Tiako P.F. (ed) *Competitive Tools, Techniques, and Methods*. Chronicle of Computing. OkIP. APDC24#6
© 2024 Oklahoma International Publishing https://doi.org/10.55432/978-1-6692-0007-9_1

Scheduling for high performance computing with reinforcement learning

Scott Hutchison[a]*, Daniel Andresen[a], William Hsu[a], Benjamin Parsons[b], Mitchell Neilsen[a]

[a]*Kansas State University, 110 Anderson Hall, 919 Mid-Campus Drive, Manhattan, KS 66506*
[b]*Engineering and Research and Development Center, 3909 Halls Ferry Road, Vicksburg, MS 39180*

ABSTRACT

Job scheduling for high performance computing systems involves building a policy to optimize for a particular metric, such as minimizing job wait time or maximizing system utilization. Different administrators may value one metric over another, and the desired policy may change over time. Tuning a scheduling application to optimize for a particular metric is challenging, time consuming, and error prone. However, reinforcement learning can quickly learn different scheduling policies dynamically from log data and effectively apply those policies to other workloads. This research demonstrates that a reinforcement learning agent trained using the proximal policy optimization algorithm performs 18.44% better than algorithmic scheduling baselines for one metric and has comparable performance for another. Reinforcement learning can learn scheduling policies which optimize for multiple different metrics and can select not only which job in the queue to schedule next, but also the machine on which to run it. The agent considers jobs with three resource constraints (CPU, GPU, and memory) while respecting individual machine resource constraints.

Keywords: High Performance Computing, Scheduling, Reinforcement Learning.

I. INTRODUCTION

Scheduling for High Performance Computing (HPC) systems is typically done using a batch scheduler. In most HPC systems, users will submit their jobs to a centralized job scheduler that will reserve and assign HPC resources according to the resources requested by the users at submission time. Typically, the system administrators managing the HPC system will configure the scheduler using an optimization goal, or metric, such as *maximizing HPC resource utilization, minimizing job wait time, maximizing job throughput*, etc. The optimization goal of system administrators may change from one time period to the next, and different HPC administrators may value one metric over another. Correctly configuring the batch scheduler to optimize for different metrics is challenging, and optimizing for one metric may adversely affect another. This research shows that reinforcement learning can learn different scheduling policies to optimize for different goals while remaining competitive with or performing better than algorithmic scheduling baselines.

Although batch scheduling has been shown to be an NP-Hard problem (Ullman, 1975), some job schedulers compute job priorities based on attributes of the submitted job. Examples of algorithmic scheduling based on job attributes include First Come First Serve (FCFS), Shortest Job First (SJF), Oracle SJF, and Best Fit Bin Packing (BFBP). More details of these algorithms are provided in Section II.A. More sophisticated schedulers use advanced techniques such as utility functions or machine learning to make their scheduling decisions. Recently, researchers have looked to Reinforcement learning (RL), one of the machine learning paradigms, to learn scheduling policies for HPC scheduling applications. With RL, we allow a machine learning agent to make scheduling decisions and provide feedback on its performance using a reward. When trained iteratively, the agent can learn a scheduling policy to maximize the reward it receives. When this reward is tied to the desired scheduling optimization goal, the RL agent makes scheduling decisions to optimize for the chosen goal. The questions this research set out to answer are as follows:

- Can RL provide a high-quality scheduling policy comparable to or better than algorithmic scheduling baselines?
- Is the learned policy only effective on the workload used for training, or can it generalize and remain effective on other previously unseen workloads?

The remainder of this paper is structured as follows: Section II provides details about the background and related works, as well as further details of the implementation of this work. Section III discusses the methodology and a comparison of the performance of a RL scheduling agent with algorithmic scheduling baselines. Section IV discusses the results of the experiment, the statistical analysis done, and the conclusions reached. Section V expands upon the challenges encountered in the work, as well as prospects for future work. Finally, section VI offers final concluding remarks.

II. BACKGROUND AND RELATED WORKS

A. *Algorithmic Scheduling Baselines*

The baseline scheduling algorithms to which the RL agent's performance will be compared are First Come First Serve (FCFS), Shortest Job First (SJF), Oracle SJF, and the Best Fit Bin Packing (BFBP) Algorithm. For FCFS, jobs are scheduled strictly in the order in which they arrive. If no machine in the HPC cluster has sufficient resources to execute the first chronological job in the queue, the scheduler waits until there are adequate resources to begin the execution of the first job in the queue. SJF will sort the job queue by requested job run time and begin running the shortest job that some machine in the cluster has adequate resources to execute. In practice, users of the HPC system tend to overestimate the amount of time and

* Corresponding author E-mail: scotthutch@ksu.edu

Presented at the 2024 OkIP International Conference on Advances in Parallel and Distributed Computing (APDC) in Oklahoma City, OK, USA, and Online, on April 3, 2024

resources their jobs require, as they do not want their job's execution to be halted by exceeding their requested resources. Overestimation of resources by the users, including requested job time, tends to degrade the performance of scheduling algorithms like SJF. Oracle SJF behaves like SJF except it sorts the job queue by actual run time instead of requested run time. This information cannot be known by the scheduler at job submission time when the scheduler is making its decisions, and scheduling using this information provides a decent lower bound for the average job waiting time. BFBP is perhaps the most relevant scheduling algorithm as it powers some scheduling applications in use on actual HPC systems today, like Slurm (Jette & Grondona, 2003). BFBP considers all jobs in the queue and the available resources for each machine in the cluster. BFBP selects the (job, machine) pair that will result in the fewest resources remaining for some machine in the cluster and begins executing that job on the chosen machine. This has been shown to take no more than $\lfloor 1.7 * OPT \rfloor$ machines (Dósa & Sgall, 2014), where OPT is the minimum number of machines required for a particular workload. Not only does this scheduling algorithm power some modern scheduling applications, but it also serves to provide a decent upper bound for the HPC cluster utilization metric for any particular workload.

B. *Reinforcement Learning for Policy Optimization*

In general, a RL agent is attempting to choose the best action to maximize its reward given the current state of the environment. The agent has knowledge of the environment through its observation space, and it sees the environment in state *S* at time *t*. The agent has certain actions available to it through its action space. The agent chooses action *A* at time *t*. This action will change the environment to a new state *S* at time *t+1*. The agent then receives some reward *R* at time *t+1* for its action. Based on the reward, the weights for the neural network powering the agent's decision-making process are updated and the process repeats until the reward converges to a maximum. The policy the agent learns over time will maximize the expected reward for a particular action given the current state of the observation space. The general framework for RL is shown in Fig. 1.

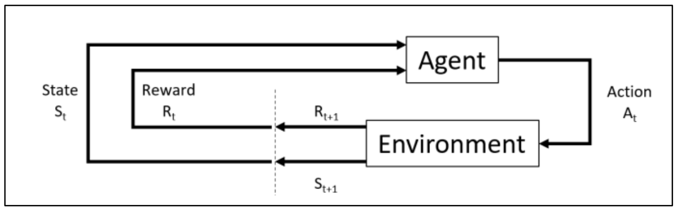

Fig. 1. The general framework for reinforcment learning

In the context of using RL to schedule for HPC systems, the observation space consists of the jobs in the job queue and the resources available on each machine. The actions available to the agent are selecting a particular job from the job queue and a machine from the cluster on which to run it. When the reward is tied to the optimization goal (job wait time or HPC utilization), the agent will learn to select the best job from the job queue and the machine on which to run it that will optimize for the desired metric. Metrics may be maximized (as with HPC system utilization) or minimized (as with job wait time). It was the goal of this research to ascertain if RL could learn multiple scheduling policies to optimize for different metrics, and if those policies could compete with algorithmic scheduling baselines on new, never-before-seen workloads. Additional details about the implementation used for this research can be found in Section II.E.

Proximal Policy Optimization (PPO) (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017) is an RL technique developed to improve upon and address some shortcomings in previous RL techniques. PPO is an on-policy technique, meaning that each update of the policy only uses data collected while acting according to the most recent version of the policy. The policy maps the states in the environment to actions taken when in those states. Generally, an objective function in RL returns the expected reward for an action in a given state. Rather than using gradient ascent on the objective function to optimize the policy (a computationally expensive process), PPO employs a surrogate objective function which gives a conservative estimate for how much the objective function will change as a result of a policy update. Large policy updates are penalized via clipping, resulting in quick training convergence and good performance for many tasks.

C. *HPC Scheduling Using Machine Learning*

Scheduling for HPC systems using RL techniques has been a topic of much research interest recently. One of the first attempts to do HPC scheduling with RL was accomplished with DeepRM (Mao, Alizadeh, Menache, & Kandula, 2016). Mao et al. showed that RL can learn multiple scheduling policies to compete with state-of-the-art heuristics when scheduling HPC jobs. Their RL agent uses gradient decent on policy parameters to maximize the expected cumulative discounted reward. It was shown to adapt to different conditions, converge quickly, and learn sensible scheduling strategies.

RLSchert (Wang, et al., 2021) was a more recent project using RL to perform HPC scheduling. This project included a remaining time predictor to better estimate how long a job will take in order to make better scheduling decisions. RLSchert incorporated requested memory and requested CPUs as resource constraints and learns a policy to select or kill jobs according to their status and estimated remaining time using the PPO technique.

Another RL HPC scheduler is A2cScheduler (Liang, Yang, Jin, & Chen, 2020). This technique uses the actor-critic deep-RL technique to perform scheduling and resource management for HPC systems. A2cScheduler also considers two job resources constraints, requested memory and requested CPUs.

The research most closely related this work is RLScheduler (Zhang, Dai, He, Bao, & Xie, 2020). Zhang et al. showed the viability of using RL and PPO to learn multiple scheduling policies for both real and constructed HPC workloads. However, the neural network powering RLScheduler only selects the next job to be started from the job queue. The machine on which the job is run is not selected by the agent, and the job is handed off to the cluster for execution on some machine. Also, each machine in the cluster is limited to running only one job at a time, which is a technique employed by some HPC systems. However, many HPC systems allow the allocation of multiple jobs to a single HPC node as resources allow, resulting in increased job throughput and higher HPC utilization. Additionally, RLScheduler only considers two resource constraints for each job, the job's requested memory and number of requested CPUs. It was the goal of this research to build upon what was accomplished with RLScheduler by including the following:

- Increase the number of constraints each job can request from two to three by allowing jobs to request memory, CPUs, and GPUs.
- Allow multiple jobs to execute on a single HPC machine while still respecting the machine's total resource constraints.
- Allow the RL agent to select not only a job from the queue, but also the machine in the cluster on which to run it.

Accomplishing these should bring HPC scheduling with RL one step closer to implementation on an actual HPC system.

D. *Workload Specification with Three Resource Constratints*

The jobs given to the scheduler to be scheduled comprise the workload. There are numerous HPC workloads available for use, and we have access to the log data of a local university HPC system to construct workloads for our use. A trend with HPC scheduling research thus far seems to be the consideration of only two resource constraints per job: the amount of requested memory and the number of requested CPUs. We believe this stems from a limitation of the Standard Workload Format (SWF) (Chapin, et al., 1999) method of specifying workloads for HPC systems. The SWF has become the de facto standard for HPC scheduling research and is used extensively in this space. Although the resource constraints included in the SWF are certainly important, analysis of our local HPC system has shown that jobs requesting GPU resources have become increasingly prevalent. This trend will likely continue as future AI researchers and others capitalize on GPUs and other hardware accelerators. In fact, on our HPC system, we have frequently observed low GPU availability being a key factor in extending job wait time. Alternatives to the SWF have been proposed, such as the Modular Workload Format (MWF) (Corbalan & D'Amico, 2021). The MWF incorporates not only non-classical computing resources (like GPUs), but also additional new job profiles as well. For this reason, consideration of a third resource constraint, requested GPUs, was seen as important for this work, as it is often overlooked in prior HPC scheduling research. Workloads were specified using a simple comma separated value (CSV) format, which included the following information about each job: *JobName, RequestedMemory, RequestedCPUs, RequestedGPUs, RequestedDuration, ActualDuration,* and *SubmitTime*. The characteristics of the machines comprising the simulated HPC cluster were specified using another CSV file with the following attributes: *MachineName, TotalMemory, TotalCPUs*, and *TotalGPUs*.

E. *OpenAI Gym Environment*

OpenAI Gym (Brockman, et al., 2016) is an open-source Python library and Application Programming Interface (API) for developing and comparing RL algorithms. OpenAI Gym specifies methods for implementing a custom environment to explore machine learning tasks. Within the custom environment, the observation space, action space, step function, and various others are defined. The RL agent's neural network provides a mapping from the observation space to the action space and allows the agent to maximize the expected return for its available actions. Invalid action masking (Huang & Ontañón, 2020) eliminates any impossible or clearly unproductive actions. In the context of our HPC Scheduling problem, invalid action masking removes any (job, machine) actions for which the machine does not have adequate available resources to run the job. Invalid action masking reduces the number of possible actions the agent must consider and has been shown to improve convergence time during training, as the agent will not attempt to schedule a job on a machine that cannot run it. If the agent were to choose an invalid action, the state of the observation space would not change, and the workload would not move any closer towards completion. Stable Baselines3 (SB3) (Raffin, et al., 2021) is a Python library that provides a set of reliable implementations of RL algorithms, and this research implemented a custom OpenAI Gym environment using a self-implemented discrete event simulator (DES) representing the HPC scheduling problem. The SB3 PPO with invalid action masking algorithm was used to train a RL agent, and then its performance was compared to algorithmic scheduling algorithms implemented in the DES. The DES keeps track of the current simulation time step and maintains lists for future jobs, queued jobs, running jobs, and completed jobs. As the simulation progresses and simulation time advances, jobs move through the lists in the DES, machines track their currently available resources. Upon simulation termination, the desired metrics can be calculated and compared to one another depending on the algorithm used to schedule the jobs to the simulated HPC.

The custom OpenAI Gym environment provides the definitions of the observation space and the action space. The observation space consisted of the first *n* schedulable jobs in the job queue. A job was considered schedulable if at least one machine in the cluster had adequate resources to schedule it. Many scheduling applications limit a maximum queue depth to search for jobs to schedule, and the code for this research was written to allow adjustment of the queue depth easily. The action space selects an individual schedulable job from the job queue and a machine on which to run it. This structure is depicted in Fig. 2. This was initially implemented as a multi-discrete action space, however, many scheduling algorithms require a discrete action space. This 2D action space was converted to a 1D action space similar to converting a 2D array into a 1D array using the following formula:

$$1D\_action = (machine\_index * num\_machines) + job\_index$$

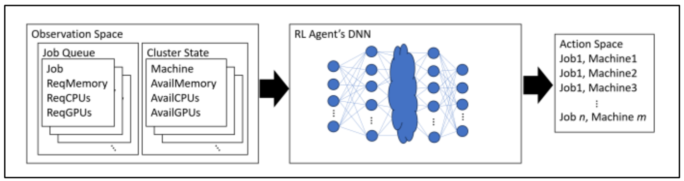

Fig. 2. The observation space and action space of the custom OpenAI Gym environment

The reward returned at each training step is tied to the desired policy. As implemented, the SB3 algorithm will maximize this episodic reward, so when minimizing job queue time, the reward per step was *reward = 0 - avg. job wait time*. When maximizing system utilization, *reward = HPC System utilization*. An episode consists of scheduling every job in a workload, and the cumulative reward across the entire episode is used to update the RL agent's learned scheduling policy.

III. METHODOLOGY

To evaluate the efficacy and the performance of using PPO with invalid action masking when scheduling jobs for HPC systems, the follow steps were taken:

- Construct 20 sets of jobs consisting of high utilization jobs (jobs requesting more than 20 CPUs) and modify the job's submission times to be the same. Add 2% of jobs requesting any number of GPU resources to the set.
- For each set of constructed jobs, train an agent using PPO with invalid action masking to minimize job wait time and another agent to maximize system utilization.
- Select 23 days' worth of jobs from HPC log data of days with approximately the same number of submitted jobs (1000 +/- 100 jobs).
- Schedule these 23 days using the algorithmic baselines.
- Use the trained agent to schedule these 23 days and compare its performance to the algorithmic scheduling baselines.
- Conduct analysis of variance (ANOVA) to determine if the differences of the means of the scheduling methods is statistically significant.
- For agents trained to minimize average job wait time, conduct pairwise t-tests between the agent's performance and the algorithmic baseline's average job wait time for the 23 days.
- For agents trained to maximize HPC system utilization, conduct pairwise t-tests between the agent's performance and the algorithmic baseline's average HPC system utilization for the 23 days.

A. *Training Workload Construction*

To construct the training workload, jobs from log data for the local HPC systems were sampled. Zhang et al. noted that the workload trajectory was important for productive training, and they utilized trajectory filtering to select only job traces that were productive for training their RL agent. Various training workloads from local HPC log data were tested for training, and the most productive and consistent training came from selecting "large jobs" that requested more than 20 CPUs. Training workloads were constructed by randomly selecting 2000 jobs requesting more than 20 CPUs from HPC log data, and randomly adding 40 jobs requesting any number of GPUs. This 2% addition of jobs requesting GPUs aligns with the historic level of such jobs from our HPC system. Then the jobs were shuffled, and modifying the submission time for all jobs such that they were submitted simultaneously. Essentially, we wanted challenging workloads for the scheduler, which would reward good scheduling decisions and punish poor ones. Each of the 20 workloads constructed was used to train a RL agent using two metrics: minimizing the average job wait time and maximizing HPC system utilization.

B. *Evaluation on Actual Workloads*

Again, log data was used to find days which had roughly the same number of submitted jobs. We sought days with enough jobs that scheduling would be a non-trivial activity for our simulated cluster of nine machines. Searching log data for days with 1000 +/- 100 submitted jobs yielded 23 different days from the log data considered. The resources requested, the submission time, and the actual run time for these jobs remained unchanged from the log data, and the 40 different agents were used to schedule the 23 days using the metric on which they were trained. The algorithmic baselines also scheduled the 23 days to provide a basis for comparison of the performance of the RL agent's scheduling.

C. *Statistical Analysis*

ANOVA (Girden, 1992) was utilized to determine if there was a statistically significant difference between the means of the models and the algorithmic scheduling baselines. Next, pairwise t-tests (Student, 1908) were conducted between each algorithmic scheduling baseline and the model to determine if difference between the means of the models and the algorithmic baseline was statistically significant. A significance level of 95% ($\propto = 0.05$) was used for all statistical tests.

IV. RESULTS

A. *Training Convergence*

Figure Fig. 3 shows the training curves for two agents trained on one set of constructed jobs. One agent was trained to minimize average job queue time, and the other was trained to maximize HPC system utilization. The training for the depicted day converges quickly to some maximal value, indicating that the agent has learned how to optimally schedule a chosen day for its given metric. Training was accomplished using HPC resources in parallel, and training each agent took no more than a few hours.

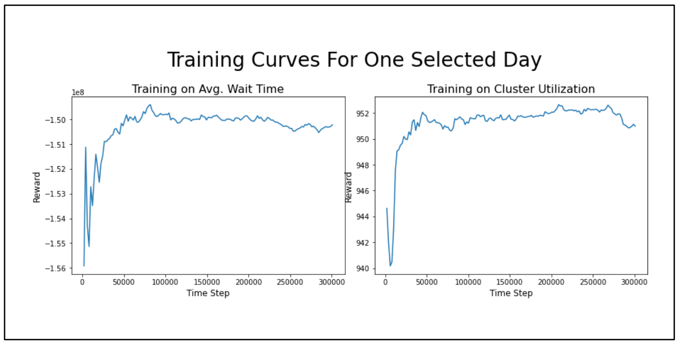

Fig. 3. The training curve of two RL agents on a selected day when trained for 300,000 steps on each of the two metrics

B. *Minimizing Average Job Wait Time*

Each of the 20 models trained to minimize average job wait time was used to schedule 23 days of jobs, and the performance of the model was compared to the algorithmic baselines of FCFS, Oracle SJF, SJF, and BFBP. Conducting ANOVA on the results showed there was a statistically significant difference between the average job wait times for the different scheduling algorithms (p-value $=1.395 * 10^{-15}$). Doing pairwise t-tests comparing the different scheduling methods to the model showed significance between the model and Oracle, BFBP, and FCFS. When using the metric of minimizing average job wait time, both SJF and Oracle scheduling performed better than the model. This is unsurprising as these scheduling algorithms are designed to minimize this metric. The model was able to schedule jobs such that the average job wait time was 18.44% lower than if they were scheduled using the BFBP algorithm. TABLE I. shows the mean average job wait time when scheduling all 23 days using the various scheduling techniques with statistically significant paired t-test p-values highlighted. A boxplot of the average job queue time can be found in Fig. 4.

TABLE I. RESULTS FOR MINIMIZING AVERAGE JOB WAIT TIME

Scheduling Method	Avg. Job Wait Time in Minutes (Lower is better)	T-test P-value vs. the Model
Oracle SJF	308.55	$3.156 * 10^{-10}$
SJF	454.05	0.9238
RL Model	456.79	n/a
BFBP	549.59	0.003718
FCFS	2779.20	$2.2 * 10^{-16}$

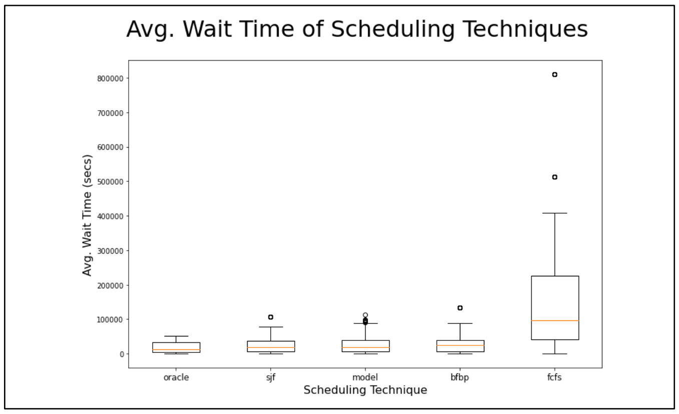

Fig. 4. Boxplot of average job wait times for the different scheudling techniques (lower is better).

C. *Maximizing HPC System Utilization*

Next, each of the 20 models trained to maximize HPC system utilization was used to schedule 23 days of jobs, and the performance of the model was compared to the same algorithmic baselines. Conducting ANOVA on the results showed there was a significant difference between the HPC system utilization for the different scheduling algorithms (p-value $=2.2 * 10^{-16}$). Doing pairwise t-test comparisons between the different scheduling methods showed significance between the performance of the model and FCFS. In terms of maximizing cluster utilization, BFBP did the best with the cluster utilization of 45.63% compared to the model's cluster utilization of 45.61%. The BFBP algorithm was designed to maximize this metric by maximizing the utilization of each of the machines in the cluster, so this is also unsurprising. Additionally, the inconclusive t-test indicates that we cannot reject the null hypothesis that the difference in the means of BFBP and the model is due to random chance. We cannot conclude that the model does better, but it is at least able to perform comparably to the algorithmic baselines when trained to maximize this metric. TABLE II. shows the mean HPC system utilization when scheduling all 23 days using the various scheduling techniques with statistically significant paired t-tests highlighted. A boxplot of the average HPC system utilization can be found in Fig. 5.

TABLE II. RESULTS FOR MAXIMIZING HPC SYSTEM UTILIZATION

Scheduling Method	Cluster Utilization Higher is better	T-test P-value vs. the Model
FCFS	41.26%	$2.2 * 10^{-16}$
SJF	44.73%	0.05552
Oracle SJF	45.18%	0.3567
RL Model	45.61%	n/a
BFBP	45.63%	0.9671

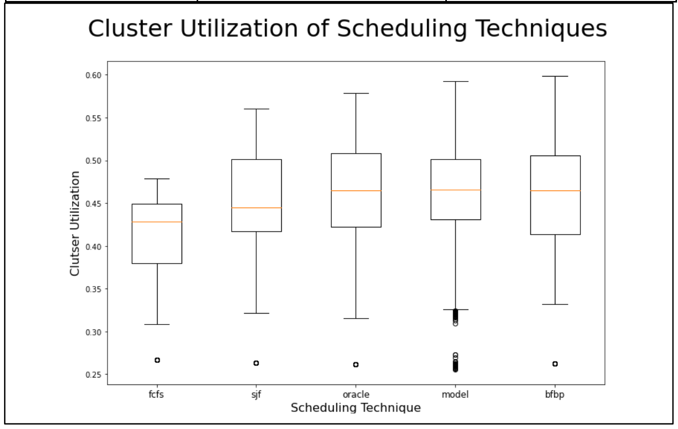

Fig. 5. Boxplot of cluster utilization for the different scheudling techniques (higher is better).

D. *Interpretation*

The RL agent trained using PPO with invalid action masking was able to beat or at least perform comparably to the algorithmic scheduling baselines. The agent's performance when minimizing average job queue time was 18.44% better than the performance of BFBP, a scheduling algorithm used to power modern scheduling applications. When maximizing HPC system utilization, there was no difference between the agent's performance and that of the algorithmic baselines, including BFBP. Returning to the research questions, it was demonstrated that RL can provide a high-quality scheduling policy comparable to or better than algorithmic scheduling baselines. Additionally, since the RL agent was trained on constructed workloads, and then evaluated using different workloads actually submitted to our local HPC system, it was able to successfully apply the scheduling policy it learned on one workload to another. Additionally, RL was able to accommodate three resource constraints per job (requested memory, requested CPUs, and requested GPUs), and successfully schedule on par with algorithmic scheduling baselines. Furthermore, the RL agent was able to schedule multiple jobs per machine and respect the resource constraints on each machine. The RL agent was able to select not only the job from the queue to schedule next, but also the machine on which to run it, similar to modern scheduling applications. Finally, this RL technique could learn different scheduling policies, one which was minimized average job queue time and one which was maximized HPC system utilization, showing its flexibility and applicability to the needs of different HPC system administrators.

V. Challenges and Future Work

Curiously, increasing the queue depth available to the RL agent beyond a certain point did not improve training convergence or the agent's performance. Queue depths of 10, 60, 100, 200, and 500 were investigated, and degradation of training value began when queue depth was greater than 100. One would think that being able to look deeper in the queue would allow for better scheduling performance. It is thought that size of the observation space was causing difficulty with the SB3 implementation of PPO with invalid action masking. The size of the observation space for our implementation was as follows:

$$Observation\ Space = queue\ depth * num\ HPC\ machines$$

As the queue depth, or the number of HPC machines increased, so did the size of the observation space. Beyond a certain point queue depth of HPC size, the sum of the probabilities of actions exceeded a threshold set in SB3. This challenge might be overcome with another implementation of PPO with invalid action masking, but it bears further investigation.

To build the observation space also requires maintaining a list of schedulable jobs. This requires iterating through each machine in the cluster for every job in the job queue to see if any machine currently has adequate resources to schedule a particular job. This operation has $O(n*m)$ time complexity, where n was the queue depth and m was the number of machines in the cluster, significantly slowed down the speed of scheduling using RL. This operation could be sped up significantly if the RL agent were first trained to choose only schedulable jobs from the queue and then given the task to choose a schedulable job from the queue and machine on which to run it. Scheduling using BFBP requires the same $O(n*m)$ operations to make its decisions, but with further algorithmic refinement, scheduling using RL could not only perform comparably to scheduling with algorithmic baselines in terms of HPC metrics, but it also would be faster and more performant. The speed of making scheduling decisions was outside the scope of what was investigated with this research, but it's likely that with some algorithmic optimizations, scheduling with RL could be faster than these algorithmic baselines as well.

Some scheduling applications, like Slurm, allow for custom user-provided plugins for scheduling. It would be of great interest to explore how a RL powered scheduling plugin for Slurm would perform on an actual or simulated HPC system. Much integration work would be required, but we believe this work demonstrates the viability of HPC scheduling with RL and moves it one step closer to an actual implementation on an HPC system.

VI. Conclusions

This work showed PPO with invalid action masking can perform better than or comparable to algorithmic scheduling baselines when scheduling for HPC systems. The RL agent was able to select not only the next job from the job queue, but also the machine on which to run it all while accommodating multiple resource constraints on each machine. Also, within the discrete event simulator, multiple jobs could execute simultaneously on any machine in the simulated cluster while still respecting the machine's resource constraints. The code used to conduct this research has been released under the GPL v3.0 license should others find it useful.

References

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. ArXiv Preprint ArXiv:1606.01540.

Chapin, S., Cirne, W., Feitelson, D., Jones, J., Leutenegger, S., Schwiegelshohn, U., . . . Talby, D. (1999). Benchmarks and standards for the evaluation of parallel job schedulers. Job Scheduling Strategies for Parallel Processing: IPPS/SPDP'99Workshop, JSSPP'99 (pp. 67-90). San Juan: Springer Berlin Heidelberg.

Corbalan, J., & D'Amico, M. (2021). Modular workload format: Extending SWF for modular systems. Workshop on Job Scheduling Strategies for Parallel Processing (pp. 43-55). Cham: Springer International Publishing.

Dósa, G., & Sgall, J. (2014). Optimal analysis of best fit bin packing. International Colloquium on Automata, Languages, and Programming (pp. 429-441). Berlin: Heidelberg: Springer Berlin Heidelberg.

Girden, E. R. (1992). ANOVA: Repeated measures (No. 84). Sage.

Huang, S., & Ontañón, S. (2020). A closer look at invalid action masking in policy gradient algorithms. arXiv preprint arXiv:2006.14171.

Jette, M., & Grondona, M. (2003). SLURM: Simple Linux Utility for Resource Management. Proceedings of ClusterWorld Conference and Expo. . San Jose, California.

Liang, S., Yang, Z., Jin, F., & Chen, Y. (2020). Data centers job scheduling with deep reinforcement learning. Advances in Knowledge Discovery and Data Mining: 24th Pacific-Asia Conference, PAKDD 2020 (pp. 906-917). Singapore: Springer International Publishing.

Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource management with deep reinforcement learning. Proceedings of the 15th ACM workshop on hot topics in networks, (pp. 50-56).

Raffin, A., Hill, A., Gleave, A., A., K., Ernestus, M., & Dormann, N. (2021). Stable-Baselines3: Reliable Reinforcement Learning Implementations. Journal of Machine Learning Research, 22, pp. 1-8. Retrieved from http://jmlr.org/papers/v22/20-1364.html

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.0634.

Student. (1908). The probable error of a mean. Biometrika, (pp. 1-25).

Ullman, J. D. (1975). NP-complete scheduling problems. Journal of Computer and System sciences, 384-393.

Wang, Q., Zhang, H., Qu, C., Shen, Y., Liu, X., & Li, J. (2021). RLSchert: an hpc job scheduler using deep reinforcement learning and remaining time prediction. Applied Sciences, (p. 9448).

Zhang, D., Dai, D., He, Y., Bao, F. S., & Xie, B. (2020). RLScheduler: an automated HPC batch job scheduler using reinforcement learning. SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-15). IEEE.