Future Generation Computer Systems 139 (2023) 239–252

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

An SMDP approach for Reinforcement Learning in HPC cluster schedulers

Renato Luiz de Freitas Cunha [a,b,*], Luiz Chaimowicz [a]

[a] Programa de Pós Graduação em Ciência da Computação, Av. Antônio Carlos, 6627, Belo Horizonte, 31270-901, MG, Brazil
[b] Microsoft, Av. Presidente J. Kubitscheck, 1909, Torre Sul, 16° andar, São Paulo, 04543-907, SP, Brazil

ARTICLE INFO

Article history:
Received 18 April 2022
Received in revised form 4 August 2022
Accepted 26 September 2022
Available online 7 October 2022

Dataset link: https://github.com/renatolfc/sched-rl-gym/

Keywords:
Deep Reinforcement Learning
Scheduling
Semi-Markov Decision Processes
Workload traces
Machine Learning
Simulation

ABSTRACT

Deep reinforcement learning applied to computing systems has shown potential for improving system performance, as well as faster discovery of better allocation strategies. In this paper, we map HPC batch job scheduling to the SMDP formalism, and present an online, deep reinforcement learning-based solution that uses a modification of the Proximal Policy Optimization algorithm for minimizing job slowdown with action masking, supporting large action spaces. In our experiments, we assess the effects of noise in run time estimates in our model, evaluating how it behaves in small (64 processors) and large (16384 processors) clusters. We also show our model is robust to changes in workload and in cluster sizes, showing transfer works with changes of cluster size of up to 10×, and changes from synthetic workload generators to supercomputing workload traces. In our experiments, the proposed model outperforms learning models from the literature and classic heuristics, making it a viable modeling approach for robust, transferable, learning scheduling models.

1. Introduction

As Machine Learning (ML) techniques become more popular, they begin to permeate all systems we interact with, including computing systems. We are now at a point that ML models are trained in High Performance Computing (HPC) clusters, and also being used to optimize various aspects related to the execution of jobs. With ML being integrated with HPC, new types of workloads arise, creating the need to systems that adapt to the type of work being submitted.

Current work on learning schedulers for HPC clusters has overlooked the effects of uncertainty in job run time estimates on such schedulers, having an implicit assumption that such estimates are accurate. We know from the literature on scheduling performance of HPC schedulers that, more often than not, uncertainty in estimates is detrimental to scheduling performance, which encouraged researchers to devise techniques to improve such estimates, or determine them automatically. Still, inaccurate run time estimates are the norm and, hence, scheduling systems, learned or not, have to deal with them.

Moreover, some works in the literature focus on learning schedulers for systems as if their configuration (*e.g.*, number of processors) never changes, which is also unrealistic, especially when HPC Cloud resources [1,2] are used, with some even arguing in favor of learning cluster-specific schedulers which are tied to the number of resources available [3]. The problem with this approach is that clusters tend to undergo changes during their lifetimes, be it by the addition or removal of computing nodes, or by partitioning of the cluster for specific projects, for example. We believe it is more important to be able to learn schedulers that can potentially work with any cluster, while enabling the learned scheduler to be tuned to the current state of the cluster.

We believe the task of learning a scheduler is an ideal setting for applying Reinforcement Learning (RL), as scheduling decisions directly influence the system the RL agent interacts with, as well as its dynamics. Fig. 1 displays a schematic view of the problem we are solving. On the left, it shows the agent, parameterized by the weights of a neural network. The output of the neural network corresponds to the index of a job in the system's admission queue, which then gets selected for execution in the system. The time it takes for the agent to see another state also depends on the schedule as, when no preemption is used, the agent only needs to make decisions when jobs arrive, or finish execution (or when a user directly interacts with the system). Due to that, differently from other approaches in the literature, we model this problem as a Semi-Markov Decision Process (SMDP), instead of as a Markov Decision Process (MDP), as SMDPs allow for temporal abstraction of actions.

In this paper, we propose a novel study of the impact of uncertainty in job run time estimates, while also proposing a

[*] Corresponding author.Microsoft, Av. Presidente J. Kubitscheck, 1909, Torre Sul, 16° andar.
E-mail address: renatoc@ufmg.br (R.L. de Freitas Cunha).

https://doi.org/10.1016/j.future.2022.09.025
0167-739X/© 2022 Elsevier B.V. All rights reserved.

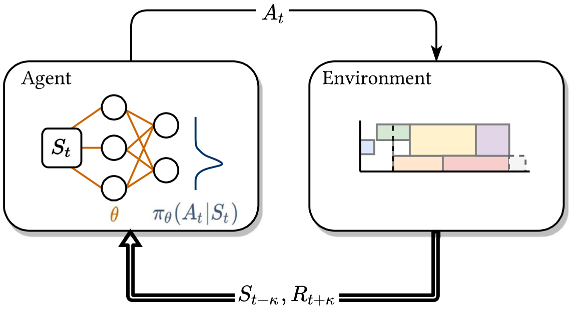


Fig. 1. Agent–environment loop. Unlike traditional modeling approaches in the literature, we model actions that take more than one time step to execute and, hence, taking action A_t on state S_t at time step t does not take the user to state S_{t+1}, but to state $S_{t+\kappa}$, with $\kappa \geq 1$.

model for learning how to schedule jobs to advance the state-of-the-art of ML-for-HPC [4]. In summary, our main contribution is a learning model based on the SMDP formalism that is able to schedule jobs minimizing average bounded slowdown (Section 4). To demonstrate the validity of this contribution, we performed the following supporting activities:

- Evaluation of the effects of noise in run time estimates in small (64 processors) and large (16384 processors) clusters alike (Sections 5.2–5.4);
- Evaluation of the model using traces from real supercomputing workloads, contrasting its performance with that of a model from the literature (Section 5.3).

The rest of this paper is structured as follows: in the next section, we present and discuss the related work. In Section 3, we discuss algorithms that enable RL in job scheduling, along with how to map scheduling to a learning setting and models that enable effective learning. In Section 4, we present how we used the algorithms discussed in the previous section to model scheduling, as well as justify our design decisions. In Section 5 we present our experiments and environment setup and, finally, in Section 6, we present a discussion and our concluding remarks.

2. Related work

The Deep Resource Management (DeepRM) [5] algorithm had great influence in the recent wave of applications of Deep Reinforcement Learning (RL) to the scheduling of computational jobs. DeepRM uses the Reinforce policy gradient algorithm with state baselines to schedule jobs based on their time, CPU, and memory requirements. DeepRM's state representation uses matrices to treat jobs as images, and using those images as input to choose which job to schedule next. DeepRM mainly optimizes for job slowdown, but both it and other methods are able to optimize for other metrics [6,3].

Domeniconi et al. [7] proposed CuSH, a system that built on DeepRM to schedule for CPUs and GPUs, but proposed a hierarchical agent. The agent works in two phases: in the first, it chooses the next job to send to the second phase, a policy network that chooses the scheduling policy to use to determine when the job will run. It is important to highlight a major difference between DeepRM and CuSH: whereas DeepRM learns the scheduling *policy* itself, CuSH is essentially a classifier, which chooses between two existing policies. This means that, even without training, CuSH's behavior is expected to be more stable than that of DeepRM, since DeepRM-style schedulers might get stuck in local minima, as reported by de Freitas Cunha and Chaimowicz [8], who investigated the behavior of DeepRM-style agents when trained with policy-gradients based RL algorithms such as Advantage Actor-Critic (A2C) and Proximal Policy Optimization (ppo), and proposed an OpenAI Gym environment for easier evaluation of RL agents for job scheduling.

Another agent that has been proposed recently, and that learns the scheduling policy itself, is rlscheduler [9]. rlscheduler is a ppo-based agent with a kernel-based design that looks at job features for scoring them. rlscheduler scores jobs in a window of up to 128 jobs over the queue. The major innovation in RLscheduler is in the training setting, in which the authors combine synthetic workload traces with real workload traces to present the agent with ever more difficult settings, similar to learning a curriculum of tasks. A potential deficiency with rlscheduler is that it only uses local job information for making decisions, without considering the cluster's occupancy state.

Similarly to CuSH, other agents that use a classification approach have been proposed. A recent one is the Deep Reinforcement agent for Scheduling in HPC (DRAS) [3], which classifies jobs in three categories: *ready*, *reserved*, and *backfilled*. After this classification step, the cluster scheduler takes the output of this classification and allocates jobs accordingly (for example, by reserving slots in the future for reserved jobs, scheduling immediately ready jobs, and finding "holes" in the schedule for backfilled jobs). DRAS uses a five-layer Convolutional Neural Network (CNN) that works in two levels, with the first level selecting jobs for immediate and reserved execution, and the second layer for backfilled execution.

Apart from HPC jobs, there have been approaches for scheduling workflows with reinforcement learning [10–13], using machine learning for deriving static scheduling policies for job scheduling [14], and optimization techniques [15]. Outside of reinforcement learning, many ML techniques have been used to aid in the scheduling of jobs, such as predicting resource and time requirements [1,16–20].

We are also starting to see promising results with SMDP models in other systems-related areas. For example, Lu et al. [13] model auto-scaling workflow services as an SMDP and show that their model can outperform both traditional, static threshold auto-scaling techniques, as well as an MDP-based, SARSA model, when tested on a three-node cluster subjected to traffic sampled from the 1998 World Cup Web site data [21], indicating a promising direction for ML and systems research.

From our point of view, few papers have recently reported the effects of decisions in modeling the scheduling HPC jobs for solving with RL while considering the impacts of uncertainty in job runtime estimates.

2.1. Differences from our previous work

The present paper expands on the results of two previous papers from our group. In the first one [8], we presented a discrete-event simulator that adheres to the OpenAI Gym [22] interface, and that was written following software engineering best practices. With that simulator, we were able to reproduce results from the literature using a small amount of code with a library of RL agents designed to work with OpenAI Gym [23].

In the second one [24], we proposed variations of MDP models for job scheduling, comparing the performance of agents trained in each of those environments, and analyzing how the design decisions of each MDP affected final agent performance.

In both cases, we assumed a rather simplistic workload model, machines of relatively small sizes, and we considered job run time estimates were perfectly accurate. In this paper, we revisit our model, formalizing the SMDP environment and providing a detailed description of the state representation and workload models used. Furthermore, we investigate the effects on agent

Table 1
Summary of the notation used in this article.

Symbol	Description
j_i	The *i*th job in the system
$t_f(j)$	The finish time of job *j*
$t_s(j)$	The submission time of job *j*
$t_e(j)$	The execution time of job *j*
$t_w(j)$	The wait time of job *j*
$t_r(j)$	The requested time of job *j*
$q(j)$	The queue size in front of job *j*
$q_w(j)$	The work in the queue in front of job *j*
$p_f(j)$	The number of free processors at *j*'s submission
$\mathcal{S}$	The set of states in an MDP
$\mathcal{A}$	The set of actions in an MDP
$\mathcal{R}$	The reward function in an MDP
$\mathcal{T}$	The transition function in an MDP
ρ	The set of initial states in an MDP
γ	Discount factor in an MDP
τ_i	A trajectory $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$ in episode i
S_t	The state seen by an agent at time-step t
A_t	The action taken by an agent at time-step t
R_t	The reward received by an agent at time-step t
θ	The parameters of an approximation function
π_θ	A policy with parameters θ
$\phi_\theta(\tau)$	The probability of following τ with policy π_θ
$\sigma(\mathbf{z})_i$	The *i*th element of the softmax of vector $\mathbf{z}$
$v_\pi(s)$	Value function for state s under policy π
$q_\pi(s,a)$	State–action value function for state s and action a under policy π
$q_\pi(s,\omega)$	State-option value function for state s and option ω under policy π

performance when run time estimates are inaccurate. We also analyze how agents fare with a realistic workload model, and in larger machines. Moreover, the state representation we propose can work with clusters of any size, taking into account both local information about jobs, and global information about the cluster.

3. Algorithms for reinforcement learning and scheduling

In this section, we use a scheduling example to introduce the key techniques used in the methods section (Section 4). To prevent the text from becoming too dry, we will interleave the mathematical definitions with a job scheduling example we will keep referring to. Table 1 summarizes the notation employed throughout this paper.

3.1. Batch job scheduling

A scheduler's main goal is to manage the job queue, coordinating execution of jobs such that they do not wait too long to execute, and the utilization of the system is high. At any given time, zero or more jobs may arrive for processing, and appended to an arrival queue. The job scheduler then allocates resources for job execution such that it satisfies the jobs' resource requirements. The simplest allocation strategy is First-Come First-Serve (FCFS), in which jobs are scheduled in order of arrival. Other heuristics have been proposed over the years, with the objective of increasing utilization and reducing wait time for jobs. Although some recent schedulers may oversubscribe resources, schedulers usually guarantee there will not be over-subscription of resources.[1] Given this primary goal, secondary goals vary between schedulers and HPC facilities, depending on whether the hosting institution prefers to satisfy the needs of individuals submitting jobs, or the whole group of users [25].

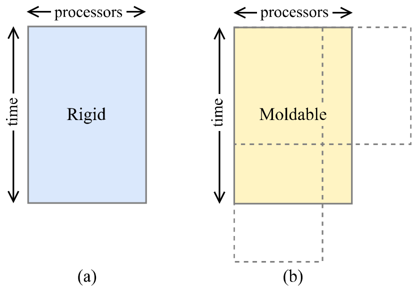


Fig. 2. Rigid and moldable job types: (a) shows rigid jobs, which define a fixed rectangle in processor × time space, while (b) shows moldable jobs, in which the scheduler may choose the number of processors from a set of possibilities.

In this paper, we will center our discussion on rigid and moldable jobs: in both cases, the number of processors required by a job are determined before a job enters the queue. Therefore, the number of processors required by the jobs we consider here will *not* change at run time. Hence, considering the number of processors required by a job in one axis, and the amount of time required by a job in another axis, the jobs considered here define a rectangle in the time × processors space, as exemplified by Fig. 2. To the area of the time × processors rectangle we give the name of *work*. For example, a job that requests 3 processors for 2 time steps is expected to take 6 units of work. In this paper, the actual amount of work taken by a job is only known after if finishes. Before that happens, our methods only used the *expected* amount of work, which serves as an upper bound for *actual* work: in our model, schedulers never give more time, nor more processors than those requested by a job.

When optimization of response time is a subgoal, it is usually modeled as the minimization of the average response time, with response time used as a synonym to turnaround time: the difference between the time a job was submitted to the time it *completed* execution. A metric commonly used to evaluate this is the *slowdown* of a job, which, for job *j* is defined as

$$\begin{aligned}\text{slowdown}(j) &= \frac{(t_f(j) - t_s(j))}{t_e(j)} = \frac{t_w(j) + t_e(j)}{t_e(j)} \\ &= \frac{1}{t_e(j)}\left(\sum_{i=1}^{t_w(j)} 1 + \sum_{i=1}^{t_e(j)} 1\right),\end{aligned} \tag{1}$$

where $t_s(j)$ is the time job *j* was submitted, $t_e(j)$ is the time it took to execute job *j*, and $t_f(j)$ is the finish time of job *j*. The equality in the middle holds because the wait time, t_w, of a job *j* is defined as $t_w(j) = t_f(j) - (t_e(j) + t_s(j))$. The form of slowdown presented in Eq. (1) is useful in our context because we can compute an approximation of slowdown online, as the job is still in the system. As soon as the job finishes execution, Eq. (1) converges to the actual slowdown.

Consider the case of three batch jobs, $j_1 =$ 🟩, $j_2 =$ 🟧, and $j_3 =$ 🟥, submitted to a scheduling system with two processors, and that the three jobs were submitted "between" time step 0 and 1, such that, when transitioning from the first time step to the second, now there are three jobs waiting. Also consider that, for these jobs, the generated schedule is the one displayed in Fig. 3. As shown in the figure, the jobs execute for two, three and four time steps respectively, and all of them use a single processor.

The reader should observe that different schedules can yield substantially different values of (average) slowdown. For example, the schedule shown in Fig. 3 has an average slowdown equal to $^1/_3 \sum_{i=1}^{3} \text{slowdown}(j_i) = {}^1/_3(1 + 1 + {}^3/_2) = {}^7/_6$, whereas, if we swapped j_3 with j_1, and started j_1 soon after j_2 finished, the slowdown would be $^1/_3(\frac{3+2}{2} + 1 + 1) = {}^9/_6 = {}^3/_2$, a ≈

[1] Some schedulers allow for over-subscription of memory resources in their default configuration, inspired by the fact that jobs do not use peak memory during their complete lifetimes.

29% increase. Therefore, a scheduler should choose job sequences wisely, otherwise its performance can be degraded.

In this paper, we will focus our discussion on what happens when a scheduling system based on RL tries to minimize the average slowdown.

An issue with using slowdown is that the metric is sensitive to small jobs: since small jobs will have smaller $t_e(j)$, any delays in scheduling small jobs will increase their slowdown. Humans do not tend to notice small delays in small jobs, and thus, an alternative metric that is not so sensitive to small jobs is the bounded slowdown, defined as

$$\text{bsld}(j) = \max\left(1, \frac{t_f(j) - t_s(j)}{\max(\epsilon, t_e(j))}\right), \tag{2}$$

where ϵ is a configurable parameter, set to the smallest time one wants to consider for jobs. In practice, and in this paper, ϵ is usually set to 10. For a study of how different values influence bounded slowdown, we direct the reader to Feitelson [26].

3.2. Deep reinforcement learning and job scheduling

In a Reinforcement Learning (RL) problem, an agent interacts with an unknown environment in which it attempts to optimize a reward signal by sequentially observing the environment's state and taking actions according to its perception. For each action, the agent receives a reward. Thus, in the end, we want to find the sequence of actions that maximizes the total reward, as we will detail in the next paragraphs.

RL formalizes the problem as a Markov Decision Process (MDP) represented by a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \rho, \gamma \rangle$.[2] At each discrete time step t the agent is in state $S_t \in \mathcal{S}$. From S_t, the agent takes an action $A_t \in \mathcal{A}$, receives reward R_{t+1} from $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and ends up in state $S_{t+1} \in \mathcal{S}$. Therefore, when we assume the first time step is 0, the interaction between agent and environment creates a sequence $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$ of states, actions and rewards. To a specific sequence $S_0, A_0, R_1, S_1, A_1, R_2, \ldots$ of states, actions, and rewards we give the name of trajectory, and will denote such sequences by τ. The transition from state S_t to S_{t+1} follows the probability distribution defined by $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ or, in an equivalent way, $\mathcal{T}$ gives the probability of reaching any new state s' when taking action a when in state s: $p(s'|s, a) = p(S_{t+1} = s'|S_t = s, A_t = a)$. ρ is a distribution of initial states, and γ is a parameter $0 \leq \gamma \leq 1$, called the discount rate. The discount rate models the present value of future rewards. For example, a reward received k steps in the future is worth only γ^k now. This discount factor is added due to the uncertainty in receiving rewards and is useful for modeling stochastic environments. In such cases, there is no guarantee an anticipated reward will actually be received and the discount rate models this uncertainty.

To map our presentation of RL into our problem of job scheduling, we consider $\rho($└──$) = 1$ (the only possible initial state is the empty cluster), with the first state consisting of the empty cluster, with no jobs in the system, $S_0 = \langle$└──$\rangle$ and $A_0 = \emptyset$, since there is no job to schedule. We also consider an episodic setting, with an episode consisting of the submission and scheduling of a set of jobs. For example, if we consider 256 jobs in an episode, the episode starts with the empty cluster, has 256 jobs submitted according to a workload model, and finishes once the 256-th job is scheduled. Every time an episode finishes, the state of the system is reverted to the empty cluster state.

Recall our discussion about classifying jobs to be processed by different policies, *versus* choosing the next job to enter the system. In this paper, we allow the agent to choose the next job, and so the agent is learning a scheduling policy. In our example, one can obtain a reward function by using the sequential version of slowdown, shown in the rightmost equality of (1), such that the reward at each time step is given by the sum of the current slowdown for all jobs in the system: $\mathcal{R} = -\sum_{j \in \mathcal{J}} {}^1\!/_{t_e(j)}$. When the reward function is such that it computes the online version of slowdown for *all jobs in the system*, if $A_1 = \emptyset$, $R_2 = {}^1\!/_2 + {}^1\!/_3 + {}^1\!/_4$. Moreover, if jobs j_1, j_2, and j_3 are chosen in sequence, the next state, shown in Fig. 3, will be given by sequentially applying the transition function $\mathcal{T}$ as $\mathcal{T}($└──, 🟩$)\mathcal{T}($└▭─, 🟧$)\mathcal{T}($└▤─, 🟥$)$. If the episode finished immediately after the state shown in Fig. 3, the trajectory τ_1 would be given by $\tau_1 = \langle S_0 =$ └──$, A_0 =$ 🟩$, R_1 = 0, S_1 =$ └▭─$, A_1 =$ 🟧$, R_2 = 0, S_2 =$ └▤─$, \ldots\rangle$.[3]

The reward signal encodes all of the agent's goals and purposes, and the agent's sole objective is to find a policy π_θ parameterized by θ that maximizes the expected return

$$\begin{aligned} G(\tau) &= R_1 + \gamma R_2 + \cdots + \gamma^{T-1} R_T \\ &= \sum_{t=0}^{T-1} \gamma^t \mathcal{R}(S_t, A_t \sim \pi_\theta(S_t)), \end{aligned} \tag{3}$$

which is the sum of discounted rewards encountered by the agent. When T is unbounded, $\gamma < 1$. Otherwise, Eq. (3) would diverge. $\pi_\theta(S_t)$ is a function that, given a state, returns an action for the agent to take. In our example, a deterministic policy that implemented the Shortest Job First (SJF) algorithm would yield $\pi_\theta(\langle$└──, 🟩, 🟧, 🟥$\rangle) =$ 🟩, while a stochastic policy would assign a probability to each job, and either choose the one with highest probability or sample from the jobs according to that distribution. In our example, for each job j_i in time step 1, π would give the probabilities of choosing each job given an empty cluster: such that, by total probability, $\pi($🟩$\|$└──$) + \pi($🟧$\|$└──$) + \pi($🟥$\|$└──$) = 1$. In practice, when neural networks are used for approximation, the last layer of the neural network is usually a softmax, or soft-argmax, function, defined as

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \tag{4}$$

for $1 \leq i \leq K$, and $\mathbf{z} = (z_1, \ldots, z_K) \in \mathbb{R}^K$, where $\mathbf{z}$ is the output vector of the neural network, and K is the number of classes to choose from (for example, when choosing jobs, K may represent the size of the vector with jobs to choose from). Since the softmax is normalized by the sum of the exponential of all individual components, the sum of the elements of $\sigma(\mathbf{z})$ equals 1, with each individual element $0 \leq \sigma(\mathbf{z})_i \leq 1$. This allows us to interpret the softmax as a probability mass function, and when K equals the number of actions available to an agent, each element of the softmax $\sigma(\mathbf{z})_i$ can be interpreted as the probability of taking the ith action available. Apart from neural networks [27,28], another popular type of function approximation are linear or polynomial combinations of features [29].

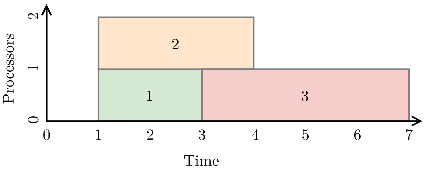


Fig. 3. A possible schedule when three jobs arrive in a scheduling system at discrete time step 1 and no more jobs are submitted to the system at least until time step 7, the last one shown in the figure.

[2] Some authors leave the γ component out of the definition of the MDP. Leaving it in the definition yields a more general formulation, since it allows one to model continuous (non-episodic) learning settings.

[3] The value shown for R_2 might contradict the previous discussion, but the MDP is set in a way that, *when jobs are scheduled successfully*, $R_{t+1} = 0$.

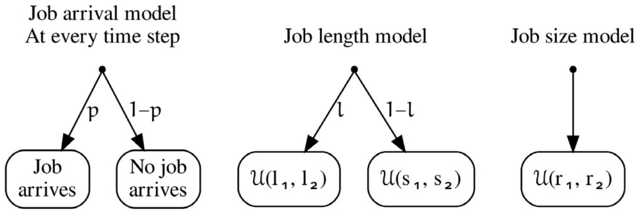


Fig. 4. Graphical representation of the workload model based on the one proposed by Mao et al. [5]. Job sizes, job arrivals, and job durations are sampled independently.

3.3. Workload models

So far, we have discussed the state transition function $\mathcal{T}$, but we still have not mentioned how jobs *arrive* in the system. This is the role fulfilled by workload models. These types of models not only determine the "shape" of jobs (how many resources they need, and how long they take to run), but also the time at which they arrive in the system. Here we consider two synthetic workload models: the Mao model [5] (Section 3.3.1), and the Lublin model [30] (Section 3.3.2).

3.3.1. The Mao model

In the Mao model, adapted from the model proposed by Mao et al. [5], job length, job arrival times and job sizes are sampled independently. The Mao model assumes that, for each simulation time step, a job has probability p of arriving in the system. If a job arrives, to determine its length, the workload model decides, with probability l, whether this is a long job, or a short job. Long jobs are sampled from $\mathcal{U}(l_1, l_2)$ and small jobs are sampled from $\mathcal{U}(s_1, s_2)$, where $\mathcal{U}(a, b)$ is the discrete uniform distribution between a and b, and $l_2 > s_2$. The job's size is sampled from $\mathcal{U}(r_1, r_2)$. Fig. 4 shows a graphical representation of this model.

Although this model is useful for testing smaller problems and for verifying whether an agent is learning, this is an unrealistic model which relies on time-based simulations due to the probability p of sampling a job depending on the current time step. A consequence of using this model is that it becomes very hard to tune, generating either too many jobs, or too few jobs, once the number of time steps in a simulation increases.

3.3.2. The Lublin model

In the Lublin model, job arrivals are sampled from two Gamma distributions, with one being used for the peak interarrival times, and another for the daily inter arrival times. Once a model is determined to arrive, modeling of the job itself follows the algorithm depicted in Fig. 5. The Lublin model is a hierarchical model that selects job sizes from a two-stage uniform distribution with four parameters. The minimum and maximum job sizes desired, and the fractions of serial (p_1 in the figure) and power-of-two (p_2 in the figure) jobs. After a job size is selected, job run times are sampled from a hyper-Gamma distribution ("Run time selection" in the figure) which depends on the job size sampled previously. Finally, arrivals to the system are modeled by two Gamma distributions, with one for the peak interarrival times, and another for the daily interarrivals cycle. Lublin and Feitelson [30] published the original workload model implementation as a C++ source file, and we use a Python wrapper for the C code from the `parallelworkloads` library.[4]

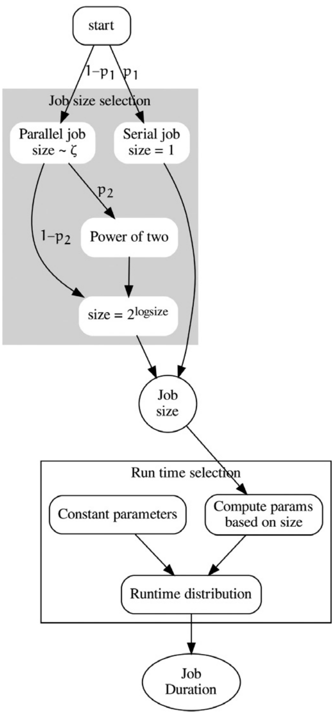


Fig. 5. Graphical representation of the workload model proposed by Lublin and Feitelson [30]. Job sizes are sampled from a distribution and then are fed to another distribution to sample job length. Outputs are the job size and job duration. These job characteristics are then combined with the arrival model to determine the job start time.

3.4. Uncertainty in job run time estimates

In our discussion so far, we assumed the run time of jobs are known by the scheduler. Various models in the literature [5,7,9] assume the availability of run time estimates, yet, real-world average job run time estimates, even when available, are inaccurate [31]. For this reason, even though some scheduling algorithms designed with accurate run time estimates in mind may perform well with noisy job run time estimates [32,33], others will suffer significant performance degradation [34]. Additionally, some systems use ML techniques to infer job resource and time requirements [16,1], and those predictions will not be perfect either. In this section, we present two models that can be used to generate noisy run time estimates from jobs sampled by the workload models discussed in the previous section.

3.4.1. The Gaussian model

In the Gaussian model, job run time estimates are sampled from a Gaussian distribution centered at the actual job run time, with a configurable standard deviation scaling parameter. The model supports three variations: one-sided overestimation, one-sided underestimation, and two-sided Gaussian. Usually, it does not make sense to generate consistently underestimated times, as jobs that exceed their run times are usually terminated by schedulers. Still, we wanted to observe the impact of such changes on learning algorithms, and left that option in the model. Formally, in this model, the estimate for job j is its actual execution time, $t_e(j)$ plus a difference, sampled from

$$\operatorname{diff}(j) = \begin{cases} -|\mathcal{N}(0, \beta(j))|, & \text{if underestimated} \\ |\mathcal{N}(0, \beta(j))|, & \text{if overestimated} \\ \mathcal{N}(0, \beta(j)), & \text{otherwise,} \end{cases} \tag{5}$$

[4] https://github.com/renatolfc/parallelworkloads.

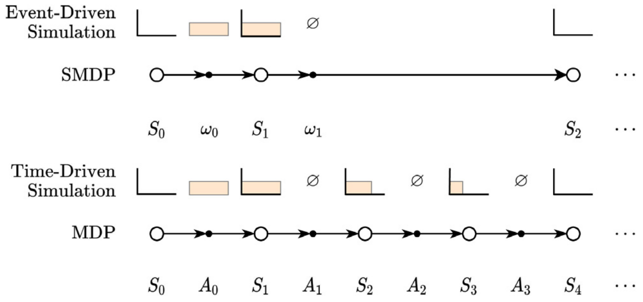


Fig. 6. Relationship between an MDP and a Semi-MDP, and how an MDP is tied to a time-driven simulation, whereas an event-driven simulation is closer to an SMDP. The illustration shows an empty cluster └─ where the agent chooses a job ▭ in the first state. In the MDP, the agent sees all time steps in which the job ▭ is processed, while in the SMDP, the agent only sees the job to select, the event for the job starting, and the event for the job finishing. In both cases, state S_1 shows the state of the cluster immediately after the job ▭ starts running. Also, in both cases, no other job arrives after ▭.

where $\mathcal{N}$ is the Gaussian distribution and $\beta(j) = \nu t_e(j)$ is a scaling parameter, controlled by parameter ν and the actual execution time of the model. For cases in which the run time estimate is smaller than 1 second, we set them to 1.

3.4.2. The Tsafrir model

User run time estimates tend to be modal due to users preferring to repeat time estimates. A more appropriate run time estimate model might be the one proposed by Tsafrir et al. [35], which was built with the modal nature of job run time estimates in mind, and which uses a histogram of the twenty most popular estimate values, as these tend to account for 90% of job run time estimates in production machines. In this paper, we used a Python wrapper from the `parallelworkloads` library, which wraps the original C++ code available from the parallel workloads archive.

3.5. Semi-MDPs

Both in state-driven simulations, and in actual job schedulers, job scheduling decisions do not necessarily take place at, say, *every second* or at fixed time intervals. Rather, scheduling decisions are made when external events happen, such as the arrival of a new job, the finishing of a running job, or a request from a user of the scheduler, such as changing the priority of a job, or requesting its termination, or when a timer expires.

In Section 3.2, we presented a way to model job scheduling as an MDP, but, when we consider the fact that decisions need only occur when events happen, the actions available to the agent are, in fact, *options*: temporal abstractions over actions, meaning that once a decision is made, the next state seen by the scheduler might not necessarily be for the next time step, but to the time of the next *event*. Moreover, any MDP with a decision process that only selects options over that MDP is an SMDP [see 36, Theorem 1]. Fig. 6 gives an intuition of the differences between an MDP and a SMDP. In the example in the figure, one agent sees five states (MDP), while the other only sees three (SMDP).

Formally, an option $\omega = \langle \pi, \gamma_\omega \rangle$, where π is a policy and γ_ω is a state-dependent termination function, is a generalization of actions, and can take more than one time-step to execute. To execute option ω at time t, an agent chooses the action $A_t \sim \pi(\omega|S_t)$, with ω terminating at time $t+1$ with probability $1 - \gamma_\omega(S_{t+1})$, at $t+2$ with probability $1 - \gamma_\omega(S_{t+2})$, and so on, until termination (of the option.)

In our event-driven context, then, the probability $\gamma_\omega(S_{t+k})$ is one for all time steps k between choosing a job for execution, and the time at which the first event happens after time step t. In other words, let u be the time step of the *first* event that happens after choosing action A_t. Then, $\gamma_\omega(S_u) = 0$ and the option terminates with probability 1 at time step u and $\gamma_\omega(S_w) = 1$ for all $t \neq u$. As exemplified by Fig. 6, in our model, when an option takes more than one time step to execute, intra-option actions are set to $\varnothing$. Since there are no decisions to be made, doing nothing is the only decision that makes sense. The advantage of using this model as opposed to an MDP is that the reinforcement signal needs only flow through actions that make a difference in agent performance, skipping "unneeded" actions, as in the bottom part of Fig. 6.

The reward when using an option model is given by

$$r(s, \omega) = \mathbb{E}\left[\sum_{i=0}^{\kappa-1} \gamma^i R_{i+1} \mid S_t = s, A \sim \pi, \kappa \sim \gamma_\omega\right]. \tag{6}$$

In other words, the reward of an option is the sum of discounted rewards for all rewards received at intermediate time steps while the option was being executed. It is important to note that γ_ω affects option termination, while γ is used for discounting.

In a similar fashion, the value of state s, $v(s)$, needs to take into account the multiple time-steps an option can take, and is defined as

$$v_\pi(s) = \sum_{\omega \in \Omega(s)} \pi(\omega|s)\left[r(s, \omega) + \sum_{s'} \mathcal{T}(s'|s, \omega) v_\pi(s')\right], \tag{7}$$

where $\mathcal{T}(s'|s, \omega) = \sum_{t=1}^{\infty} \gamma^t p(S_i = s', \kappa = t \mid S_t = s, A \sim \pi, \kappa \sim \gamma_\omega)$ is a discounted weighting of state, option pairs from (s, ω), and $\Omega(s)$ is the set of options available at state s. The value of executing option ω at state s is given by

$$q_\pi(s, \omega) = r(s, \omega) + \sum_{s'} \mathcal{T}(s'|s, \omega) v_\pi(s'). \tag{8}$$

Both value functions give the value of scheduling a job until the next event happens. With these two functions, we can also define a third function, which encodes the *advantage* of choosing option ω in state s over the average value of state s: $a_\pi(s, \omega) = q_\pi(s, \omega) - v_\pi(s)$. We will return to these functions in our discussion of how our model was implemented (Section 4.1).

In practice, what changes in the implementation from an action-based agent to an option-based agent is how the discount factor is handled. While in the state value and in the state–action value functions the γ term appears explicitly, when using options, discounts are made implicitly, within $r(s, \omega)$.

3.6. Policy gradients

In this section we present the main optimization method we use to find policies: policy gradients. As implied by the name, we compute gradients of policy approximations, and use them to find better parameters for those functions.

Formally, we generalize policies to define distributions over trajectories with

$$\phi_\theta(\tau) = \rho(S_0) \prod_t \pi_\theta(A_t|S_t) \underbrace{\mathcal{T}(S_{t+1}|S_t, A_t)}_{\text{Environment}}, \tag{9}$$

in which π_θ is being optimized by the agent, and ρ and $\mathcal{T}$ are provided by the environment. What (9) says is that we can assign probabilities to any trajectory, since we know the distribution of initial states ρ, and we know that the policy will assign probabilities to actions given states, and that, when such actions are taken, the environment will sample a new state for the agent. When we do so, we can define an optimization objective to find the optimal set of parameters

$$\theta^* = \arg\max_\theta J(\theta) = \arg\max_\theta \int_\tau G(\tau)\phi_\theta(\tau) d\tau, \tag{10}$$

Table 2
Job features in the state representation.

Feature	Symbol	Description
Submission time	$t_s(j)$	Time at which the job was submitted
Requested time	$t_r(j)$	Amount of time requested to execute the job
Requested processors	$r_p(j)$	Number of processors requested at job submission time
Queue size	$q_s(j)$	Number of jobs in the wait queue at job submission time
Queued work	$q_w(j)$	Amount of work that was in the queue at job submission time
Free processors	$f_p(j)$	Amount of free processors when the job was submitted
Can start now	$c(j)$	Whether job j fits the cluster to start at current time

where $J(\theta)$ is the performance measure given by the expected return of a trajectory, which can be approximated by a Monte Carlo estimate $\widehat{J(\theta)} = {}^1\!/_N \sum_i G(\tau_i)\phi_\theta(\tau_i)$.[5] If we construct $\widehat{J(\theta)}$ such that it is differentiable, we can approximate θ^* by gradient ascent in θ, such that $\theta_{j+1} \leftarrow \theta_j + \alpha\nabla\widehat{J(\theta)}$, with $\alpha > 0$.

When using a neural network, we can use an automatic differentiation system [37] to perform this approximation by sampling trajectories from an environment in which we use the neural network for decision-making, and perform back propagation from the returns of episodes [8].

3.7. Maskable PPO

Optimizing the policy gradient objective (10) is the main method we use for learning, but some improvements to (10) have been proposed over the years. One such improved method is the Proximal Policy Optimization (PPO) algorithm, whose main ideas are (I) limiting the updates of the policy to within a region near the current parameters, (II) using multiple agents for collecting trajectories, and (III) estimating the advantage function of each state to be used as a baseline for variance reduction [38,8]. For this last idea, PPO uses two networks: a *policy* network, for predicting the next action, and a *value* network, for estimating the advantage of a state.

Especially in early parts of training, the learned policy will generate actions that are "invalid". The definition of what is or is not a valid action might vary, but as an example, when using a multi-layer perceptron, the neural network requires a representation of fixed size. In such a case, if jobs were chosen from a window over the waiting queue, and the queue was smaller than the window, there would be a non-zero probability of the agent choosing a job that does not exist. An approach for solving this problem is masking out invalid actions (jobs that do not exist), and sampling from the set of valid actions. This becomes especially useful as the number of actions available increase, speeding up learning by not letting the agent waste time sampling useless actions. For this reason, instead of using PPO directly, we use a *maskable* PPO implementation, which applies a *state-dependent differentiable function* during calculation of action probabilities [6].

4. The SMDP RL environment

The focus of this work is on the development and evaluation of agents that are able to schedule jobs in clusters of varying sizes, without retraining, and to assess their robustness to uncertainty in job run time estimates. In this section we detail how we modeled job scheduling as an SMDP and the learning algorithm we used.

4.1. The reinforcement learning model

We begin by describing the reinforcement learning environment, with state representation, state transition function, and reward function, and discuss how we connect these components for learning.

4.1.1. State representation

In previous research [24], we outlined the state representation we were using, but, due to space limitations, we did not provide details on how to construct it. In this section, we provide a detailed description of the state representation, and how to construct it from simulator variables. The state representation we use, summarized in Fig. 7, has three components:

The current allocation state of the cluster with the amount of processors allocated and amount of processors free. This component also includes, up to a horizon size H, the time offset of the next H events, with the allocation state of the cluster immediately after those events happen.

Description of jobs in the waiting queue up to a number of jobs in a window of attention of size W. The features used in this component are shown in Table 2.

Summary statistics containing: the remaining work for running jobs, the next time at which there will be free processors in the cluster, with allocation state, the size of the backlog (number of jobs in the waiting queue outside of the window of attention), and the current size of the waiting queue.

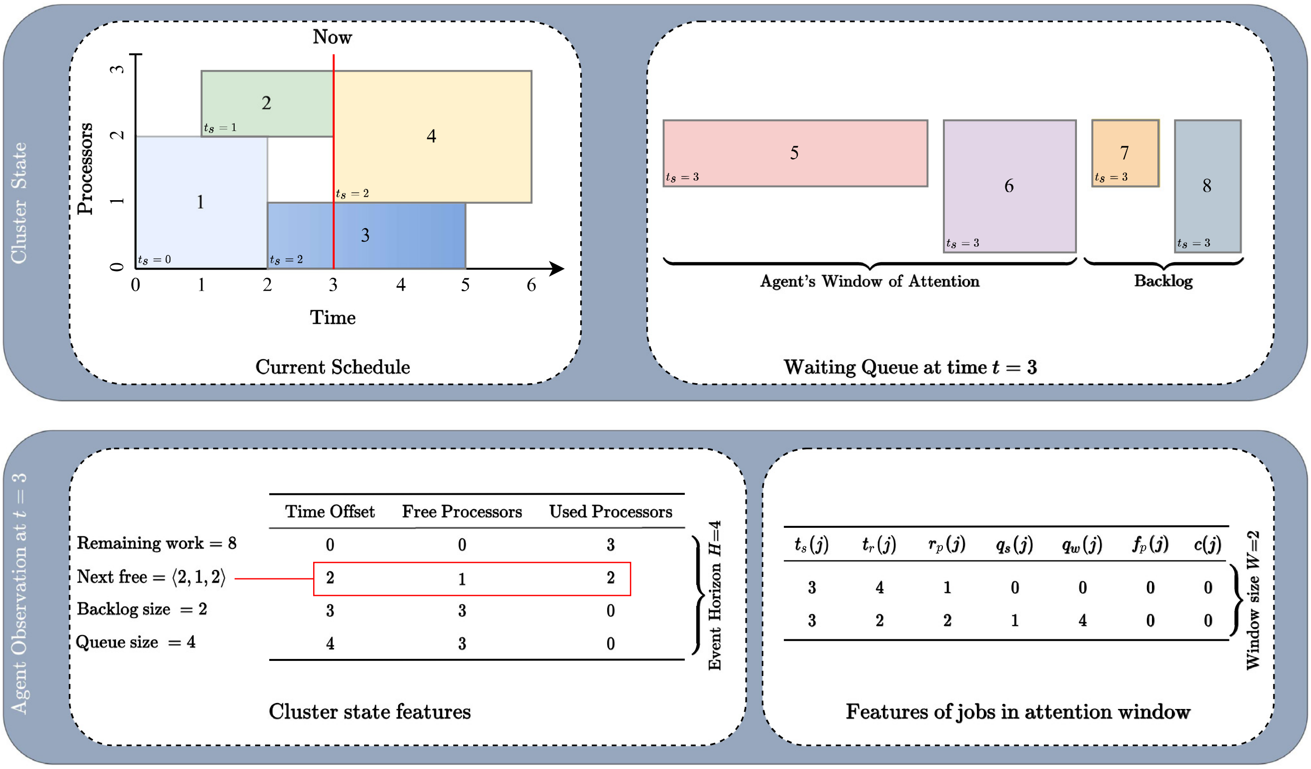
In the example shown in Fig. 7, the agent's window of attention W is 2 jobs, and the horizon of events H is set to 4 events.

For the cluster state, the first point to consider is that, since the agent is the scheduler, it has access to the events it expects to happen. For example, if at time step t the agent schedules a job that requests execution for 10 time-steps, assuming the job starts executing immediately, then the agent expects that, at time-step $t+10$, the resources used by that job will be freed. Hence, we are able to compute not only instantaneous features for the cluster, but also features of future states as expected by the agent. Even though the expected event times are not guaranteed to occur at the expected time (a job that requests 10 time steps can run for 5, for example), they help the agent in planning for the future. For each event in the horizon of size H, we represent the amount of processors in use, and the amount of free processors in the cluster. Additionally, we also represent the current size of the queue and the remaining work[6] in the cluster.

For the job features, we compute them considering the state of the cluster *when the job was submitted*. So, for example, the queue size feature for a particular job is not updated when jobs enter or leave the system. A summary of the job features is shown in Table 2.

[5] Normalization is needed to approximate the average value of $\widehat{J(\theta)}$. Otherwise, $\widehat{J(\theta)} \to \infty$ as $N \to \infty$.

[6] The summation of the remaining execution time times the number of processors requested by each job running in the cluster.



	Time Offset	Free Processors	Used Processors
Remaining work = 8	0	0	3
Next free = ⟨2, 1, 2⟩	2	1	2
Backlog size = 2	3	3	0
Queue size = 4	4	3	0

$t_s(j)$	$t_r(j)$	$r_p(j)$	$q_s(j)$	$q_w(j)$	$f_p(j)$	$c(j)$
3	4	1	0	0	0	0
3	2	2	1	4	0	0

Fig. 7. Example of state representation construction from the current schedule in the simulator at time step $t = 3$. As shown in the figure, there are no new events after time offset 3 ($t = 6$). Since the event horizon $H = 4$ is larger than the number of events the agent expects to see, the remaining events are filled by copying the last state the agent expects to see, with the time updated. The representation the agent sees is the concatenation of all values in the bottom row of the figure (Agent observation at $t = 3$), namely: $\langle \underbrace{8, 2, 1, 2, 2, 4, 0, 0, 3, 2, 1, 2, 3, 3, 0, 4, 3, 0}_{\text{Cluster state features}}, \underbrace{3, 4, 1, 0, 0, 0, 0, 3, 2, 2, 1, 4, 0, 0}_{\text{Features of jobs in attention window}} \rangle$.

4.1.2. State transition function

In this section, we discuss how we defined the state transition function $\mathcal{T}(s'|s, \omega)$. Recall from our discussion of SMDPs (Section 3.5) that we are using an event-driven simulation as our environment. In the discussion below, when we say time proceeds, we mean that the simulation proceeds normally, updating any queues and statistics without presenting intermediate states to the agent. The relevant state transitions are:

A new job arrives in a cluster with empty queues. In this case, when transitioning from state s to s', time advances from the current time t to the submission time of $t_s(j)$ of job j. Since queues were empty before the arrival of j, s' is now the representation of an empty cluster with a single job in queue.

The last running job finishes execution. If new jobs arrive, the next state s' is determined as in the previous case. Otherwise, if no more jobs arrive, this means the workload model has done generating jobs, and this is the end of the episode.

All cluster processors are in use. If there are jobs in the queue, time will proceed until the number of free processors in the cluster is such that at least one job j in the queue fits in the cluster, at which point the agent may decide to schedule j, or to schedule no jobs (this will happen, for example, if the agent decides to leave room in the cluster for short jobs, and there are no short jobs in the queue right now.) If there are *no* jobs in the queue, time will proceed until a new job arrives and at least one job in the queue fits in the cluster.

Some job in the queue fits the cluster. If the agent decides to schedule a job that does not fit the cluster, or if it chooses to not schedule any jobs, time proceeds by one time step, with the state updated accordingly. If this time coincides with the arrival of a new set of jobs, those jobs enter the queue before the generation of the new state s'.

Jobs arrive in the system following either one of the workload models present in `sched-rl-gym`, or following a trace of an existing machine.

4.1.3. Reward function

As discussed in Section 3.1, we use the negative slowdown of jobs in the system as input to the reward function. But, differently from previous research [8,24], instead of computing the slowdown of all jobs in the system, we compute the reward function using only the jobs in the agent's window of attention W. If, on a given time step a job is scheduled successfully by an agent, the reward is zero. Using (1), the reward for taking an action in a given state is

$$\mathcal{R}(s, a) = \begin{cases} 0, & \text{if successful} \\ -\sum_{j \in W} \frac{1}{t_e(j)} \left(\sum_{i=1}^{t_w(j)} + \sum_{i=1}^{t_e(j)} \right), & \text{otherwise,} \end{cases} \tag{11}$$

where by successful we mean that the agent picked a job that fits the cluster, and the job can begin running.

Notice that, when the selected job fills the cluster or is the only job in the queue, if such an action was taken at time step t, the next time step seen by the agent will *not* be $t + 1$ (Section 4.1.3). In such case, the reward for the action will be computed after termination of the option, with proper discounts for intermediate time steps, following (6).

In previous work [24], we had argued that tweaking the reward function would improve learning, but our experiments had failed to show any significant gain, mostly due to the short length of the simulations in our experiments. In Fig. 8, we show the difference in mean episode length when using all jobs in the system

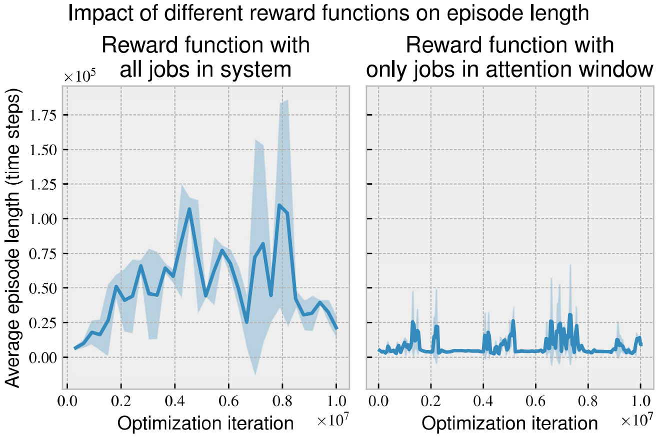


Fig. 8. Difference in episode length given reward computation method, with all parameters being equal (lower is better). On the left, we see the average episode length when using all jobs in the system to compute reward, whereas, on the right, we only consider jobs which the agent can affect. The curves were computed using the average of six parallel agents, with the shaded area representing one standard deviation. Both agents were trained in clusters with 128 processors and with the Lublin [30] workload generator generating 512 jobs per episode.

to compute rewards, versus using only the jobs upon which the agent can act, when using longer simulations (weeks, as opposed to seconds). As can be seen in the figure, computing rewards using all jobs makes learning unstable, whereas using a reduced set of jobs constrains learning to a region in which it trades off performance in the task, with entropy in its action distribution. Moreover, the worst training "performance"[7] achieved by our agent is better than the best performance of the method found in the literature [5,8].

4.1.4. Learning algorithms and implementation

At first sight, the main difficulty with using an SMDP model as we do here is that there is a shortage of open-source learning algorithms and environments designed to work with SMDPs. For the experiments in paper, we adapted the `sched-rl-gym`[8] OpenAI Gym environment to support an SMDP formulation by implementing equation (6) as reward function following the definition from Section 4.1.3, Eq. (11). As for the learning agent, we used the Maskable PPO [6,38] implementation of Stable Baselines 3 [23], but to prevent "double" discounting in the policy gradient, we set the PPO γ parameter to 1, while leaving equation (6)'s γ set to 0.99 to perform discounting. The reason for such a change is that the value function for state s, $v_\pi(s)$, in Eq. (7) does not have an explicit discount factor γ, whereas in the definition of the value function for state s' in an MDP there is an explicit discount factor.

PPO is an actor–critic algorithm (Section 3.6) that, as the name implies, has two components: an actor, which implements policy $A_t \sim \pi_\theta$, and a critic, which estimates the advantage function of taking action A_t at state S_t.

The variables represented in the state of our environment have different ranges and orders of magnitude. To reduce the likelihood of our policy network having too large weights, it might help to normalize the inputs to the neural network. We normalize features by dividing them by their maximum possible value. For time-related features and for features that depend on time (such as work), we log-transform the data prior to normalization, a practice common in workload modeling [30].

To mask actions, we have extended the OpenAI Gym environment to return, for each state, a boolean mask indicating which actions are valid for that state. That way, maskable PPO is able to compute actions correctly when learning.

4.2. Workload models

The `sched-rl-gym` environment supports three different types of workloads: jobs generated by the Mao model (Section 3.3.1), jobs generated by the Lublin model (Section 3.3.2), and jobs loaded from workload traces in the Standard Workload Format (SWF). The environment also supports generating uncertain estimates with both the Gaussian (Section 3.4.1) and Tsafrir (Section 3.4.2) models. The workload to use in a given instance of the environment is passed as an argument to OpenAI Gym's environment construction function.

For the Lublin model, `sched-rl-gym` will generate parallel jobs with sizes ranging from 2 processors to up to the size of the cluster following a two-stage uniform distribution, and the run time of jobs depend on the number of nodes in the cluster, with that number feeding a parameter of the hyper-Gamma distribution from which job durations are sampled.

5. Evaluation

5.1. Environment setup and metrics

In all our experiments, we used the `smdp` branch of the `sched-rl-gym` package, with the `CompactRM-v0` OpenAI Gym environment. We performed three sets of experiments in increasing order of complexity, described in the next sections. Except where explicitly mentioned, we measured final model performance using the average slowdown (1) of the simulation.

5.2. Mao workload model with Gaussian uncertainty

Before performing a complex simulation, it may be worth evaluating a model in a more controllable setting. For this reason, our first experiment builds on a simple problem with the Mao (Section 3.3.1) workload model with Gaussian noise in job run time estimates. In this set of experiments, we considered two different environments, called "long" and "short", with configuration displayed on Table 3. The reader will notice that simulations run for a short time in this scenario. This is due to the fact that, in the Mao workload model, jobs may arrive at every time step, given some base probability. In this model, it is hard to find a set of parameters that do not deliver an overly full, or an empty, cluster. Moreover, the reader will notice that times in the long environment are 10× larger than those of the short environment, with probability p of sampling jobs 10× smaller.

For both environments, we varied the ν factor of the Gaussian model (Section 3.3.2) from 0 (perfect estimates) to 2 (two standard deviations for noise), with both the overestimated and underestimated Gaussian noise models. For this experiment, to enable easier comparison with a model from the literature, we used the same learning parameters used by de Freitas Cunha and Chaimowicz [24] with *no* action masking, so that Maskable PPO works exactly as PPO. Also, for this set of experiments, instead of constructing the agent observation with future events it expected to happen, we constructed it with a time horizon, to be compatible with the work by de Freitas Cunha and Chaimowicz [24].

Although unrealistic in practice, for this experiment, we decided to evaluate the impact on the schedule if under-estimated

[7] Episode length is not a performance measure, but in this environment, it acts as a proxy for the quality of the generated schedules: worse agents will tend to see more states, increasing episode length.

[8] https://github.com/renatolfc/sched-rl-gym.

Table 3
Environment, workload model, and learning parameters used in the Mao-Gaussian set of experiments.

Parameter	Value	
	Short	Long
Time limit	100	1000
Job rate (p)	0.3	0.03
Long job lower bound (l_1)	32	320
Long job upper bound (l_2)	48	480
Short job lower bound (s_1)	1	1
Short job upper bound (s_2)	9	90
Smallest job size (r_1)	16	
Largest job size (r_2)	32	
Backlog size	60	
Window of attention	10	
Time horizon	20	
Long job chance (l)	0.2	
Surrogate epochs	10	
n steps	50	
batch size	64	
Clipping ϵ	0.2	
Value coefficient	0.5	
GAE λ	0.95	
Discount factor γ	0.99	
Entropy coefficient	10^{-2}	
Learning rate	10^{-4}	
Total iterations	3×10^6	

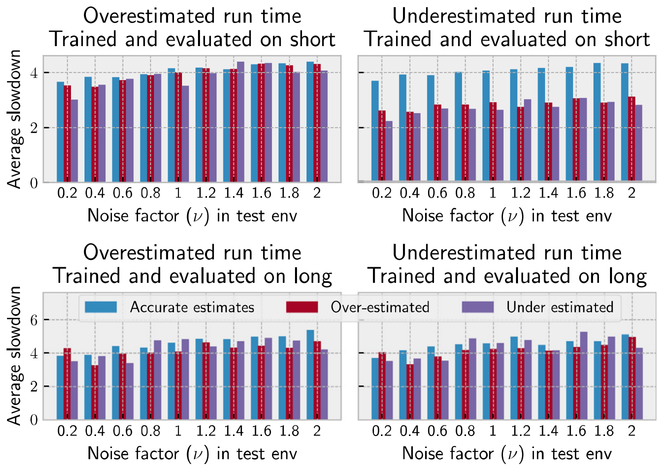


Fig. 9. Performance of the Mao workload model when using different noise factors in the Gaussian uncertainty model.

jobs were not terminated by the scheduler: run time estimates were merely used as "hints" to guide scheduling decisions. So, for each environment, we compared the performance of a baseline model, trained with accurate estimates, with models that used the overestimated and underestimated run time estimates.

The results are presented in Fig. 9. In the figure, we see that, as expected, as the noise factor ν increases, scheduling performance decreases (average slowdown increases). Also as expected, the models trained with noisy estimates (red and purple bars) tend to outperform the baseline model (blue bars). The models trained with underestimated noisy estimates outperform others in the short environment, whereas, in the long environment, we do not see any strong trends.

In the overestimated environments, there will be more "holes" in the schedule, due to the excess time between actual and estimated run times. This should make these scenarios easier than the underestimated scenarios without job termination when time was exceeded, due to the more "packed" nature of the schedule, giving less room for error for optimization of the learning algorithm. We believe this is the main reason for better performance in the short environment. In the long environment, due to the lower probability of arrivals, the cluster tends to be more free, reducing the difference between algorithms.

Table 4
Environment, workload model, and learning parameters used in the experiments that used the Lublin workload model.

Surrogate epochs	10
n steps	50
batch size	64
Clipping ϵ	0.2
Value coefficient	0.5
GAE λ	0.95
Discount factor γ	0.99
Entropy coefficient	10^{-4}
Learning rate	Linear decay $\left[3 \times 10^{-4}, 10^{-5}\right]$
Total iterations	10^6
Event horizon size	60
Window of attention size	128

Table 5
Comparison of the average bounded slowdown achieved by our SMDP model versus the ones reported by RLscheduler [9]. For the SMDP model, we report the average bounded slowdown and standard deviation after the ± symbol. Lower bounded slowdown is better. In the table, "RLS" stands for the base RLscheduler model, whereas "RLSB" stands for the RLscheduler model with backfilling. In this experiment, all SMDP models were trained with the Lublin workload model. Values in the Trace column describe the trace file used for evaluation.

Trace	RLS	RLSB	SMDP (Ours)
Lublin-1	254.67	58.64	67.55 ± 20.21
Lublin-2	724.51	118.79	225.58 ± 124.46
HPC2N	117.01	86.14	59.77 ± 87.45
SDSC-SP2	466.44	397.82	121.62 ± 137.81

5.3. Comparison with models from the literature

One question that might arise is how the model proposed here compares with ones in the literature. To answer that question, we trained a model using the Lublin workload generator for a 256 processor cluster. We then evaluated that model using the same methodology proposed by Zhang et al. [9]: 10 independent evaluations of random samples of job traces with 1024 jobs in each dataset. To ensure fairness in the comparison, we downloaded and replicated the sampling code of RLscheduler, abbreviated to RLS in the following discussion, by Zhang et al. [9], such that we used the same offsets in the jobs traces as they used in their evaluation.

All the models in this and following sections were trained in an episodic manner, with an episode ending every time 256 jobs were successfully scheduled. Between episodes, the system state was reset to the empty cluster ($\rho = \{\sqcup\}$). All sets of 256 jobs were generated by the Lublin workload model, with different random seeds, such that each episode is different from the other. The agents were trained with Maskable PPO for a million events, which translates to training for little more than one hour for each setting on a Core i7-8700K Desktop running Arch Linux with kernel version 5.16.2, with the *performance* CPU frequency governor and an NVIDIA GeForce GTX1070 GPU. For training, we used Maskable PPO with learning and environment parameters from Table 4. In these experiments, we trained a two-layer neural network with 256 units in the first layer and 128 in the second one, with no parameter sharing between value and policy networks, and the rectified linear unit activation function.

Table 5 shows a comparison of average bounded slowdown achieved by our model and compared with RLscheduler. For our

Table 6
Comparison of the average resource utilization achieved by our SMDP model versus the ones reported by RLscheduler [9]. For the SMDP model, we report the average utilization and standard deviation after the ± symbol. Higher utilization is better. In the table, "RLS" stands for the base RLscheduler model, whereas "RLSB" stands for the RLscheduler model with backfilling. In this experiment, all SMDP models were trained with the Lublin workload model and trained to optimize for slowdown. Values in the Trace column describe the trace file used for evaluation.

Trace	RLS	RLSB	SMDP (Ours)
Lublin-1	0.714	0.850	0.795 ± 0.025
Lublin-2	0.562	0.593	0.647 ± 0.078
HPC2N	0.640	0.642	0.593 ± 0.163
SDSC-SP2	0.671	0.707	0.770 ± 0.061

Table 7
Workload traces used in the evaluation with real traces. The trace column represents the name of the log in the Parallel Workloads Archive, with the version column representing the version used. The processors column corresponds to the number of processors in the machine, while the model column tells the number of processors that were used in training our models.

Trace	Version	Processors	Model
ANL-Intrepid [39]	1	163840	16384
CTC-SP2 [40]	3.1-cln	338	256
HPC2N	2.2-cln	240	256
SDSC-SP2	4.2-cln	128	128
SDSC-BLUE	4.2-cln	1152	2048

model, we report both the average value, and standard deviation. For the other values, we only report the mean, as we only reproduce what was reported by Zhang et al. [9]. We also evaluated the average resource utilization for the same datasets, and report the results in Table 6. The reader should be aware that while we compare performance in the four datasets of Tables 5–6, our model had never seen any of that data before evaluation.

From the data, we see that our SMDP model outperforms the base RLscheduler model (RLS) in most cases, while not necessarily doing so against the backfilling variant of RLscheduler (RLSB). We attribute this from the fact that despite our model not using backfilling explicitly, the use of an *event horizon* with events the agent expects to happen allows it to perform better planning ahead. Backfilling benefits RLscheduler because it uses a kernel to compute the priority of individual jobs in the queue, making local decisions, whereas backfilling gives a more global view of the schedule to the algorithm.

In the cases where our model has worse performance than the RLSB model (Lublin-1 & Lublin-2), we performed a t-test to check whether of our average bounded slowdown observations when considering the RLSB values as a population mean. Hence, the null hypothesis is that our distributions are the same, whereas the alternative hypothesis is that they are not. For the Lublin-1 case, we fail to reject the null hypothesis, but in the Lublin-2 case, we do reject it (p-value < 0.05). For the other two datasets (HPC2N and SDSC-SP2), we reject the null hypothesis (p-value < 0.05).

A remarkable difference between the SMDP model and RLS is on how performance of models trained on a given trace degraded when applied to another trace. For example, when RLS was trained on datasets other than Lublin-1 [see 9, Table VII], its performance degraded by ≈ 89% (SDSC-SP2), 11% (HPC2N), and 31% (Lublin-2). We attribute this difference to the local focus of the kernel layer RLS uses, which contrasts with the more global approach of the SMDP model.

In the original HPC2N and SDSC-SP2 traces, average bounded slowdown for the same jobs we used in our evaluation are 156.12 ± 173.42 and 202.65 ± 215.75 respectively, and due to the high variance of the average bounded slowdowns of the original schedules, when using Welch's t-test, we fail to reject the null hypothesis that the average bounded slowdown of the schedules of the learned agent *versus* those of the original traces have different means. Still, the schedule generated by a learning agent has much lower variance than the original schedule, making it a better choice for a more stable behavior. Concerning the utilization results in Table 6, they are interesting because our SMDP model was trained to minimize bounded slowdown, not utilization, and, despite that, it achieves good performance when compared against models that were explicitly trained to maximize utilization, outperforming them in 3 out of 4 cases.

Another difference in the data is that both in the Lublin-1 and Lublin-2 datasets, run time estimates are missing. What both RLscheduler and our model do is assume accurate run time estimates, while in the HPC2N and SDSC-SP2 datasets, actual user run time estimates are present. This should tend to make models that rely more heavily on accurate estimates to have better performance in the Lublin-1 and Lublin-2 datasets, and worse performance in the datasets with actual user run time estimates. Given that our model uses global cluster state features as well as job features, this would explain why RLscheduler performs better in the Lublin-1 and Lublin-2 datasets: RLscheduler probably gives more weight to the user run time estimates, having its performance degraded as the quality of estimates degrade.

5.4. Effects of noisy estimates with synthetic workload models

The models from the previous section were trained with accurate run time estimates, and evaluated with both accurate estimates (Lublin-1 and Lublin-2), and actual user run time estimates (HPC2N and SDSC-SP2). Now that we have established in the previous section that the performance of our SMDP model is at least comparable with, and arguably better than, a state-of-the-art model, we evaluate how the SMDP model fares under different synthetic run time estimate models. We trained models in clusters of 128, 256, 512, 1024, 2048, and 16384 processors with accurate run time estimates, with the Gaussian model (with ν varying from 0.5 to 2.0), and with the Tsafrir run time estimate model. Table 8 summarizes the results of the evaluation of the aforementioned models under the same settings. From the values shown, we do not see a trend in any direction, and the average bounded slowdown obtained with models trained with inaccurate user run time estimates are not different from the models with accurate run time estimates in a significant manner. This is different from what we expected, as we expected that models trained with uncertain run time estimates would perform better than models trained on accurate estimates. Perhaps, due to the way we build the state representation, the model does not rely that much on the run time estimate feature itself. In hindsight, although adding Gaussian noise may help in situations with small amounts of data, due to our use of a simulator and the Gaussian noise having zero mean, it might have been the case that the effects of the noise averaged out, not affecting much the performance of the system.

5.4.1. Evaluation with workload traces

Given the performance of the models when trained with uncertain estimates, we now evaluate what happens when we evaluate our models under real production workloads. For doing so, we selected a set of different job traces of clusters with varied numbers of processors. All datasets we used were downloaded from the Parallel Workloads Archive, and are summarized in Table 7. In the table we also show the number of processors we used to train our models when mapping to each workload trace. From the table, for example, we see that the same model was used in the CTC-SP2 workload and in the HPC2N workload, since we used the 256-processor model with both workload traces.

Table 8
Average bounded slowdown of models evaluated with the Lublin workload model and with run time estimates generated by the Tsafrir model. After performing a t-test between the models trained with accurate estimates and the models trained with inaccurate estimates, we did not find any statistically-significant (p-value < 0.05) differences between models. All models were evaluated with the same random seeds and we report the average and standard deviation of 10 independent evaluations.

	Processors					
Uncertainty model	128	256	512	1024	2048	16384
Accurate estimates	17.9 ± 19.0	22.8 ± 9.48	57.1 ± 43.7	20.7 ± 30.6	33.5 ± 47.0	32.7 ± 35.9
$\nu = 0.5$	18.4 ± 19.7	23.2 ± 9.35	57.9 ± 44.5	36.0 ± 75.6	24.8 ± 33.1	32.8 ± 34.4
$\nu = 1.0$	16.0 ± 18.4	25.3 ± 13.5	60.9 ± 46.8	20.5 ± 31.0	24.5 ± 24.1	32.4 ± 33.8
$\nu = 1.5$	14.5 ± 18.1	22.4 ± 7.82	57.5 ± 43.1	19.1 ± 29.5	34.4 ± 47.4	31.8 ± 35.2
$\nu = 2.0$	16.4 ± 18.5	21.6 ± 6.73	59.6 ± 46.6	35.1 ± 73.7	25.2 ± 24.5	33.4 ± 34.5
Tsafrir	14.6 ± 17.8	20.8 ± 6.83	59.0 ± 45.3	35.8 ± 75.6	27.1 ± 29.5	33.4 ± 33.6

Table 9
Average bounded slowdown for the trace files used in this paper. Underlined entries represent models that outperform the base model trained on accurate estimates.

Processors	128	256		2048	16384
	SDSC-SP2	CTC-SP2	HPC2N	SDSC-BLUE	ANL-Intrepid
Accurate estimates	247.81 ± 165.76	7.22 ± 4.15	43.53 ± 65.02	22.35 ± 16.90	1.26 ± 0.31
$\nu = 0.5$	236.77 ± 168.06	6.49 ± 3.85	46.58 ± 68.42	20.60 ± 16.92	1.30 ± 0.38
$\nu = 1.0$	230.26 ± 158.46	6.89 ± 4.80	43.41 ± 60.45	21.81 ± 17.79	1.27 ± 0.27
$\nu = 1.5$	244.10 ± 169.84	6.67 ± 3.84	44.74 ± 67.47	21.40 ± 17.39	1.23 ± 0.26
$\nu = 2.0$	225.03 ± 164.49	7.34 ± 4.81	46.32 ± 59.16	21.42 ± 16.68	1.31 ± 0.34
Tsafrir	243.30 ± 143.33	6.76 ± 4.15	39.30 ± 58.26	21.11 ± 15.75	1.34 ± 0.45
Shortest job first	240.10 ± 107.08	60.27 ± 103.52	55.63 ± 62.61	67.99 ± 99.30	1.78 ± 0.77
Packing heuristic	305.04 ± 221.95	6.98 ± 2.93	42.58 ± 59.07	23.12 ± 18.96	3.84 ± 6.17
Actual schedule	271.53 ± 201.28	86.05 ± 104.36	161.90 ± 184.35	242.618 ± 171.508	16.67 ± 16.07

We computed both the average slowdown obtained by each model, shown in Table 9, and the cluster utilization, shown in Table 10. In the tables, we included not only the metrics for our models, but we also computed the average bounded slowdown and utilization achieved by the original scheduler of the logs, and also the metrics of a scheduler that uses the SJF heuristic, and the packing heuristic proposed by Grandl et al. [41]. Including the real scheduler performance serves as a qualitative comparison, since, differently from our evaluation scenario, the real clusters were not empty when the evaluation jobs arrived in the system. This is especially salient in the CTC-SP2 cluster, which differ by one order of magnitude between the simulation and actual settings. Due to that, we also included the performance of SJF (which minimizes slowdown with accurate estimates), and the packing heuristic, which tends to increase resource utilization. We see that for most simulation scenarios, the variance of the average bounded slowdown metric is on the same order of magnitude of the metric itself, making it hard to differentiate between models in a statistically-significant way.

In Table 9, we underlined models trained with noise that outperformed the corresponding model trained with accurate estimates. In general, although with not too large a difference, at least one model trained with inaccurate estimates outperformed the same model when trained with accurate estimates. What is interesting in the metrics shown is that the learned model is more stable than the heuristics compared. For example, in the SDSC-BLUE case, the average bounded slowdown is ≈ 3× larger than that of the learned models and even when the learning model trained with accurate estimates is outperformed by a heuristic, the maximum difference (SDSC-SP2, SJF) is on the order of ≈ 3%. It is also somewhat surprising that the packing heuristic has worse performance than the actual schedule in the SDSC-SP2 trace.

Observing the results from Table 10, we see that utilization with the learning models is, for most workload traces, larger than those of the heuristics (although not statistically-significant), giving us confidence that even when optimizing for one metric (average bounded slowdown), our model is still competitive when we consider other metrics (utilization). In the table, we see that, for some systems, actual utilization is much lower than the computed utilization using the algorithms implemented in this paper. We believe this happens due to differences in number of queues (in our simulations, we used a single incoming queue, whereas the actual systems may have multiple queues with different restrictions), and jobs submitted to different partitions of the clusters.

6. Discussion & conclusion

In this paper, we proposed a way to model job scheduling as an SMDP with a recently-introduced RL algorithm (Maskable PPO). We showed, through our experiments, that our proposed model works with off-the-shelf implementations, by way of being compatible with the OpenAI Gym environment, and that it is competitive with other learning algorithms and heuristics, while training in a short time. More importantly, we were able to train good models with a third of training iterations of our previous work [24] while observing the effects of uncertainty in user run time estimates, a factor that is often overlooked with reinforcement learning agents for resource management, and we have shown these models are competitive when evaluated with real workload traces.

We showed that while training with noisy run time estimates improved the model in simpler settings, doing so failed to provide statistically-significant improvements both with synthetic, but realistic, workload models and with workload traces from real systems. Still, we showed that models trained with accurate run time estimates perform well even with noisy estimates from real workload traces, which we attribute to the model being able to "reason" over possible future cluster states at each decision point. Additionally, we showed our approach of training with realistic workload models outperforms models from the literature even on unseen data. Moreover, our models are competitive when transferred from one workload trace to another, even outperforming models from the literature that were trained in the target environment. Overall, our experiments suggest that training with

Table 10
Utilization for the trace files used in this paper.

Processors	128	256		2048	16384
	SDSC-SP2	CTC-SP2	HPC2N	SDSC-BLUE	ANL-Intrepid
Accurate estimates	0.79 ± 0.036	0.75 ± 0.081	0.61 ± 0.140	0.69 ± 0.076	0.71 ± 0.15
$\nu = 0.5$	0.79 ± 0.034	0.75 ± 0.080	0.61 ± 0.140	0.68 ± 0.074	0.71 ± 0.15
$\nu = 1.0$	0.79 ± 0.038	0.75 ± 0.080	0.61 ± 0.140	0.68 ± 0.074	0.71 ± 0.15
$\nu = 1.5$	0.78 ± 0.038	0.75 ± 0.080	0.61 ± 0.140	0.68 ± 0.074	0.71 ± 0.15
$\nu = 2.0$	0.79 ± 0.039	0.75 ± 0.082	0.61 ± 0.140	0.68 ± 0.074	0.71 ± 0.15
Tsafrir	0.79 ± 0.040	0.75 ± 0.080	0.61 ± 0.140	0.69 ± 0.072	0.71 ± 0.16
Shortest job first	0.74 ± 0.055	0.75 ± 0.031	0.54 ± 0.099	0.64 ± 0.092	0.73 ± 0.14
Packing heuristic	0.77 ± 0.054	0.77 ± 0.033	0.55 ± 0.100	0.66 ± 0.091	0.69 ± 0.13
Actual schedule	0.76 ± 0.082	0.58 ± 0.093	0.41 ± 0.147	0.41 ± 0.107	0.38 ± 0.07

synthetic workload models yields models that can generalize well to unseen situations.

One limitation of this study is that, for the synthetic workloads we generate, we do not model whether users, or the applications they execute, have consistent run time estimate biases. We do not think identifying the characteristics of user applications, or learning a user model belong, necessarily, in the RL agent, and we think the scheduling system as a whole could implement these models, and provide this information to the RL agent, perhaps as an embedding layer to the agent's policy and value networks. Due to that, we do not model specific user behavior. We do not think that is a problem though, as run time estimates in HPC clusters tend to gravitate around a set of twenty most popular estimates [35].

Given the above, we did not model user types in this study as we were focused on the RL model, nor did we think it was appropriate to introduce such a feature, although other models in the literature use it. Moreover, without an appropriate user model, simply adding a user identifier to the job state representation would probably work if jobs were submitted by a fixed set of users. Still, such an agent would fail to generalize once a new user type was introduced, reducing its usefulness in practice.

As presented, the way we model observations make the problem we try to solve a Partially-Observable Markov Decision Process (POMDP), not an MDP. We believe that, similarly to the Atari case, in which the approach of stacking frames prior to providing them to the agent effectively turns the problem from a POMDP to an MDP [42], adding the event horizon to our agent, along with a window of attention of 128 jobs, also brings the problem closer to an MDP. To overcome the limitation of using the backlog in the state representation, it might be possible to integrate an attention layer [43] to act upon the complete queue state of the system, allowing the agent to select any job for running, and not only jobs in the window of attention. Such a modification is left for future work.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The source code used in this research is publicly available at https://github.com/renatolfc/sched-rl-gym/.

Acknowledgments

The workload logs we used were downloaded from the Parallel Workloads Archive, which is maintained by Dror Feitelson. The original HPC2N, SDSC-SP2 workload logs were provided by Ake Sandgren, and Victor Hazlewood, respectively, while the SDSC-BLUE workload log was provided by Travis Earheart and Nancy Wilkins-Diehr. Dror Feitelson worked with the providers of logs to add background information, and perform interpretation, to the archive.

References

[1] R.L.F. Cunha, E.R. Rodrigues, L.P. Tizzei, M.A.S. Netto, Job placement advisor based on turnaround predictions for HPC hybrid clouds, Future Gener. Comput. Syst. 67 (2017) 35–46.

[2] M.A. Netto, R.N. Calheiros, E.R. Rodrigues, R.L. Cunha, R. Buyya, HPC cloud for scientific and business applications: Taxonomy, vision, and research challenges, ACM Comput. Surv. 51 (1) (2018) 1–29.

[3] Y. Fan, Z. Lan, T. Childers, P. Rich, W. Allcock, M.E. Papka, Deep reinforcement agent for scheduling in HPC, 2021, arXiv preprint arXiv:2102.06243.

[4] G. Fox, J.A. Glazier, J. Kadupitiya, V. Jadhao, M. Kim, J. Qiu, J.P. Sluka, E. Somogyi, M. Marathe, A. Adiga, et al., Learning everywhere: Pervasive machine learning for effective high-performance computation, in: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, IEEE, 2019, pp. 422–429.

[5] H. Mao, M. Alizadeh, I. Menache, S. Kandula, Resource management with deep reinforcement learning, in: Proceedings of the 15th ACM Workshop on Hot Topics in Networks, 2016, pp. 50–56.

[6] S. Huang, S. Ontañón, A closer look at invalid action masking in policy gradient algorithms, 2020, arXiv:2006.14171.

[7] G. Domeniconi, E.K. Lee, A. Morari, CuSH: Cognitive ScHeduler for heterogeneous high performance computing system, in: Proceedings of DRL4KDD 19: Workshop on Deep Reinforcement Learning for Knowledge Discovery, DRL4KDD, 2019.

[8] R.L. de Freitas Cunha, L. Chaimowicz, Towards a common environment for learning scheduling algorithms, in: 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS, 2020, pp. 1–8.

[9] D. Zhang, D. Dai, Y. He, F.S. Bao, B. Xie, RLScheduler: An automated HPC batch job scheduler using reinforcement learning, in: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2020, pp. 1–15.

[10] H. Mao, M. Schwarzkopf, S.B. Venkatakrishnan, Z. Meng, M. Alizadeh, Learning scheduling algorithms for data processing clusters, in: Proceedings of the ACM Special Interest Group on Data Communication, 2019, pp. 270–288.

[11] Y. Wang, H. Liu, W. Zheng, Y. Xia, Y. Li, P. Chen, K. Guo, H. Xie, Multi-objective workflow scheduling with deep-Q-network-based multi-agent reinforcement learning, IEEE Access 7 (2019) 39974–39982.

[12] A.M. Kintsakis, F.E. Psomopoulos, P.A. Mitkas, Reinforcement learning based scheduling in a workflow management system, Eng. Appl. Artif. Intell. 81 (2019) 94–106.

[13] J.-b. Lu, Y. Yu, M.-l. Pan, Reinforcement learning-based auto-scaling algorithm for elastic cloud workflow service, in: H. Shen, Y. Sang, Y. Zhang, N. Xiao, H.R. Arabnia, G. Fox, A. Gupta, M. Malek (Eds.), Parallel and Distributed Computing, Applications and Technologies, Springer International Publishing, Cham, 2022, pp. 303–310.

[14] D. Carastan-Santos, R.Y. De Camargo, Obtaining dynamic scheduling policies with simulation and machine learning, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017, pp. 1–13.

[15] Y. Fan, Z. Lan, Exploiting multi-resource scheduling for HPC, SC Poster (2019).

[16] E.R. Rodrigues, R.L. Cunha, M.A. Netto, M. Spriggs, Helping HPC users specify job memory requirements via machine learning, in: 2016 Third International Workshop on HPC User Support Tools, HUST, IEEE, 2016, pp. 6–13.

[17] Y. Fan, P. Rich, W.E. Allcock, M.E. Papka, Z. Lan, Trade-off between prediction accuracy and underestimation rate in job runtime estimates, in: 2017 IEEE International Conference on Cluster Computing, CLUSTER, 2017, pp. 530–540, http://dx.doi.org/10.1109/CLUSTER.2017.11.

[18] W. Smith, V. Taylor, I. Foster, Using run-time predictions to estimate queue wait times and improve scheduler performance, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 1999, pp. 202–219.

[19] W. Smith, I. Foster, V. Taylor, Predicting application run times with historical information, J. Parallel Distrib. Comput. 64 (9) (2004) 1007–1016.

[20] M. Xu, C. Song, H. Wu, S.S. Gill, K. Ye, C. Xu, EsDNN: Deep neural network based multivariate workload prediction in cloud computing environments, ACM Trans. Internet Technol. (2022) http://dx.doi.org/10.1145/3524114, Just Accepted.

[21] M. Arlitt, T. Jin, A workload characterization study of the 1998 world cup web site, IEEE Netw. 14 (3) (2000) 30–37.

[22] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, 2016, arXiv preprint arXiv:1606.01540.

[23] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, N. Dormann, Stable-Baselines3: Reliable reinforcement learning implementations, J. Mach. Learn. Res. (2021).

[24] R.L. de Freitas Cunha, L. Chaimowicz, On the impact of MDP design for reinforcement learning agents in resource management, in: Brazilian Conference on Intelligent Systems, Springer, 2021, pp. 79–93.

[25] D.G. Feitelson, L. Rudolph, Toward convergence in job schedulers for parallel supercomputers, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 1996, pp. 1–26.

[26] D.G. Feitelson, Metrics for parallel job scheduling and their convergence, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 2001, pp. 188–205.

[27] G. Tesauro, TD-Gammon, a self-teaching backgammon program, achieves master-level play, Neural Comput. 6 (2) (1994) 215–219.

[28] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al., A general reinforcement learning algorithm that masters chess, shogi, and go through self-play, Science 362 (6419) (2018) 1140–1144.

[29] Y. Liang, M.C. Machado, E. Talvitie, M. Bowling, State of the art control of atari games using shallow reinforcement learning, in: AAMAS, 2016, pp. 485–493.

[30] U. Lublin, D.G. Feitelson, The workload on parallel supercomputers: Modeling the characteristics of rigid jobs, J. Parallel Distrib. Comput. 63 (11) (2003) 1105–1122, http://dx.doi.org/10.1016/S0743-7315(03)00108-4, URL: https://www.sciencedirect.com/science/article/pii/S0743731503001084.

[31] C.B. Lee, Y. Schwartzman, J. Hardy, A. Snavely, Are user runtime estimates inherently inaccurate? in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 2004, pp. 253–263.

[32] D. Zotkin, P.J. Keleher, Job-length estimation and performance in backfilling schedulers, in: Proceedings. the Eighth International Symposium on High Performance Distributed Computing (Cat. No. 99TH8469), IEEE, 1999, pp. 236–243.

[33] A.W. Mu'alem, D.G. Feitelson, Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling, IEEE Trans. Parallel Distrib. Syst. 12 (6) (2001) 529–543.

[34] S.-H. Chiang, A. Arpaci-Dusseau, M.K. Vernon, The impact of more accurate requested runtimes on production job scheduling performance, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 2002, pp. 103–127.

[35] D. Tsafrir, Y. Etsion, D.G. Feitelson, Modeling user runtime estimates, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 2005, pp. 1–35.

[36] R.S. Sutton, D. Precup, S. Singh, Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning, Artificial Intelligence 112 (1) (1999) 181–211, http://dx.doi.org/10.1016/S0004-3702(99)00052-1.

[37] A.G. Baydin, B.A. Pearlmutter, A.A. Radul, J.M. Siskind, Automatic differentiation in machine learning: A survey, J. Mach. Learn. Res. 18 (2018) 1–43.

[38] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, 2017, arXiv preprint arXiv:1707.06347.

[39] W. Tang, Z. Lan, N. Desai, D. Buettner, Y. Yu, Reducing fragmentation on torus-connected supercomputers, in: 2011 IEEE International Parallel & Distributed Processing Symposium, IEEE, 2011, pp. 828–839.

[40] S. Hotovy, Workload evolution on the Cornell theory center IBM SP2, in: Workshop on Job Scheduling Strategies for Parallel Processing, Springer, 1996, pp. 27–40.

[41] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, A. Akella, Multi-resource packing for cluster schedulers, ACM SIGCOMM Comput. Commun. Rev. 44 (4) (2014) 455–466.

[42] M. Hausknecht, P. Stone, Deep recurrent q-learning for partially observable mdps, in: 2015 AAAI Fall Symposium Series, 2015.

[43] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Adv. Neural Inf. Process. Syst. 30 (2017).

Renato Luiz de Freitas Cunha is a Software Engineer at the Research for Industry group at Microsoft Research. He received his Ph.D. degree in Computer Science from Universidade Federal de Minas Gerais (UFMG) in 2022. His research interests include the intersection between Machine Learning and systems, with a focus in Deep Reinforcement Learning.

Luiz Chaimowicz is a Full Professor at the Computer Science Department of the Universidade Federal de Minas Gerais (UFMG). He received his Ph.D. degree in Computer Science from this same university in 2002, and from 2003 to 2004, he held a Postdoctoral Research appointment with the GRASP Laboratory at University of Pennsylvania. Dr. Chaimowicz co-directs UFMG's Vision and Robotics Lab. (VeRLab) and the J Lab –Multidisciplinary Research Lab. on Digital Games. His research interests include the application of Artificial Intelligence and Machine Learning in different areas such as Robotics, Digital Games and Computer Systems.