applied sciences

MDPI

Article

RLSchert: An HPC Job Scheduler Using Deep Reinforcement Learning and Remaining Time Prediction

Qiqi Wang [1], Hongjie Zhang [1], Cheng Qu [1], Yu Shen [2], Xiaohui Liu [2] and Jing Li [1,2,*]

[1] School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China; wang77@mail.ustc.edu.cn (Q.W.); zhanghongjie@mail.ustc.edu.cn (H.Z.); qucheng@mail.ustc.edu.cn (C.Q.)
[2] Supercomputing Center, University of Science and Technology of China, Hefei 230026, China; shenyu@ustc.edu.cn (Y.S.); liuxiaohui@ustc.edu.cn (X.L.)
* Correspondence: lj@ustc.edu.cn

Abstract: The job scheduler plays a vital role in high-performance computing platforms. It determines the execution order of the jobs and the allocation of resources, which in turn affect the resource utilization of the entire system. As the scale and complexity of HPC continue to grow, job scheduling is becoming increasingly important and difficult. Existing studies relied on user-specified or regression techniques to give fixed runtime prediction values and used the values in static heuristic scheduling algorithms. However, these approaches require very accurate runtime predictions to produce better results, and fixed heuristic scheduling strategies cannot adapt to changes in the workload. In this work, we propose RLSchert, a job scheduler based on deep reinforcement learning and remaining runtime prediction. Firstly, RLSchert estimates the state of the system by using a dynamic job remaining runtime predictor, thereby providing an accurate spatiotemporal view of the cluster status. Secondly, RLSchert learns the optimal policy to select or kill jobs according to the status through imitation learning and the proximal policy optimization algorithm. Extensive experiments on real-world job logs at the USTC Supercomputing Center showed that RLSchert is superior to static heuristic policies and outperforms the learning-based scheduler DeepRM. In addition, the dynamic predictor gives a more accurate remaining runtime prediction result, which is essential for most learning-based schedulers.

Keywords: high-performance computing; RLSchert; scheduling; deep reinforcement learning; remaining runtime prediction

Citation: Wang, Q.; Zhang, H.; Qu, C.; Shen, Y.; Liu, X.; Li, J. RLSchert: An HPC Job Scheduler Using Deep Reinforcement Learning and Remaining Time Prediction. *Appl. Sci.* **2021**, *11*, 9448. https://doi.org/10.3390/app11209448

Academic Editors: Antonio J. Pena and Pedro Valero-Lara

Received: 9 September 2021
Accepted: 7 October 2021
Published: 12 October 2021

1. Introduction

Job scheduling is becoming increasingly critical and challenging in large-scale high-performance computing (HPC) platforms as the scale and complexity of the platforms continue to increase [1,2]. Users submit their jobs to a centralized job scheduler through scripts or commands. The submitted job information includes the required resources, requested runtime, job ID, queue, and so on. Jobs wait in the queue until there are enough resources in the system, and the scheduler arranges them to execute in a certain order. A reasonable job scheduler can achieve high resource utilization, ensure fairness among different jobs, reduce the average wait time, and improve user satisfaction. As a well-known NP-complete problem [3], the most straightforward and common solution is to make scheduling decisions through a heuristic priority function, which assigns the priority to each job based on its attributes [3–8]. Some scheduling strategies use the simplest priority function, such as first come, first served (FCFS) using job submission time, shortest job first (SJF) [9] using runtime, and smallest area first (SAF) [10] using the job requested area: runtime × resources. In addition, some schedulers calculate the priority based on more attributes such as UNICEF [4] and F1 [5]. In order to increase the utilization of HPC resources, these heuristic scheduling methods are combined with a backfilling

mechanism [11,12]. Backfilling allows small jobs in the waiting queue to run in advance without affecting other jobs being executed.

However, today's complex HPC environments make the aforementioned heuristic strategies no longer efficient, and even using backfilling optimization is difficult because of the need to accurately predict the runtime of jobs. As the scale of HPC becomes larger, the number of jobs and the characteristics change randomly. It is difficult for heuristic indicators designed manually to improve system performance in a targeted manner, and they are unable to cope with changes in workload and optimization goals.

In recent years, learning-based scheduling strategies have been proposed. Reinforcement learning (RL) [13,14] is a general solution for sequential decision-making tasks, and RL agents continuously interact with the environment and continuously improve their own strategies based on the interaction experience and finally learn the optimal strategy. Therefore, an RL-based scheduling algorithm [15–19] can effectively deal with the complex and changeable HPC environment and naturally solves the shortcomings of the static heuristic strategy. This kind of scheduler can learn high-quality scheduling policies towards various workloads and different optimization goals with relatively low computation costs. Unfortunately, the existing RL-based schedulers rely on the estimated job completion time. Overestimation or underestimation will cause the scheduler to have a deviation in the current cluster load estimation, resulting in unreasonable job scheduling. For example, overestimating the completion time of a running job will make the scheduler mistakenly believe that there are enough free resources for backfilling short jobs. After backfilling, subsequent jobs will be delayed. Conversely, underestimation will make the scheduler consider that there are not enough free resources at present and thus lose the opportunity to backfill short jobs. Accurate estimation of the remaining completion time of running jobs is particularly important for scheduling strategies. Therefore, designing an RL-based scheduler has to confront two challenges: Firstly, it is a difficult problem to predict the job runtime. Runtime prediction is the key to obtaining the state of the system. Secondly, the environmental state changes are more complicated, and the scheduler must be able to learn the optimal strategy efficiently.

To solve the above-mentioned challenges, we propose RLSchert, a reinforcement learning scheduler based on the job remaining time prediction. In response to the first challenge, we designed a job remaining runtime prediction algorithm based on the intermediate log during job running and the characteristics of job input parameters. No matter what kind of job, the intermediate state can always be obtained in some way, such as a job output log, resource monitoring log, or instrumenting job code. We generated time series data by obtaining the intermediate results. Based on the job's running features (time series) and input characteristics (parameter vectors), we designed a multimodal job remaining time prediction algorithm. As the job runs, the estimation of the completion time will be more and more accurate. In this paper, we made runtime predictions for VASP [20], the most common and challenging HPC job. For the second challenge, we designed a parallel deep reinforcement learning training algorithm based on proximal policy optimization (PPO) [21] and imitation learning [22] to efficiently learn the optimal scheduling strategy. In order to improve scheduling performance, our scheduler dynamically selects jobs based on job remaining time predictions or kills and reschedules jobs from the current cluster. Since the action space of the agent is larger than the existing RL-based algorithm, this paper proposes a training method based on imitation learning and reinforcement learning, which greatly reduces the training cost. We used the VASP job logs of the University of Science and Technology of China (USTC) Supercomputing Center [23] as the simulation data to verify the effectiveness of RLSchert. The experimental results showed that RLSchert is significantly better than the static heuristic scheduling strategy and the RL-based strategy DeepRM [15] in terms of average slowdown and completion time.

In summary, our main contributions are summarized as follows:

- We propose a job running prediction model for the first time. This model dynamically predicts the remaining time according to the characteristics of the job running process and provides a more accurate view of the cluster status for the scheduler;
- We propose RLSchert (https://www.dropbox.com/sh/s08pqhxa6psyl33/AACFzEeQ0xcG5wsj534GlVNYa?dl=0 accessed on 10 October 2021), a job scheduler based on deep reinforcement learning (DRL) and remaining time prediction with the dynamically changing cluster state information. RLSchert uses imitation learning and reinforcement learning to efficiently train the scheduling strategy and successfully introduces runtime information into the strategy;
- We use real job logs of the Supercomputing Center of USTC as the simulation data. Experiments showed that RLSchert is significantly better than the existing algorithms in terms of average slowdown and completion time. At the same time, the runtime prediction algorithm proposed in this paper is effective at predicting the job remaining runtime and provides better support for other leaning-based scheduling strategies.

The reminder of this paper is organized as follows: We begin with preliminary knowledge of deep reinforcement learning in Section 2. In Section 3, we present a thorough description of the proposed RLSchert. Section 4 presents the experimental setup and results. The discussion of the related work is summarized in Section 5. Finally, we conclude this paper and discuss the future work in Section 6.

2. Preliminaries

First, we review the basic concepts and policy gradient of reinforcement learning as the preliminaries of RLSchert. Then, we introduce the PPO algorithm in detail.

2.1. Basic Concepts of Deep Reinforcement Learning

Deep reinforcement learning has achieved outstanding results in sequential decision-making tasks. Generally speaking, there are two components: environment and agent. The agent receives the state of the environment, usually a high-dimensional observation, such as an image, signal, etc. The agent evaluates the rewards obtained by using different actions and outputs the action with the highest reward to the environment. The environment changes its state according to the state transition function and returns the reward. The process is repeated until the end of the environment. The goal of the agent is to maximize the cumulative reward of the entire process, that is the sum of the reward at each time step. This is why deep reinforcement learning is more difficult than traditional supervised learning.

The task of reinforcement learning needs to be modeled as a Markov decision process (MDP), which is defined by a five-tuple (S, A, R, P, γ). The formal definition of MDP is shown as follows.

Definition 1. *MDP is defined by a five-tuple (S,A,R,P,γ):*

1. *S is the state space;*
2. *A is the action space;*
3. *P is the state transition function:*

$$P^a_{ss'} = \mathbb{P}(S_{t+1} = s' | S_t = s, A_t = a) \tag{1}$$

4. *R is the reward function,* $R^a_s = \mathbb{E}[R_t | S_t = s, A_t = a]$;
5. *γ is the discount factor,* $\gamma \in (0, 1]$.

The goal of DRL is to learn a state–action map function that maximizes the cumulative reward. The state–action function is named the policy function, $\pi : S \rightarrow S$, which is represented by a deep neural network. The cumulative reward under policy π is defined in Equation (2), where τ is the interaction sequence between the agent and the environment, $(s_0, a_0, r_0, s_1, a_1, r_1, \cdots)$. The probability of the trajectory $p(\tau)$ is the joint

probability distribution of the state transition function P and policy π. s_t is the state at time step t, and $a_t \sim \pi(\cdot|s_t)$ is the action at time step t. r_t is the reward at time step t.

$$J(\pi) = \mathbb{E}_{\tau \sim p(\tau)} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \tag{2}$$

2.2. Policy Gradient

The policy gradient is the parameter update direction of the policy function, and the optimal policy is learned along the policy gradient. In this work, the policy is a deep neural network parameterized by θ. The policy is represented by $\pi_\theta(\cdot|s)$. We derive the gradient of the optimization target $J(\pi)$ to the parameter θ, as shown in Equation (3), where G_t is the cumulative reward. In order to reduce the variance of policy gradient, we replace G_t with a value function or advantage function.

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) G_t] \tag{3}$$

The policy gradient with the advantage function is called the advantage actor–critic (A2C), which is defined in Equation (4), where $Q_w(s,a)$ represents the cumulative reward after performing the action a in the state s and continues to use the policy π_θ. $V_w(s)$ is the value function of state s.

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a)(Q_w(s,a) - V_w(s))] \tag{4}$$

The behavior policy and target policy of the A2C must be consistent, which is an on-policy algorithm. The on-policy algorithm cannot use the data generated by the old policy, which reduces the data efficiency. Proximal policy optimization (PPO) extends the A2C, which is defined in Equation (5), where $\pi_{\theta_{old}}$ represents the behavior policy, which is a delayed policy with π_θ. KL is the Kullback–Leibler divergence to estimate the difference between two distributions. By constraining the similarity between the behavior and target policy, PPO will converge.

$$\begin{aligned} \mathbb{E}_{\tau \sim p(\tau)} \left[\sum_{t \geq 0} \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} (Q_w(s,a) - V_w(s)) \right] \\ s.t. KL(\pi_\theta(a_t|s_t) || \pi_{\theta_{old}}(a_t|s_t)) < \delta \end{aligned} \tag{5}$$

3. RLSchert

3.1. Overview

Figure 1 shows the main components and their relationships in the HPC scheduling simulator. The entire framework includes two parts: environment simulator and scheduler agent. The simulator contains the main components of the HPC job scheduler, including the jobs data generator, remaining time predictor, backlog, waiting queue, cluster status, running queue, ready queue, and scheduler. In this work, we used the real-world VASP jobs from the Supercomputing Center of USTC as our simulation dataset. The predictor estimates the remaining runtime of each job at any time by using the input parameters and running logs while the job runs. The simulator adds new jobs one by one according to the Bernoulli distribution. The scheduler selects jobs from the waiting queue for execution on the cluster or terminates jobs in the cluster and appends them to the waiting queue for rescheduling. According to the current cluster state s, the agent selects the scheduling action a, and the cluster executes it. The simulator performs the action, updates the status, and returns the reward to the agent. The waiting queue, cluster status, and running queue are used as the input to the agent. The interaction between the agent and the environment is serial. In this work, the agent executes multiple actions at the same time. We discuss all of the components in our RLSchert below.

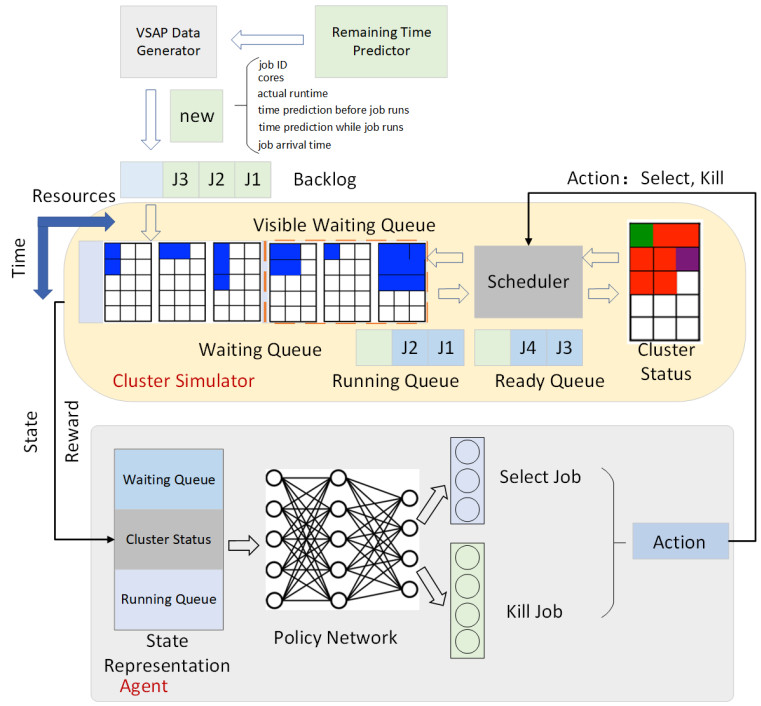


Figure 1. The overview of RLSchert and HPC simulator.

3.2. Remaining Time Predictor

We used the real-world VASP data from the Supercomputing Center of USTC as the simulation jobs. In the VASP data generator, we make runtime and remaining time predictions for each time step of the job. The remaining time prediction is performed during the job runs. We simplify the prediction process in the simulator. Before the job enters the waiting queue, the prediction of each time step has been completed. Specifically, we predict the remaining time of the job based on the input parameters and the intermediate results during the job runs. The input parameters of VASP are complex. We selected the parameters that had a key impact on the runtime, as shown in Table 1.

We also extract the running features from the intermediate results during the job runs, including **energy-change** and **force**. The running features are time series data generated at each time step during the job runs. In order to combine the input parameters and time series features, we designed a deep learning model with multimodal features, as shown in Figure 2. Specifically, the predictor contains a recurrent neural net (RNN)-based model to encode the time series features and a fully connected model to combine the input parameters to predict the final runtime.

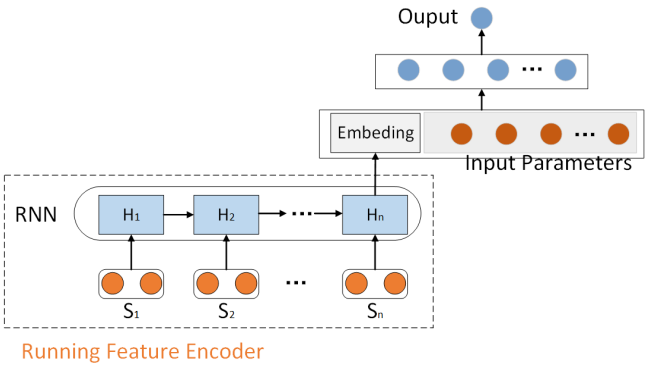


Figure 2. The neural network of the remaining time predictor.

Table 1. Features extracted from the VASP input files and system logs.

Feature	Description
NELECT	total number of electrons
NSW	number of steps for ionic upd
SystemVolume	volume of the system
AtomNumber	number of atoms
ElecNumber	number of electrons
NELM	maximum steps of electronic self-consistent iteration
ElementType Number	number of element types that appear
ENCUT	energy cutoff in eV
ALGO index	algorithm: normal (Davidson), fast, very fast
EDIFFG	stopping criterion for ionic upd
EDIFF	stopping criterion for electronic upd
Exhosts group	the group of exhosts
numExhosts	number of exhosts
PREC index	precession: medium, high, or low
cores	number of cores
NELMIN	No. of electronic steps
ISPIN	whether to perform spin polarization calculation
GGA index	xc-type: PW PB LM or 91
SIGMA	broadening in eV: -4-tet-1-fermi 0-gaus
numExhosts	number of exhosts
KPointMethod	the type of KPoint
ICHARGE	initialized charge density method
ISYM	symmetry: 0-nonsym 1-usesym
ISMEAR	part. occupancies
LREAL index	nonlocal projectors in real space
IBRION	ionic relaxation
ISTART	startjob: 0-new 1-cont 2-samecut

3.3. Job Scheduler Based on Deep Reinforcement Learning

This section describes the job scheduler based on deep reinforcement learning. We modeled job scheduling as a sequential decision problem and formally defined the basic elements of the Markov decision process (MDP). We derived the policy gradient (PG) based on the MDP and used PPO to train the agent. Specifically, we designed a scheduler with the job remaining runtime and a job termination rescheduling policy. At last, we propose an efficient training algorithm to learn the policy.

HPC Job Scheduler Model

The HPC simulator designed in this work is close to the real-world environment. As Figure 1 shows, the simulator contains the model of clusters, queues, and jobs. The idea of RLSchert is in both the time and resource dimensions. Assuming that we can accurately predict the job runtime, the scheduling becomes a two-dimensional bin packing (BP) problem. It is possible to obtain the optimal solution through DRL, which is better than the heuristic schedulers such as SJF. In the simulator, the cluster is modeled as a CPU pool composed of finite cores. Jobs arrive step by step in discrete time and wait in the queue. The scheduler selects one or more jobs from the waiting queue to the cluster at each time step. The following parts are formal definitions of the main modules.

Job: The job is the basic unit of scheduling, and a job has a corresponding feature description when the job arrives. The formal definition of job j is shown in Equation (6), where ID is the identification of the job, which is unique in the simulator. res represents the requested CPU cores. $enter\_time$ is the timestamps when the job arrives. $start\_time$ is the timestamps when the job starts to run. len_real represents the actual runtime of the job, and len_param is the predicted runtime based on input parameters before the job runs. $[run_1, run_2, \cdots, run_n]$ represents the predicted remaining runtime at each time step. $finish\_time$ is the timestamps when the job finishes, and $finish\_time = start\_time + len\_real$. $finish\_time\_hat$ is the predicted finish time. $finish\_time\_hat = start\_time + len\_param$ be-

fore the job runs, and $finish\_time\_hat = start\_time + run_i$ during the job runs. For the scheduler, if a job finishes before the predicted time, the rest resources are released immediately. If a job exceeds the predicted time, the simulator will add resources to it until the job finishes. Based on the prediction of the remaining runtime, the predicted finish time of the job must be re-estimated at each time step, and resources will be added or deleted in advance.

$$\begin{aligned} job_j = (&ID, res, enter\_time, start\_time, finish\_time, finish\_time\_hat, \\ &len\_real, len\_param, [run_1, run_2, \cdots, run_n]) \end{aligned} \tag{6}$$

Queue: After a new job is generated, the simulator puts it into the backlog, and then, the job enters the waiting queue and waits for the scheduler to schedule it. The size of the waiting queue is unlimited, but the number of visible jobs to the scheduler is num_nw. Controlling the number of visible jobs reduces the complexity of the scheduler and is beneficial for the training scheduling policy by DRL. Some slots in the visible jobs may be empty. For example, the number of jobs is less than num_nw. The running queue is a fixed-length queue that contains jobs running in the cluster and provides information for the scheduler to decide to kill a job. The ready queue is a temporary queue for each job in the cluster. We used the remaining time prediction, and the finish time of the job may change during job runs. For example, the allocated time of a job is too short, and additional time needs to be added, so the following jobs need to be reallocated. To simplify the simulator's design, we put jobs with $start\_time$ greater than the current time into the ready queue and wait for reallocating.

Cluster: The cluster is the computing resources. We did not consider the modeling of computing nodes and network communication. All computing resources in the cluster were regarded as a resource pool. The formal definition of the cluster is shown in Equation (7), where num_res represents the number of resource types. In this work, we only considered the CPU, which means $num\_res = 1$. $time\_horizon$ is the maximum time step available to the scheduler, which is greater than the maximum runtime of the jobs. res_slot is the number of CPU cores in the cluster. $avbl\_slot$ represents the number of available resources at each time step, which is defined as a vector $< v_1, v_2, \cdots, v_n >$. The same as DeepRM, $canvas$ is the resource occupation of each job in the cluster, which is a matrix M. The $M_{i,j}$ indicates which job occupies the j-th CPU at time step i. The job is also assigned a unique color.

$$cluster = (num\_res, time\_horizon, res\_slot, avbl\_slot, canvas) \tag{7}$$

Scheduler: The scheduler selects jobs from the waiting queue to the cluster according to the information such as the waiting queue, the running queue, the computing cluster, and the job features, or kills a job from the running jobs for rescheduling. The kill policy includes a rule-based policy and a PPO-based policy. A job is killed by the rule-based policy means that: the time of the job has been running is less than the predicted time before running (len_param), and $len\_param < 10$, the predicted time during running (run_i) is greater than 10, while the related error $|run_i - len\_param| > 1$. The PPO-based kill policy is described in detail below. The scheduler is a state–action function, which is defined in Equation (8), where $cluster\_status$, $Waiting\_queue$, and $Running\_queue$ represent the status of the cluster, the waiting queue, and the running queue, respectively. A is the action, which can be the slot index in the waiting queue, or the slot index where the job is terminated, or the empty action that controls the progress of the simulator.

$$f : (cluster\_status, Waiting\_queue, Running\_queue) \rightarrow A \tag{8}$$

3.4. Five-Tuple Definition of RLSchert

We used deep reinforcement learning to train the scheduler RLSchert, and a deep neural network represents the agent. According to the formal definition of the simulator,

we modeled the scheduling process as an MDP and trained the policy based on the policy gradient. An MDP can be described by a five-tuple (S, A, R, P, γ). The five-tuple of RLSchert is defined as follows.

S: The state space defines the state of the environment visible to the scheduler. Figure 3 shows the state space of RLSchert. Specifically, the state is represented by a 2D matrix, where the horizontal dimension represents the resource and the vertical dimension represents time. The leftmost is the cluster status. Colors represent different jobs, and free resources are represented by 0. On the right is the status of num_nw jobs in the waiting queue. Then, there is the status of the running queue. We extract features for each job as the input to the kill policy. Specifically, it contains five features, namely S_1, S_2, S_3, S_4, S_5. The last column is the extra information, namely the arrival time of the latest job.

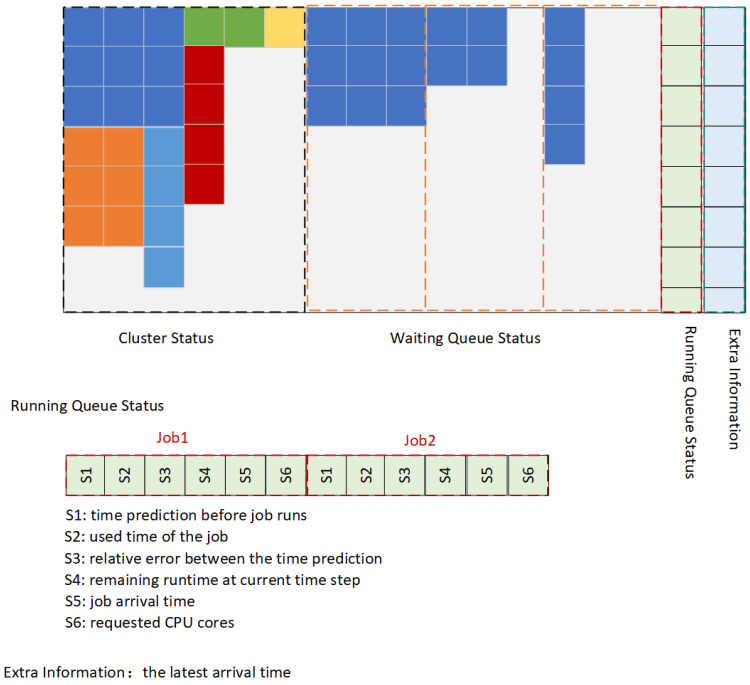


Figure 3. The state space of RLSchert.

A: The action space defines the actions that the scheduler could perform. Specifically, it contains num_nw select actions, num_kill kill actions, and a None action. The None action makes the simulator time move forward. The empty place in the queue is selected, which also represents the None action.

R: The reward function guides the policy improvement, which is designed by the target of the scheduler. In this work, we considered two scheduler metrics: the average slowdown and completion time. The slowdown of job i is defined as $S_j = (finish\_time - start\_time)/len\_real$, and the slowdown is always greater than one. The reward function for the average slowdown is defined in Equation (9), where $Predictor(j)[curr\_time]$ represents the predicted remaining runtime of job j at the current time step. The sum of R_t at each time step t is equal to the sum of the slowdown. A reward function such as this is to make the reward dense and accelerate the learning of the policy.

$$R_t = \sum_{j \in RunningQueue} -\frac{1}{Predictor(j)[curr\_time]} + \sum_{j \in ReadyQueue} -\frac{1}{Predictor(j)[0]} + \sum_{j \in WaitingQueue} -\frac{1}{Predictor(j)[0]} + \sum_{j \in Backlog} -\frac{1}{Predictor(j)[0]} \tag{9}$$

Furthermore, the completion time of each job is defined as $C_j = finish\_time - start\_time$. The completion time C_j represents the response time of job j. The reward function of the

completion time is defined in Equation (10). The sum of R_t at each time step t is equal to the sum of the completion time.

$$R_t = \sum_{j \in RunningQueue} -1 + \sum_{j \in ReadyQueue} -1 + \sum_{j \in WaitingQueue} -1 + \sum_{j \in Backlog} -1 \quad (10)$$

P: The state transition function defines the next state S_{t+1}after executing action A_t, that is $P : S \times A \rightarrow S$. The state transition is determined by the process of the simulator, which is a function that the agent needs to learn implicitly.

γ**:** This is the discount factor of the MDP. To ensure the finite value function and the convergence of the policy, the discount factor is less than or equal to one, as shown in the reward function. When the discount factor is equal to one, the cumulative reward is the slowdown or completion time. In the experiment, we set $\gamma = 0.99$, which also ensured the correct policy learning.

3.5. The Scheduler with Remaining Runtime Prediction Based on PPO

Based on the five-tuple definition of RLSchert, we used the most advanced DRL algorithm, PPO, to train the scheduling policy. The scheduling policy is represented by a deep neural network, $a_t \sim \pi_\theta(\cdot|s_t)$. The policy predicts the action a_t based on the current simulator state s_t. First, we describe the update rule of PPO. In each episode, a set of jobs arrive and are scheduled for execution. When all jobs are finished, the episode is complete. All data $(s_1, a_1, r_1, s_2, a_2, r_2, \cdots)$ generated in the episode are used to update the policy network. To speed up policy training, we used parallel PPO [24], as shown in Figure 4.

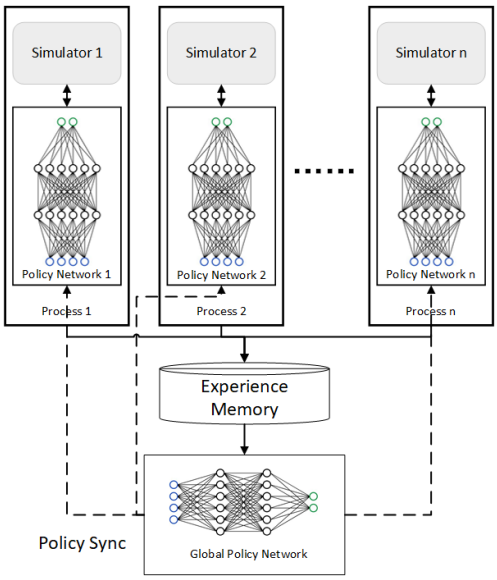


Figure 4. The parallel PPO algorithm.

The parallel PPO runs multiple simulators to generate data and then trains the policy network. These processes run independently to generate samples that are saved in the experience memory (EM). The global policy network samples data from the EM and updates the policy parameters by PPO. Then, we synchronize the latest policy network with each subprocess network. Through parallel sampling, the speed of the samples' generation increases, and the correlation between samples reduces. The RLSchert with runtime prediction before the job runs has the same training process, but the reward function R and state transition P are different. Specifically, in the slowdown metric, the first part of reward function is $\sum_{j \in RunningQueue} -1/Predictor(j)[0]$.

Algorithm 1 shows the pseudocode of PPO. In the parameters of the algorithm, *batchsize* is the number of data used at each policy update. *epochs* represents the maximum training steps of RLSchert. *ppo_epoch* is the maximum training steps. *KLtarget* represents

the stop condition of the PPO update. γ is the discount factor. η is the learning rate, and E is the number of parallel simulators.

Algorithm 1 The RLSchert with remaining runtime prediction based on PPO.

Input: batch size, epochs, ppo_epoch, KL target, γ, η, E

Output: RLSchert π_θ

initialize E parallel simulators
randomly generate an E job sequence to each simulator
for t = 1 to *epochs* **do**
  for e = 1 to E **do**
    interact with simulator e with the latest policy π_θ
    put the trajectory $(s_1^e, a_1^e, r_1^e, v_1^e, p_1^e, s_2^e, a_2^e, \cdots)$ into the EM
    calculate the cumulative rewards $R_t^e = \sum_{t' \geq t} \gamma^{t'-t} r_{t'}^e$
    calculate the advantage function $A_t^e = R_t^e - v_t^e$
  end for
  for i = 1 to ppo_epochs **do**
    sample a mini-batch of data from EM $(s_t, a_t, r_t, v_t, p_t, R_t, A_t, s_{t+1})$
    update the parameters of policy network $\Delta\theta \leftarrow \Delta\theta + \eta \nabla_\theta \frac{\nabla_\theta \pi_\theta(a_t|s_t)}{p_t} A_t$
    calculate the KL value $KL(\pi_\theta(a_t|s_t)||p_t)$
    if KL > 1.5 $\times$ KL target **then**
      break
    end if
  end for
end for

3.6. The Scheduler with the Kill Policy Based on PPO

Based on the scheduler with remaining runtime prediction, we added a set of kill actions to the agent and trained the policy by PPO. The addition of the kill policy leads to a larger action space, and the state transition P becomes more complicated. This makes policy training more difficult. It is impossible to interact with the simulator and improve the policy through a randomly initialized neural network. We used imitation learning and reinforcement learning to train the policy network with kill actions, thereby reducing the exploration and improving training efficiency.

Imitation learning was added to the PPO loss function, and the final policy loss function is formally defined as Equation (11), where $p(a|s)$ represents the action probability of behavior policy. $A(s,a)$ is the advantage function. $R(s)$ represents the cumulative reward after state s, which means the value function. $entropy(\cdot|s)$ is the entropy of the policy. The last term of the loss function is the imitation loss, where $a_{teacher}$ represents the action of the teacher policy.

$$\begin{aligned} Loss_{mimic} = & \frac{\pi_\theta(a|s)}{p(a|s)} A(s,a) + \alpha_v MSE(v_\theta(s), R(s)) + \alpha_e entropy(\cdot|s) \\ & + \alpha_t(-log\pi_\theta(a_{teacher}|s)) \end{aligned} \tag{11}$$

The hyperparameter α_t controls the strength of imitation loss. At the beginning of policy training, we set a larger α_t to learn the teacher policy. At the end of the training, we set α_t to zero, which improves the policy by reinforcement learning. In this work, the teacher policy contains two subpolicies. The first one is selecting jobs from the waiting queue. The second one is killing jobs from the running queue. The teacher of the select

policy is the RLSchert with the remaining runtime prediction. The teacher of the kill policy is a rule-based policy. Specifically, the actual running time of the jobs worth killing is long, and the prediction time before running is short. The job has been running for a short time. Then, the scheduler kills the mispredicted long job and selects a short job in the waiting queue to execute. Although the slowdown of the killed job increases, the overall average slowdown will decrease. The process of the RLSchert with the kill policy is the same as Algorithm 1. The difference is in the parameter update on Line 12. The kill policy needs to add an imitation loss function. As the training progresses, the imitating strength decreases gradually.

4. Experimental Results

In this work, we propose a DRL-based HPC scheduler with job remaining time prediction. Our method RLSchert tries to predict the remaining runtime of each running job and adjust the time slice of the job computing resource. The scheduler dynamically releases or allocates the computing resource of each running job to provide more opportunities for the waiting jobs. In addition, the scheduler kills the misclassified long jobs and reschedules them after re-estimating the runtime. This is the first time that the HPC scheduler has considered the remaining runtime of the running job to estimate the cluster status more accurately. RLSchert makes full use of the remaining time information to achieve the best performance based on the policy gradient. In this section, we verify the effectiveness of RLSchert based on the real job logs of the Supercomputing Center of USTC. There are three questions that need to be answered:

1. Compared with the heuristic static scheduling strategies, can RLSchert achieve a significant improvement in scheduling performance?
2. Does RLSchert support different optimization goals?
3. What is the feasibility and effectiveness of job killing and re-scheduling?

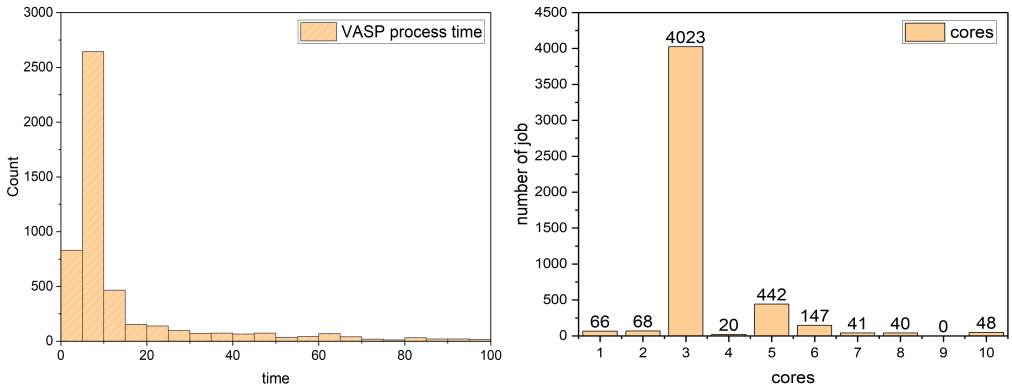
4.1. Experiment Setup

4.1.1. Real-World Job Dataset

We verified the effectiveness of RLSchert based on the VASP job logs of the Supercomputing Center of USTC. We extracted the input parameters before the job runs and extracted the running features when the job runs. Based on the input parameters, we predicted the runtime of each job. According to the running features, we predicted the remaining time of each running job and provided the current status of cluster resources. In the dataset, single ionic step jobs account for 70%. The rest are multiple ionic steps jobs usually with longer runtimes. In the experiment, the time of scheduling and job runtime was discretized. Specifically, we discretized time in units of 5 min. In order to reduce the complexity of the scheduler, the job's requested resource was divided by 10. The final runtime of the job and the distribution of requested resources are shown in Figure 5.

Figure 5a shows the runtime distribution of VASP. The average job runtime is 10 min (discretized by 5 min, meaning 50 min). Due to there being only a few jobs that run more than 100 min (1000 min), we limited the maximum runtime to 100 min. Figure 5b shows the distribution of the requested CPU cores of VASP. The average number of cores was three (meaning 30 cores in the real world). The same as the runtime, we limited the CPU cores to 10 (100 cores).

In the cluster simulator, the arrival time of each job is a Bernoulli process. The parameters of the Bernoulli process determine the job arrival rate and cluster load. The higher the job arrival rate, the higher the cluster load is, and vice versa. Each new job is randomly sampled from the VASP job dataset.



(**a**) The distribution of the runtime of VASP (**b**) The distribution of the CPU cores of VASP

Figure 5. The distribution of the runtime and CPU cores of VASP.

4.1.2. Cluster Simulator Setup

The environment simulator simulates the job arrival, job execution, job scheduling, and log process of the Supercomputing Center. It is also the target environment of the deep reinforcement agent. The detailed parameters of the cluster simulator are shown in Table 2. *simu_len* represents the maximum number of jobs in each simulation sequence. *num_ex* is the number of parallel simulators that execute independently. *max_seq* limits the maximum number of interactions between the agent and the simulator. *num_nw* represents the number of visible jobs of the scheduler. *time_horizon* is the visible resource slots of the time dimension. *job_len* and *job_size* separately represent the maximum job runtime and job requested resources, which were set according to the actual VASP job set. *res_slot* is the number of resources in our cluster, adjusted in the range of 12 to 24. The backlog is the queue, which contains the most recent new jobs. *job_rate* controls the job arrival rate, ranging from 0.3 to 0.95, and decides the cluster load. For example, "$job\_rate = 0.7$" means that at the current time step, there is a 70% probability that a new job will arrive.

Table 2. The parameters of the cluster simulator.

Parameter	simu_len	num_ex	max_seq	num_nw	time_horizon
Value	200	16	3000	5	120
Parameter	job_len	job_size	res_slot	backlog	job_rate
Value	100	10	12→24	600	0.3→0.95

4.1.3. PPO Setup

RLSchert adopts the proximal policy optimization algorithm. We implemented the policy network and training algorithm based on Python 3.6.12 and PyTorch 1.1.0. The hardware equipment includes a forty-eight-core Intel(R) Xeon(R) Gold 5118 CPU @ 2.30 GHz processors and four GeForce GTX-1080 Ti 12 GB GPUs. The policy network of RLSchert is a two-layer fully connected neural network, which contains 64 neurons at each layer. The input of the policy network is the status of the cluster simulator, which is a 120×72 matrix. The output dimension of the policy is 14, including 5 select actions, 8 kill actions, and 1 None action. The number of parameters is 557,952. We also designed other types of policy networks and RL algorithms, such as a convolutional neural network (CNN). The scheduling performances of each network and RL algorithm are shown in Table 3.

`pg_bs128_mlp64_64` indicates that the policy gradient is used to train RLSchert, and bs128 indicates that the batch size during training is 128. `mlp64_64` means that the policy function (mlp) is represented by a two-layer fully connected neural network, with 64 neurons in each layer. In the same way, `ppo_bs128_mlp64_64` means using PPO to train RLSchert. Both the first and second layers of the policy network have 64 neurons.

`ppo_bs128_cnn` uses a CNN to represent the policy function, and each layer of convolution has 16 kernels. The results showed that `ppo_bs128_mlp64_64` is the best configuration, and the experimental results were based on this configuration.

Table 3. The scheduling performances of each network and the RL algorithm.

RL Network Type	`pg_bs128_mlp64_64`	`ppo_bs128_cnn`	`ppo_bs128_mlp64_64`
average slowdown	7.16	7.59	7.03

The hyperparameters of PPO are shown in Table 4. The epochs represent the maximum number of training iterations. τ is the parameter of the advantage function. The KL target controls the early stopping of PPO. γ is the discount factor in DRL, which makes the training stable. The learning rate is the learning step of PPO. Specifically, we used the Adam optimizer to train the policy network. ϵ is the clip threshold. α_v, α_t, and α_e represent the weight of the value loss, mimic learning, and entropy, respectively.

Table 4. The parameters of the PPO algorithm.

Parameter	batch size	epochs	τ	KL target	γ
Value	128	5	0.95	0.01	0.99
Parameter	learning rate	ϵ	α_v	α_t	α_e
Value	0.0001	0.2	0.25	0.5	0.02

4.1.4. Compared Algorithm

In this work, we not only compared RLSchert with the traditional heuristic scheduling policies, we also compared DeepRM based on the runtime prediction before job runs and give the results of each scheduler using and not using the kill policy. The details of each policy are as follows:

1. **Random:** The scheduler randomly selects a job from the waiting queue and allocates resources;
2. **SJF:** This is the shortest job first. The SJF assigns jobs in increasing order of their predicted time before the job runs;
3. **SAF:** This is the smallest area first. Using the predicted runtime before the job runs and request resources, the SAF orders jobs by their area (runtime$\times$ resources);
4. **Tetris [25]:** Taking job request resources and cluster available resources as indicators, the scheduler selects the job with the smallest value, which is an implementation of packing [c];
5. **Tetris* [15]:** This is a scheduler based on Tetris, which balances preferences for short jobs and resource packing in a combined score;
6. **DeepRM [15]:** This is based on the predicted time before the job runs as information, using the PPO algorithm to learn the model.

4.1.5. Metrics

We used the mean absolute percentage error (MAPE) to evaluate the performance of the remaining time prediction, which is defined in Equation (12), where N is the number of jobs. y_i represents the actual runtime of job i, and $\hat{y}_i$ is the predicted runtime.

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{12}$$

In the HPC scheduling, the most useful metric is the average slowdown, which is defined in Equation (13), where N represents the number of jobs. t_{finish_i} is the finish timestamps of job i, and $t_{arrival_i}$ is the arrival timestamps of job i. $t_{duration_i}$ represents the

real execute time of job i. The average slowdown represents the normalized response time of each job.

$$average \quad slowdown = \frac{1}{N}\sum_{i=1}^{N}\frac{t_{finish_i} - t_{arrival_i}}{t_{duration_i}} \tag{13}$$

In addition, another important metric is the average completion time, which is defined in Equation (14), where N represents the number of jobs. t_{finish_i} is the finish timestamps of job i, and $t_{arrival_i}$ is the arrival timestamps of job i. The average completion time represents the response time of each job.

$$average \quad completion = \frac{1}{N}\sum_{i=1}^{N}[t_{finish_i} - t_{arrival_i}] \tag{14}$$

4.2. Prediction Error of Remaining Runtime

In the VASP dataset, 70% of them are single ionic jobs, which is the short job, and 30% of them are multiple ionic jobs, which is the long job. Its input parameters can recognize the type of VASP. We used extreme gradient boosting (XGBoost) [26] to predict the runtime based on the input parameters before the job runs. We used RLSchert to predict the remaining time of each running job based on the running features. To verify the performance of running prediction with different time steps, we split the running time series features into ten parts, representing the time already used by the job. For example, we could use the first 10% of the running features to predict the remaining time of the job. Table 5 shows the MAPE of each runtime predictor. The result shows that RLSchert with running time series features outperforms the predictor only using the input parameters significantly.

Table 5. The MAPE of each runtime predictor.

Methods	Single Ionic Step	Multiple Ionic Steps	Overall
XGBoost	29.9	67.3	39.8
RLSchert	**4.9**	**25.4**	**10.3**

Figure 6 shows the MAPE of RLSchert with different time steps. The MAPE of the runtime decreased significantly as the percentage of the time steps increased. The MAPE of the single ionic VASP reached about 3.55% near the end of the jobs. The MAPE of multiple ionic VASP reached about 17.9% near the end of the jobs.

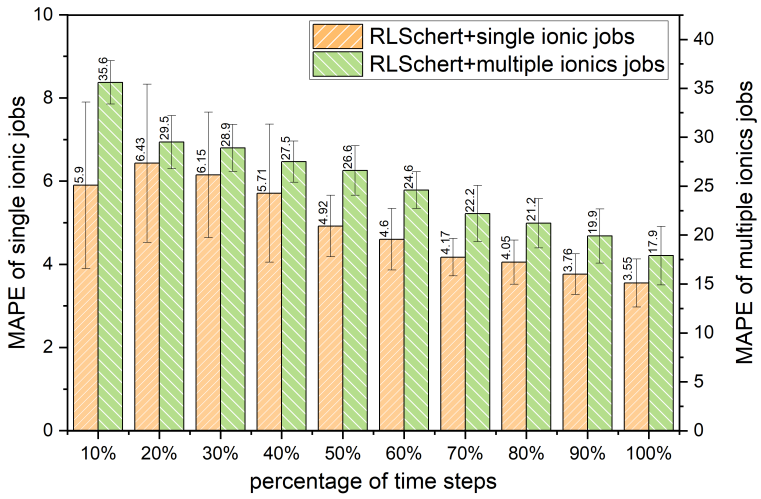
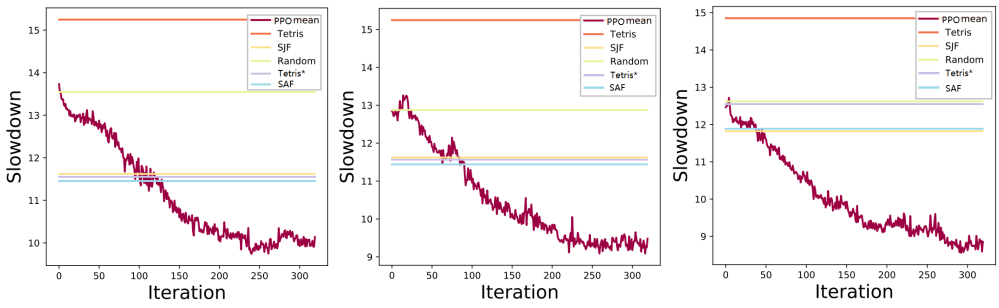


Figure 6. The MAPE of RLSchert with different time steps.

4.3. Convergence Behavior

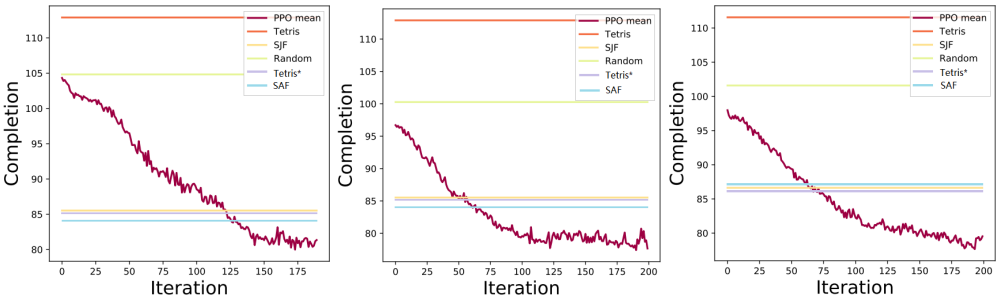
The policy function of RLSchert is learned by the policy gradient (PG) algorithm, which is randomly initialized. Specifically, we used the state-of-the-art method PPO. The convergence curves of RLSchert are shown in Figure 7a–c. It can be seen from the policy convergence curve that the PPO initially maintains a similar quality to the random policy and rapidly improves the policy quality. After training to 100 steps, the PPO algorithm surpasses the best static scheduling policy, the SAF, and converges to the optimal policy after 250 steps. From Figure 7a–c, we notice that RLSchert performs better than DeepRM, which only considers the runtime prediction before the job runs. In addition, RLSchert is better than RLSchert without the kill policy, which shows that the kill policy according to the rules is reasonable. Since the heuristic policy does not consider the future available resources in the cluster, it has no effect on them based on the time predictions before and during the job runs, which is shown as a straight line in the figure.

In addition, we verified the policy convergence behavior of each scheduler on the completion time metric, as shown in Figure 8. The experimental results are similar to the convergence curve of the slowdown. The policy training was completed at about 200 steps, and RLSchert was also significantly better than the heuristic static policies. The SAF scheduler can also be surpassed at about 100 steps by RLSchert.



(**a**) DeepRM. (**b**) RLSchert w/o kill. (**c**) RLSchert.

Figure 7. The convergence behavior of each scheduler on the average slowdown.

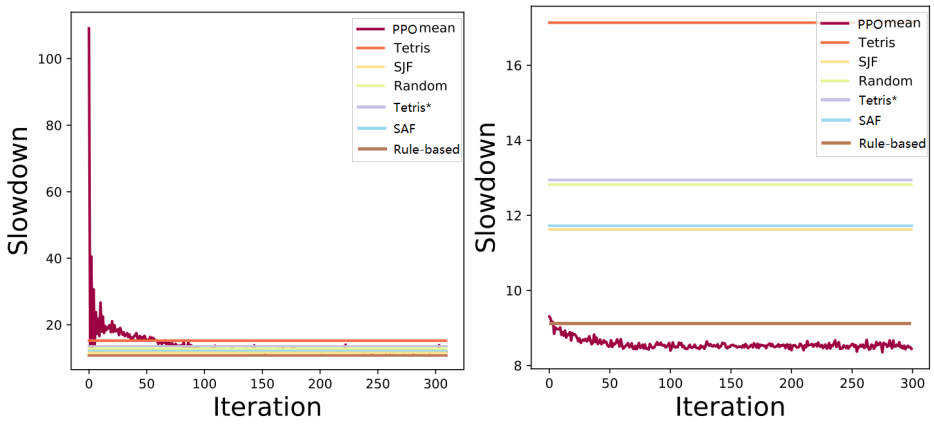


(**a**) DeepRM. (**b**) RLSchert w/o kill. (**c**) RLSchert.

Figure 8. The convergence behavior of each scheduler on the completion time metric.

The kill policy used in the previous experiment is a rule-based policy. If the policy network can learn when and which job needs to be killed, the effect may be better than the fixed policy. Therefore, we added the kill policy as an additional action to the policy network and trained it together with selecting the job. The addition of the kill policy significantly increases the action dimension, and the state transition function of the simulator is more complicated. In order to train the kill policy effectively, we used the "rule-based kill" and "RLSchert without kill" as the "teacher" policies, let the agent pretrain the RLSchert policy through imitation learning, and then performed PPO reinforcement learning on the pretrained model. Imitation learning reduces the random exploration in the simulator and ensures that the agent starts with a better policy to improve. After completing the

two stages of imitation learning and reinforcement learning, the PPO-based dynamic kill policy finally surpasses the rule-based kill policy. The kill policy training curve of imitation learning and reinforcement learning is shown in Figure 9.



(**a**) Imitation learning. (**b**) Reinforcement learning.

Figure 9. The convergence behavior of imitation and reinforcement learning with the kill policy.

4.4. Comparing Scheduling Efficiency

After analyzing the RLSchert convergence curve, this section discusses the final policy quality of each scheduler in detail. To verify the generalization of RLSchert, in addition to the training set, 200 × 16 VASP jobs that did not participate in the training were randomly selected as the test set, where 200 is the maximum length of the job sequence and 16 is the number of job sequences. This section verifies two metrics of each scheduling policy on the test set: average slowdown and average completion time.

4.4.1. Average Slowdown Comparison

First, the final policy quality of each scheduler on the training set was verified. In DeepRM and RLSchert, the job_rate was set to 0.7. The comparison result of the slowdown of each scheduler on the training set is shown in Table 6. From the results, we can observe that, regardless of whether the kill policy is used, RLSchert performs better than other schedulers. Besides, the kill policy used here is the PPO-based policy. If the rule-based kill policy is used, the average slowdown is 8.93 (not shown in the table), which is bigger than 8.83. This also illustrates that the dynamic kill policy enhanced by PPO is better than the rule-based policy.

Table 6. The policy quality of the average slowdown on the training set.

Method	Tetris	Random	SJF	Tetris*	SAF	DeepRM	**RLSchert**
*Scheduling **without** kill policy*							
AveSlowdown	15.31	12.81	11.63	11.62	11.44	10.6	**9.7**
*Scheduling **with** kill policy*							
AveSlowdown	14.82	12.65	11.81	12.64	11.88	-	**8.83**

The average slowdown comparison result of each scheduler on the test set is shown in Table 7, which is consistent with the result on the training set. RLSchert with the PPO-based kill policy is the best. Besides, the result of RLSchert with the rule-based kill policy is 8.69. At the same time, as a static scheduling policy, the SAF is superior to other heuristic scheduling algorithms in the slowdown metrics.

Table 7. The policy quality of the average slowdown on the test set.

Method	Tetris	Random	SJF	Tetris*	SAF	DeepRM	**RLSchert**
*Scheduling **without** kill policy*							
AveSlowdown	13.77	12.21	10.29	10.24	10.23	9.63	**8.93**
*Scheduling **with** kill policy*							
AveSlowdown	14.75	11.2	10.25	10.41	10.33	-	**8.32**

4.4.2. Average Completion Time Comparison

As a valuable metric of the HPC scheduler, completion time reflects the response time of users' jobs and determines user experience. The comparison results of the completion time of each scheduler on the test set is shown in Table 8. Similarly, based on dynamic resource scaling, RLSchert is better than DeepRM and other heuristic algorithms. However, the improvement of RLSchert compared to RLSchert without the kill policy is not very obvious. This is because killing the job being executed, the time that the job has been executed is wasted, and its completion time increases. Unlike the slowdown metric, killing a long job will reduce the slowdown of other short jobs. The runtime wasted by killing long jobs is not enough to be compensated. The experimental results also verified this conclusion. However, compared with other algorithms, RLSchert still significantly decreases the completion time.

Table 8. The policy quality of the average completion time on the test set.

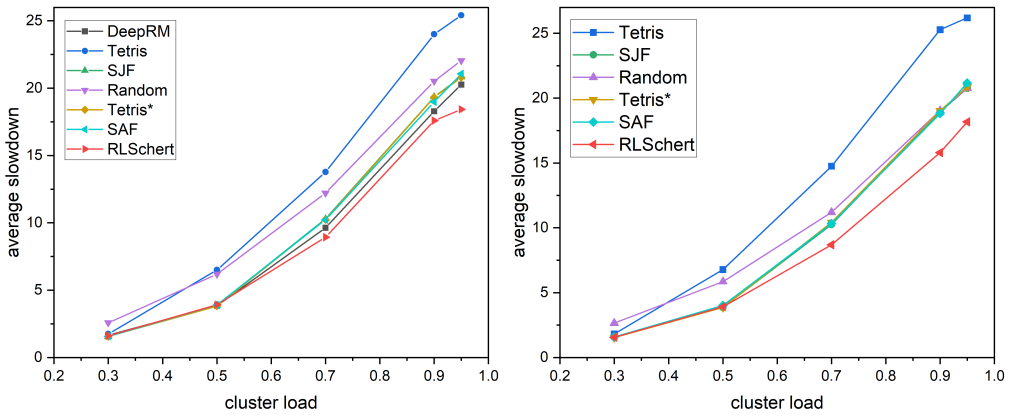
Method	Tetris	Random	SJF	Tetris*	SAF	DeepRM	**RLSchert**
*Scheduling **without** kill policy*							
completion time	98.66	98.15	80.36	80.37	80.3	77.18	**76.27**
*Scheduling **with** kill policy*							
completion time	105.66	94.95	80.82	80.93	81.13	-	**76.18**

4.5. The Influence of Cluster Setup

In the simulator, we set up variable parameters to verify the adaptability of RLSchert. This section studies two important parameters. The first is the cluster load, which simulates the cluster load by adjusting the job arrival rate (job_rate). The second is the maximum resources of the cluster, which is affected by the maximum number of cores (res_slot) in the cluster.

4.5.1. The Influence of the Cluster Load

The cluster load is controlled by job_rate. For example job_rate = 0.7 means that the probability of a job arriving at each time step is 0.7. The average slowdown of each scheduler varies with the cluster load, as shown in Figure 10. The results showed that as the cluster load increases, the average slowdown of all schedulers gradually increases. Figure 10a shows that without using the kill policy, both DeepRM and RLSchert based on DRL are better than traditional heuristic algorithms. Figure 10b shows that when using the kill policy, RLSchert is significantly better than the other compared schedulers. When the cluster load is too low, the job can be executed at any time without queuing, so all schedulers show a similar average slowdown. The higher the cluster load, the more tasks are queued, and RLSchert gradually shows its advantages. In addition, it can be observed from the figure that among all the static algorithms, the SAF performs best under different loads.

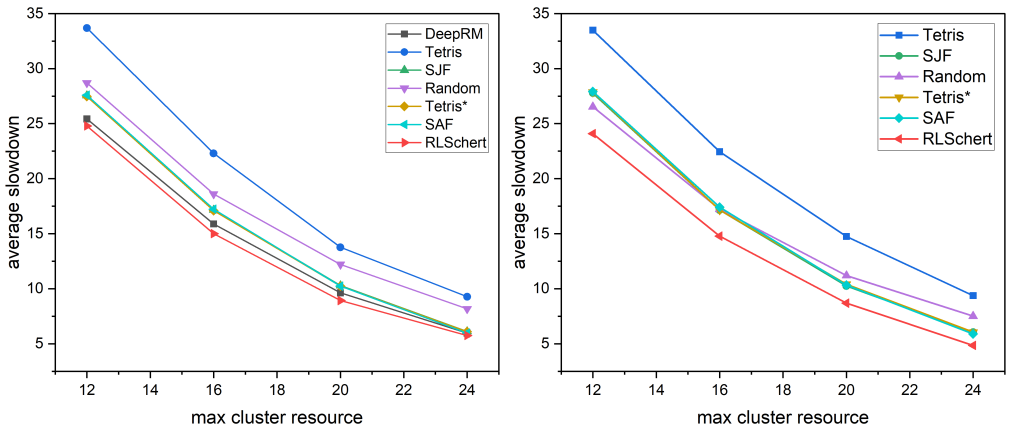


(**a**) Schedulers without the kill policy. (**b**) Schedulers with the kill policy.

Figure 10. The average slowdown of each scheduler varies with the cluster load.

4.5.2. The Influence of Maximum Resources

The maximum resource of the cluster has a significant impact on job scheduling. The more resources available in the cluster, the more jobs can be processed at the same time, and the lower the average slowdown is. The slowdown of each scheduler varies with the number of available resources in the cluster, as shown in Figure 11. The results showed that as the maximum cluster resources increases from 12 to 24, the average slowdown gradually decreases. Figure 11a shows the slowdown of each scheduler without the kill policy. Although RLSchert can scale resources through the remaining time prediction, the effect is not significant. Besides, the SAF is the best scheduler compared to other static schedulers under different available resources, and RLSchert is better than the SAF under various resources. Moreover, the fewer available resources, the more significant the slowdown improvement of RLSchert is. This is similar to the cluster load. The less the available resources, the longer the job will wait, and vice versa. Figure 11b shows the slowdown with available resources for each scheduler based on the kill policy. Under this policy, the effect of RLSchert is significantly better than the other schedulers, highlighting the importance of the remaining time prediction and dynamic kill policy.



(**a**) Schedulers without the kill policy. (**b**) Schedulers with the kill policy.

Figure 11. The average slowdown of each scheduler varies with the cluster maximum resources.

4.5.3. The Influence of The Predictable Job Proportion

In addition, we experimented with the effect of the predictable job proportion on RLSchert. In the real world, not all the remaining times of the jobs can be predicted while the job is running. We controlled the percentage of predictable jobs and give the average

slowdown of the entire job set in Table 9. The result showed that as the proportion of predictable jobs increases, the slowdown of RLSchert decreases. Besides, due to original SJF strategy being only related to the initial predicted value of job runtime, the execution order of the jobs will never be adjusted, and the slowdown of the SJF is a fixed value.

Table 9. The average slowdown with different predictable proportions.

Scheduler	0%	50%	80%	100%
SJF	10.29	10.29	10.29	10.29
RLSchert	9.63	8.55	8.36	8.32

4.5.4. The Influence of Mixed Applications

Since there are other types of jobs in HPC platforms, we discuss the influence of mixed applications in this section. VASP jobs account for about 50% of the Supercomputing Center of USTC [27]; therefore, the prediction model in RLSchert was trained after the feature engineering of VASP jobs. For other types of iterative jobs, a different feature engineering needs to be performed (this is beyond of the scope of this work). For this part of the jobs whose input parameters cannot be extracted or feature engineering has not been completed, we used the mature general predictor based on user behavior [28]. The predictor does not need to know the input parameters and running features. The features it uses include: user ID, job submission time, previously completed jobs from the same user, request cores, queue ID, group ID, and so on. We implemented the user behavior predictor to predict all applications. The prediction results are shown in Table 10.

Table 10. The MAPE of each prediction model.

Methods	VASP	Others	Overall
UserBehavior [28]	101.5%	121.2%	110.8%
XGBoost	39.8%	-	39.8%
RLSchert	10.3%	-	10.3%

Especially, to run predictable jobs (VASP), RLSchert predicts them. For the rest of the random applications, the general predictor predicts them, which basically addresses the random applications issue. Furthermore, we experimented with our RLSchert based on all jobs (VASP and other applications) of the USTC Supercomputing Center. The experimental results are shown in Table 11. Table 11 shows the average slowdown of each scheduler when all VASP jobs use RLScher to predict the runtime and others use the user behavior predictor. The experimental results showed that RLSchert has the lowest average slowdown, which illustrates the effectiveness of RLSchert in practical HPC with a mix of applications.

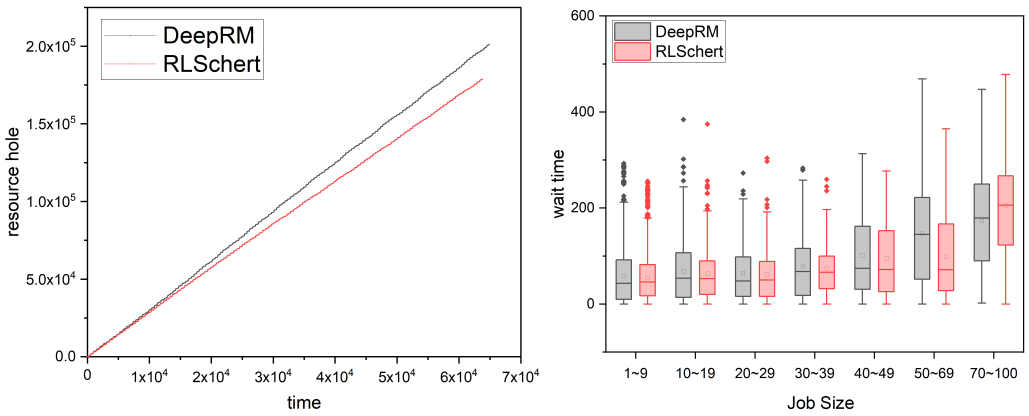
Table 11. The average slowdown of each scheduler.

Method	Tetris	Random	SJF	Tetris*	SAF	DeepRM	**RLSchert**
AveSlowdown	14.42	12.81	11.9	11.08	11.7	9.89	**8.41**

4.6. Analysis of RLSchert's Performance Gain

4.6.1. Resource Utilization

Both RLSchert and DeepRM are scheduling algorithms based on deep reinforcement learning. Compared with DeepRM, RLSchert dynamically adjusts the resources based on the remaining time prediction, thereby reducing the resource fragmentation (as shown in Figure 12).



(**a**) Resource hole in DeepRM and RLSchert. (**b**) Job wait time in DeepRM and RLSchert.

Figure 12. The comparison of resource utilization and job wait time.

RLSchert can better utilize the cluster resources. To verify this, we counted the number of resource holes along with the time step. Resource holes are the resources wasted at the current time step (idle resources). Figure 12a shows the accumulative resource hole with the time step. RLSchert consistently outperforms DeepRM, which reduces the occurrence of resource holes. Figure 12b shows the wait time distribution of jobs when using different schedulers. Although the wait time for long jobs using RL is longer, the wait time for short jobs is shorter, indicating that short jobs are more likely to be backfilled into the resource hole through runtime prediction. We can conclude that RLSchert precisely adjusts the resources, giving more opportunity to better utilize the cluster resources and improve the performance.

4.6.2. Job Kill Policy Visualization

RLSchert can more accurately know the remaining runtime of the task by predicting the runtime of the running job. In the VASP job time prediction, many jobs run too long or too short, which leads to incorrect estimates of the runtime. For the slowdown metric, scheduling a short job first will have a lower average slowdown. However, the job runtime prediction before running is unreliable. If a long job is predicted to be a short job and is scheduled to be executed first, this will inevitably lead to an increase in the slowdown of the actual short job in the waiting queue. Suppose we can quickly find such jobs and terminate them immediately and append them into the waiting queue. In that case, we will have the opportunity to schedule other short jobs, thereby reducing the overall slowdown. This section visualizes the jobs killed by RLSchert, as shown in Figure 13. The results showed that some jobs' runtimes are significantly underestimated, which affects the average slowdown. RLSchert automatically recognizes such jobs through reinforcement learning and kills them in time. In Figure 13, the job that is killed has a low runtime prediction before running and a long actual runtime. At the same time, the used time of the job is also low, which means this kind of job will be found to be mispredicted not long after being scheduled. In this way, the job is worth killing.

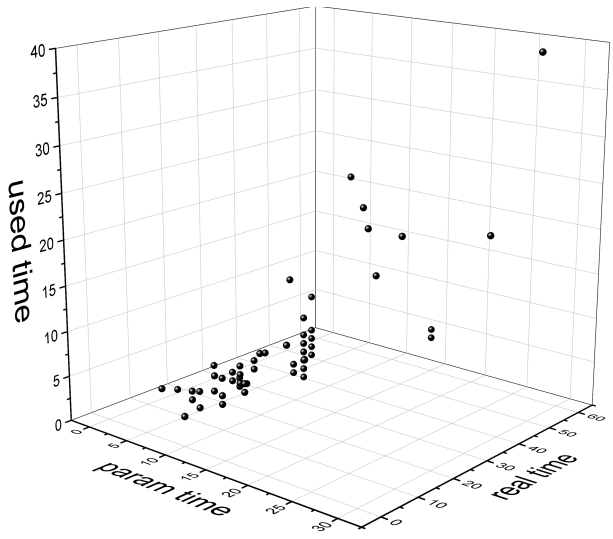


Figure 13. Visualization of tasks killed by RLSchert.

5. Related Work

In HPC, job scheduling has been a long-standing research topic [2–12,29]. Maximizing resource utilization, reducing resource fragmentation, and improving user satisfaction have always been the goals of researchers. Therefore, various scheduling strategies have been proposed and developed, ranging from the simplest and straightforward heuristic algorithms (such as first come, first served (FCFS), short job first (SJF) and short area first (SAF) [10]) to more sophisticated policies (WFP3 and UNICEF) [4]. Subsequently, machine learning techniques have been applied in the field of scheduling. F1, F2, F3, and F4 [5] are nonlinear functions obtained using nonlinear regression. The calculated value of the function corresponds to the priority in the execution order of the job. These functions are constructed by brute force simulation on a large number of samples. The classification technology is also combined with the HPC scheduler. Jobs are classified into long and short jobs through classification, and short jobs are executed first [30]. In addition, there are some optimized backfilling algorithms: combine job runtime prediction with easy backfilling [31], and dynamically adjust the estimated job completion time during the simulation process [11,12].

Recently, reinforcement learning has been used for various learning tasks, ranging from computer games, autonomous driving, and robotics [13,14,32]. RL can learn policies according to the environment, so it is applied to various system optimization tasks. Combining RL with resource scheduling is also a new trend in recent years, such as DeepRM [15,16], CuSH [17], Decima [33], and RLScheuler [34]. Although these studies used reinforcement learning methods as RLSchert, the runtime estimates they used are all fixed values. Regardless of whether these values are accurate or not, they have not verified the impact of incorrectly estimated runtime on the system. Therefore, it cannot be proven that these algorithms can still make correct decisions in a real dynamic HPC environment.

6. Conclusions and Future Work

This study presents RLSchert, an HPC job scheduler based on deep reinforcement learning and remaining time prediction. RLSchert is a scheduler that integrates runtime prediction and job scheduling. Accurate prediction of the remaining runtime of the job provides a guarantee for grasping the status of the cluster, and the scheduling strategy based on reinforcement learning can make correct decisions based on the status of the cluster. The kill policy based on imitation learning further guarantees the efficiency of the scheduler, and when the optimization goal changes, the scheduler can still learn the best policy. In the selection of the dataset, we chose the most common and most complex HPC job: VASP. We conducted extensive evaluations and confirmed that RLSchert performs well

on real-world workloads and different optimization goals. The lowest average slowdown demonstrated high stability, and the shortest job completion time illustrated a reasonably low computational cost.

There are two derived interesting questions that could be studied in the future: the first is extending this study to other types of iterative computing jobs in real HPC platforms to prove the universality of our model; secondly, we would like to study multi-index optimization more and integrate RLSchert into a real HPC management system.

Author Contributions: Conceptualization, Q.W.; methodology, Q.W.; software, Q.W., H.Z. and C.Q.; data curation, Y.S. and X.L.; writing, Q.W. and J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Strategic Priority Research Program of the Chinese Academy of Sciences, Grant No. XDA19020102.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data produced by the experiments conducted in this paper can be found in https://www.dropbox.com/sh/s08pqhxa6psyl33/AACFzEeQ0xcG5wsj534GlVNYa?dl=0 accessed on 10 October 2021.

Acknowledgments: We are thankful for the support from the Supercomputing Center (SCC) of USTC. We are thankful to the administrators of the SCC for the data and professional knowledge support. We gratefully acknowledge the computing resources provided by the Network and Information Center to run our experiments.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Geist, A.; Reed, D.A. A survey of high-performance computing scaling challenges. *Int. J. High Perform. Comput. Appl.* **2017**, *31*, 104–113. [CrossRef]
2. Feitelson, D.G.; Rudolph, L.; Schwiegelshohn, U.; Sevcik, K.C.; Wong, P. Theory and Practice in Parallel Job Scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*; Springer: Berlin/Heidelberg, Germany, 1997; pp. 1–34.
3. Ullman, J.D. Np-complete scheduling problems. *J. Comput. Syst. Sci.* **1975**, *10*, 384–393. [CrossRef]
4. Tang, W.; Lan, Z.; Desai, N.; Buettner, D. Fault-aware, utility-based job scheduling on Blue, Gene/P systems. In Proceedings of the 2009 IEEE International Conference on Cluster Computing and Workshops, New Orleans, LA, USA, 31 August–4 September 2009; pp. 1–10.
5. Carastan-Santos, D.; de Camargo, R.Y. Obtaining dynamic scheduling policies with simulation and machine learning. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 12 November 2017; pp. 32:1–32:13.
6. Xhafa, F.; Abraham, A. Computational models and heuristic methods for Grid scheduling problems. *Future Gener. Comput. Syst.* **2010**, *26*, 608–621. [CrossRef]
7. Mu, A.W.; Feitelson, D.G. Utilization, Predictability, Workloads, and user Runtime Estimates in Scheduling the IBM SP2 with ling. *IEEE Trans. Parallel Distrib. Syst.* **2001**, *12*, 529–543.
8. Agarwal, A.; Colak, S.; Jacob, V.S.; Pirkul, H. Heuristics and augmented neural networks for task scheduling with non-identical machines. *Eur. J. Oper. Res.* **2006**, *175*, 296–317. [CrossRef]
9. Pinedo, M. *Scheduling*; Springer: New York, NY, USA, 2012.
10. Carastan-Santos, D.; Camargo, R.Y.D.; Trystram, D.; Zrigui, S. One can only gain by replacing EASY Backfilling: A simple scheduling policies case study. In Proceedings of the CCGrid 2019—International Symposium in Cluster, Cloud, and Grid Computing, Larnaca, Cyprus, 1 May 2019; pp. 1–10.
11. Tsafrir, D.; Etsion, Y.; Feitelson, D.G. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Trans. Parallel Distrib. Syst.* **2007**, *18*, 789–803. [CrossRef]
12. Li, J.; Zhang, X.; Han, L.; Ji, Z.; Dong, X.; Hu, C. OKCM: Improving parallel task scheduling in high-performance computing systems using online learning. *J. Supercomput.* **2021**, *77*, 5960–5983. [CrossRef]
13. Kaelbling, L.P.; Littman, M.L.; Moore, A.W. Reinforcement learning: A survey. *J. Artif. Intell. Res.* **1996**, *4*, 237–285. [CrossRef]
14. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*; MIT Press: Cambridge, MA, USA, 2018.
15. Mao, H.; Alizadeh, M.; Menache, I.; Kandula, S. Resource management with deep reinforcement learning. In Proceedings of the 15th ACM Workshop on Hot Topics in Networks, New York, NY, USA, 9 November 2016; pp. 50–56.

16. Chen, W.; Xu, Y.; Wu, X. Deep reinforcement learning for multi-resource multi-machine job scheduling. *arXiv* **2017**, arXiv:1711.07440.
17. Domeniconi, G.; Lee, E.K.; Venkataswamy, V.; Dola, S. Cush: Cognitive scheduler for heterogeneous high performance computing system. In Proceedings of the DRL4KDD 19: Workshop on Deep Reinforcement Learning for Knowledge Discover, Anchorage, AK, USA, 5 August 2019.
18. Qin, H.; Zawad, S.; Zhou, Y.; Yang, L.; Zhao, D.; Yan, F. Swift machine learning model serving scheduling: A region based reinforcement learning approach. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 17 November 2019; pp. 1–23.
19. Fan, Y.; Lan, Z.; Childers, T.; Rich, P.; Allcock, W.; Papka, M.E. Deep Reinforcement Agent for Scheduling in HPC. *arXiv* **2021**, arXiv:2102.06243.
20. The Vienna Ab Initio Simulation Package. Available online: https://www.vasp.at/ (accessed on 9 September 2021).
21. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal policy optimization algorithms. *arXiv* **2021**, arXiv:1707.06347.
22. Schmitt, S.; Hudson, J.J.; Zidek, A.; Osindero, S.; Doersch, C.; Czarnecki, W.M. Kickstarting deep reinforcement learning. *arXiv* **2018**, arXiv:1803.03835.
23. Supercomputing Center of USTC. Available online: http://scc.ustc.edu.cn/ (accessed on 9 September 2021).
24. Heess, N.; TB, D.; Sriram, S.; Lemmon, J.; Merel, J.; Wayne, G. Emergence of locomotion behaviours in rich environments. *arXiv* **2017**, arXiv:1707.02286.
25. Grandl, R.; Ananthanarayanan, G.; Kandula, S.; Rao, S.; Akella, A. Multi-resource packing for cluster schedulers. In Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14), Chicago, IL, USA, 17–22 August 2014; pp. 455–466.
26. Chen, T.; Guestrin, C. Xgboost: A scalable tree boosting system. In Proceedings of the 22nd ACM Sigkdd International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, 13–17 August 2016; pp. 785–794.
27. Wang, Q.; Li, J.; Wang, S.; Wu, G. A novel two-step job runtime estimation method based on input parameters in HPC system. In Proceedings of the 2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA), Chengdu, China, 12–15 April 2019; pp. 311–316.
28. Gaussier, E.; Glesser, D.; Reis, V.; Trystram, D. Improving backfilling by using machine learning to predict running times. In Proceedings of the SC15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Austin, TX, USA, 15–20 November 2015; pp. 1–10.
29. Iosup, A.; Ostermann, S.; Yigitbasi, M.N.; Prodan, R.; Fahringer, T.; Epema, D. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Trans. Parallel Distrib. Syst.* **2011**, *22*, 931–945. [CrossRef]
30. Zrigui, S.; de Camargo, R.; Legrand, A.; Trystram, D. Improving the Performance of Batch Schedulers Using Online Job Runtime Classification. In press. Available online: https://hal.archives-ouvertes.fr/hal-03023222 (accessed on 10 October 2021).
31. Lifka, D.A. The anl/ibm sp scheduling system. In Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing, CA, USA, 25 April 1995; pp. 295–303.
32. Kober, J.; Bagnell, J.A.; Peters, J. Reinforcement learning in robotics: A survey. *Int. J. Robot. Res.* **2013**, *32*, 1238–1274. [CrossRef]
33. Mao, H.; Schwarzkopf, M.; Venkatakrishnan, S.B.; Meng, Z.; Alizadeh, M. Learning scheduling algorithms for data processing clusters. In Proceedings of the ACM Special Interest Group on Data Communication, Beijing, China, 19–23 August 2019; pp. 270–288.
34. Zhang, D.; Dai, D.; He, Y.; Bao, F.S.; Xie, B. RLScheduler: An automated HPC batch job scheduler using reinforcement learning. In Proceedings of the SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 9–19 November 2020; pp. 1–15.