the performance of RLScheduler learned models and our DRAS-PG on Theta and Cori test data in Fig. 11. Clearly, DRAS-PG outperforms all three RLScheduler models. This suggests that customized RL based scheduler can capture system- and workload-specific features and thus deliver better performance.

6.7 Runtime Overhead

The experiments in this paper are conducted on a personal computer configured with Intel quad-core 2.6 GHz CPU with 16 GB memory. *DRAS-PG*, *DRAS-A2C* and *DRAS-PPO* take less than 1 second for each network parameter update. Since DRAS-DQL only processes one job at each time, its network parameter update takes longer time ($\leq$2 seconds) than other DRAS agents. In practice, HPC cluster scheduling is typically required to make decisions in 15-30 seconds [3]. In other words, all *DRAS* agents impose trivial overhead, hence being feasible for online deployment.

In our experiments, we spent less than 3 hours on a personal computer to obtain a converged model in Fig. 6. Any system change, such as adding or removing nodes in a system, requires DRAS to re-train the model. Considering that system changes are not very frequent and DRAS can avoid complicated manual tuning policies, it is worth to re-train the model when the system changes.

7 Conclusion

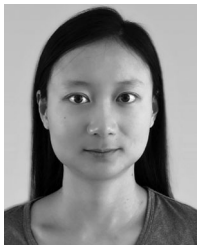
In this paper, we have designed DRAS, a RL-empowered job scheduling agent tailored for HPC systems and workloads. DRAS represents the scheduling policy as a hierarchical neural network and automatically learns customized policies through training with the system-specific workloads. Four RL algorithms, i.e., DQL, PG, A2C, and PPO, have been implemented as DRAS agents. To bridge RL and HPC scheduling, we have developed CQGym, a common platform to quantitatively evaluate various HPC job scheduling policies, including both manually designed policies and DRAS agents. Our case study demonstrates that DRAS is capable of grasping system- and workload-specific characteristics, preventing large jobs from starvation, adapting to workload changes without human intervention, and outperforming existing scheduling policies by up to 50%.

References

[1] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 6, pp. 529–543, Jun. 2001.

[2] X. Yang et al., "Integrating dynamic pricing of electricity into energy aware scheduling for HPC systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2013, pp. 1–11.

[3] Y. Fan et al., "Scheduling beyond CPUs for HPC," in *Proc. 28th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2019, pp. 97–108.
[4] H. Sun, P. Stolf, J. Pierson, and G. Costa, "Energy-efficient and thermal-aware resource management for heterogeneous datacenters," *Sustain. Comput.: Informat. Syst.*, vol. 4, pp. 292–306, 2014.
[5] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 455–466.
[6] H. Mao, M. Schwarzkopf, S. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM Special Int. Group Data Commun.*, 2019, pp. 270–288.
[7] Y. Fan, Z. Lan, T. Childers, P. Rich, W. Allcock, and M. E. Papka, "Deep reinforcement agent for scheduling in HPC," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 807–816.
[8] A. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electron. Imag.*, vol. 19, pp. 70–76, 2017.
[9] T. Johannink et al., "Residual reinforcement learning for robot control," in *Proc. Int. Conf. Robot. Automat.*, 2019, pp. 6023–6029.
[10] V. Mnih et al., "Playing Atari with deep reinforcement learning," in *Proc. NIPS Deep Learn. Workshop*, 2013.
[11] D. Silver et al., "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.
[12] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2017.
[13] V. Mnih et al., "Asynchronous methods for deep reinforcement learning," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 1928–1937.
[14] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv: 1707.06347*.
[15] H. Mao et al., "Park: An open platform for learning-augmented computer systems," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst.*, 2019, Art. no. 224.
[16] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Topics Netw.*, 2016, pp. 50–56.
[17] D. Zhang, D. Dai, Y. He, F. S. Bao, and B. Xie, "RLScheduler: An automated HPC batch job scheduler using reinforcement learning," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, Art. no. 31.
[18] R. Cunha and L. Chaimowicz, "Towards a common environment for learning scheduling algorithms," in *Proc. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2020, pp. 1–8.
[19] S. Liang, Z. Yang, F. Jin, and Y. Chen, "Data centers job scheduling with deep reinforcement learning," in *Proc. Pacific-Asia Conf. Knowl. Discov. Data Mining*, 2020, pp. 906–917.
[20] F. Li and B. Hu, "DeepJS: Job scheduling based on deep reinforcement learning in cloud data center," in *Proc. 4th Int. Conf. Big Data Comput.*, 2019, pp. 48–53.
[21] CQGym Github Repository, 2021. [Online]. Available: https://github.com/SPEAR-IIT/CQGym
[22] OpenAI Gym, 2020. [Online]. Available: https://gym.openai.com/
[23] CQSim Github Repository, 2021. [Online]. Available: https://github.com/SPEAR-IIT/CQSim
[24] B. Li, S. Chunduri, K. Harms, Y. Fan, and Z. Lan, "The effect of system utilization on application performance variability," in *Proc. 9th Int. Workshop Runtime Operating Syst. Supercomput.*, 2019, pp. 11–18.
[25] Y. Fan and Z. Lan, "DRAS-CQSim: A reinforcement learning based framework for HPC cluster scheduling," *Softw. Impacts*, vol. 8, 2021, Art. no. 100077.
[26] W. Allcock, P. Rich, Y. Fan, and Z. Lan, "Experience and practice of batch scheduling on leadership supercomputers at argonne," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 2017, pp. 1–24.
[27] L. Yu, Z. Zhou, Y. Fan, M. Papka, and Z. Lan, "System-wide trade-off modeling of performance, power, and resilience on petascale systems," *J. Supercomput.*, vol. 74, pp. 3168–3192, 2018.
[28] M. Jette, A. Yoo, and M. Grondona, "SLURM: Simple Linux utility for resource management," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, 2003, pp. 44–60.
[29] Moab, 2019. [Online]. Available: http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/
[30] PBS Professional, 2019. [Online]. Available: http://www.pbsworks.com/
[31] Cobalt, 2019. [Online]. Available: https://www.alcf.anl.gov/cobalt-scheduler
[32] Y. Fan, P. Rich, W. Allcock, M. E. Papka, and Z. Lan, "Trade-off between prediction accuracy and underestimation rate in job runtime estimates," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2017, pp. 530–540.
[33] W. Chen, Y. Xu, and X. Wu, "Deep reinforcement learning for multi-resource multi-machine job scheduling," 2017, *arXiv: 1711.07440*.
[34] E. İpek, O. Mutlu, J. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proc. Int. Symp. Comput. Archit.*, 2008, pp. 39–50.
[35] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, 2015.
[36] V. Mnih et al., "Asynchronous methods for deep reinforcement learning," in *Proc. 33rd Int. Conf. Mach. Learn.*, 2016, pp. 1928–1937.
[37] J. X. Wang et al., "Learning to reinforcement learn," 2016.
[38] OpenAI Baselines: ACKTR & A2C, 2021. [Online]. Available: https://openai.com/blog/baselines-acktr-a2c/
[39] NERSC Queue Policies, 2020. [Online]. Available: https://docs.nersc.gov/jobs/policy/
[40] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.
[41] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015.
[42] R. J. Williams and J. Peng, "Function optimization using connectionist reinforcement learning algorithms," *Connection Sci.*, vol. 3, pp. 241–268, 1991.
[43] Tensorflow, 2020. [Online]. Available: https://www.tensorflow.org/
[44] PyTorch, 2020. [Online]. Available: https://pytorch.org/
[45] N. A. Simakov et al., "A Slurm simulator: Implementation and parametric analysis," in *Proc. Int. Workshop Perform. Model. Benchmarking Simul. High Perform. Comput. Syst.*, 2018, pp. 197–217.
[46] OpenAI Baselines, 2020. [Online]. Available: https://github.com/openai/baselines
[47] S. Wallace et al., "A data driven scheduling approach for power management on HPC systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 656–666.
[48] L. Rao, X. Liu, L. Xie, and W. Liu, "Minimizing electricity cost: Optimization of distributed internet data centers in a multi-electricity-market environment," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.
[49] Theta, 2020. [Online]. Available: https://www.alcf.anl.gov/theta
[50] Job Scheduling Policy for Theta, 2020. [Online]. Available: https://www.alcf.anl.gov/support-center/theta/job-scheduling-policy-theta
[51] Cori, 2020. [Online]. Available: https://docs.nersc.gov/systems/cori/



Yuping Fan received the MS degree in electrical engineering and the PhD degree in computer science, both from the Illinois Institute of Technology, in 2015 and 2021. Her research interests include parallel and distributed computing, resource management, and job scheduling in large-scale systems. She is a student member of the IEEE Computer Society.

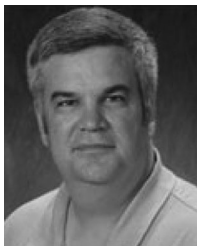
Boyang Li received the MS degree in electrical engineering from Clarkson University, in 2016. He is currently working toward the PhD degree in computer Science with the Illinois Institute of Technology. His research interests include resource management and job scheduling on HPC systems.

Dustin Favorite received the BS degree in computer science from the University of Central Arkansas, in 2017, and the MS degree from the Illinois Institute of Technology, in May 2022. His research interests include machine learning research methods and application, particularly reinforcement learning.

Naunidh Singh received the BTech degree in mathematics and computing from Delhi Technological University, in 2015, and the MS degree in computer science from the Illinois Institute of Technology, in 2021. He is currently a software engineer with Google Cloud. His research interest include parallel and distributed computing in general.

Taylor Childers is a computer scientist with ALCF. He has a background in high-energy physics, having worked with the CERN Laboratory in Geneva Switzerland on the ATLAS experiment. His experience focuses on applying machine learning methods to scientific challenges including reinforcement learning, classification, and semantic segmentation methods, and scaling machine learning model training and scientific simulations.

Paul Rich received the BA degree from Northwestern University, in 2003, and the MS degree from the University of Chicago, in 2006. He is a principal software development specialist with ALCF. He is a member of the Operations team and works primarily on large scale system scheduling, schedulers and scheduling policies. He is the lead developer for Cobalt and is also a contributor to OpenPBS. His main interests are system software, especially exascale system software, and HPC scheduling.



William Allcock received the bachelors of science degree in computer science from the University of Wisconsin — Oshkosh, and the master's of science degree in paper science from the Institute of Paper Science and Technology, Atlanta, GA. He has been working with Argonne National Laboratory in various scientific computing domains for more than 20 years. He is the director of operations for the ALCF and was the team lead for the GridFTP Software Team.

Michael E. Papka is a senior scientist and deputy associate laboratory director for computing, environment, and life sciences with Argonne National Laboratory. At Argonne, he directs the Argonne Leadership Computing Facility. He is interested in scientific visualization, large-scale data analysis, and the development and deployment of research infrastructure in support of science. He is also a professor of computer science with the University of Illinois Chicago.

Zhiling Lan received the PhD degree in computer engineering from Northwestern University, in 2002. She has since joined the faculty of the Illinois Institute of Technology and is currently a professor of computer science. She is also a guest research faculty with the Argonne National Laboratory. Her research interests include workload management, interconnect networking, performance modeling, and simulation. She is a senior member of the IEEE Computer Society.