MRSch: Multi-Resource Scheduling for HPC

Boyang Li*, Yuping Fan*, Matthew Dearing*, Zhiling Lan*, Paul Rich†, William Allcock†, Michael Papka†,‡

**Department of Computer Science, Illinois Institute of Technology, Chicago, IL, USA*
{bli70, yfan22, mdearing}@hawk.iit.edu, lan@iit.edu

† *Argonne National Laboratory, Lemont, IL , USA*
richp,allcock,papka@anl.gov

‡ *Northern Illinois University, IL, USA*

Abstract—Emerging workloads in high-performance computing (HPC) are embracing significant changes, such as having diverse resource requirements instead of being CPU-centric. This advancement forces cluster schedulers to consider multiple schedulable resources during decision-making. Existing scheduling studies rely on heuristic or optimization methods, which are limited by an inability to adapt to new scenarios for ensuring long-term scheduling performance. We present an intelligent scheduling agent named MRSch for multi-resource scheduling in HPC that leverages direct future prediction (DFP), an advanced multi-objective reinforcement learning algorithm. While DFP demonstrated outstanding performance in a gaming competition, it has not been previously explored in the context of HPC scheduling. Several key techniques are developed in this study to tackle the challenges involved in multi-resource scheduling. These techniques enable MRSch to learn an appropriate scheduling policy automatically and dynamically adapt its policy in response to workload changes via dynamic resource prioritizing. We compare MRSch with existing scheduling methods through extensive trace-base simulations. Our results demonstrate that MRSch improves scheduling performance by up to 48% compared to the existing scheduling methods.

Index Terms—cluster scheduling; multi-resource scheduling; direct future prediction; reinforcement learning

I. INTRODUCTION

The cluster scheduler, also known as a batch scheduler, plays a critical role in high-performance computing (HPC), with the responsibility of determining the order in which jobs are executed. Existing cluster schedulers are CPU-centric. However, exponential growth in computing power has enabled HPC systems to tackle much more complex scientific problems. These emerging workloads have diverse resource requirements beyond the CPU. For example, I/O intensive applications can take advantage of a burst buffer with dramatically improved performance [1]. For these applications, raw CPU power is not necessarily the primary resource that determines performance, but the allocation with respect to fast storage is more crucial. Such a change requires the scheduler to consider *multi-resource scheduling* where the scheduling problem is to optimize the use of multiple schedulable resources, e.g., CPU, burst buffer, power, and so on.

Existing multi-resource scheduling methods often rely on heuristics [2], [3], [4], [5]. Among them, dominant resource fairness (DRF) [2] and Tetris [3] are widely cited. While these heuristics have been demonstrated to be effective for the workloads in data centers, they are not suitable for multi-resource scheduling in HPC because these two communities adopt different computing modes and target very different workloads. For example, in DRF, each job consists of multiple tasks, and the scheduling process is to determine the proper number of tasks per job (i.e., a malleable job) to maximize the minimum dominant share among jobs. In contrast, HPC is dominated by rigid parallel jobs with a fixed number of tasks. A key feature of HPC scheduling is to improve resource utilization while preventing job starvation where large-sized or long-running jobs are perpetually held in a waiting queue.

A few research studies presented heuristic or classical optimization methods for multi-resource scheduling in HPC. Sun et al. discussed list scheduling and pack scheduling, both being proposed for scheduling moldable jobs [6], [7]. One variant of the list scheduling method extends first-come, first-serve (FCFS) to multi-resource scheduling [8]. While heuristic methods are fast, they cannot deliver an optimal solution to a scheduling problem. Optimization methods were also explored for multi-resource scheduling [9], [10], [11], [12], [13]. These methods formulate the scheduling problem into a single-objective or multi-objective optimization problem. Studies suggested that the optimization-based methods, especially the multi-objective optimization approach, result in better scheduling performance [13].

Recent efforts explored reinforcement learning (RL) for cluster scheduling [14], [15], [16]. Distinguishing from heuristic and optimization methods that concentrate on the *immediate effect*, reinforcement learning processes a sequence of decisions where each decision can impact the next. Through training, an RL agent learns to make an informative decision to optimize the *long-term* effect resulting from each scheduling decision as a sequence of actions (e.g., effects on the current and future resource utilization) [17]. Moreover, a common drawback of heuristics and optimization methods is the *lack of adaptation*. An intriguing feature of RL is its ability to *adapt its actions* automatically to dynamic changes in workloads or system states. As such, RL offers a promising direction for improving cluster scheduling. Also, existing RL-driven scheduling techniques mainly concentrate on single-resource scheduling.

In this study, we suggest that *multi-objective reinforcement learning (MORL)* is a natural approach for multi-resource

Job	Resource A	Resource B	Runtime (h)
J1	70%	10%	1
J2	50%	60%	1
J3	30%	30%	1
J4	40%	10%	1

(a) Job waiting queue.

Solution	Job scheduling order	Makespan (h)
Fixed-priority method	(J2, J3) → (J1) → (J4)	3
Ideal method	(J1, J3) → (J2, J4)	2

(b) Scheduling decisions by different methods.

Fig. 1: An example illustrating the limitation when fixing the priority of each objective for job scheduling.

scheduling. A simple approach may extend existing RL-driven scheduling algorithms to multi-resource scheduling by using a scalar reward with a fixed priority per resource (e.g., assigning a weight of 50% per resource for two-resource scheduling). However, such an extension has a serious drawback. The following example illustrates the limitation of the fixed priority method. Consider a two-resource scheduling scenario, denoted by Resources A and B, with an empty initial system state. Four jobs in a waiting queue have a different resource demand expressed by the percentage of the system resource capacity needed, as shown in Figure 1(a). When using a fixed weight method (e.g., equally maximize the utilization of Resources A and B), the job scheduling order for this fixed weight method is (J2, J3), (J1), and (J4). As a result, the makespan (i.e., the time spent to complete all jobs) is three hours. However, the ideal job scheduling order should be (J1, J3) followed by (J2, J4), which results in a shorter makespan of only two hours. This example demonstrates that statically weighting multiple resources fails to provide an efficient scheduling.

Inspired by recent advancements in MORL, we investigate a new approach for multi-resource scheduling by developing an intelligent scheduling agent named *MRSch*. Our design leverages an advanced MORL method called *direct future prediction (DFP)* [18], which was proposed by Intel for gaming. Similar to classic RL methods, a trained DFP agent can make an intelligent decision by considering the long-term effect of each scheduling decision in a sequence of actions. Distinguishing from classical RL using a scalar reward, DFP dynamically prioritizes each objective at runtime after being properly set up with a goal vector (§III-B). Such an adaption is essential for dynamic resource changes experienced in multi-resource scheduling scenarios.

While the inherent advantages of DFP are appealing, it cannot be directly applied to our problem, and several technical challenges must be overcome. The design of MRSch contains several core components that address these challenges. First, the original DFP algorithm uses an image input to encode each frame of a video game. Previous scheduling studies for data centers also used a fixed-size 2D image for encoding job and system information (i.e., one dimension for resource availability and the other for time duration) [14]. Unfortunately, an image-based state representation is not appropriate for HPC scheduling. Unlike the tasks in data centers with a fixed time duration, HPC jobs may take seconds, days, or weeks to complete. As such, image-based encoding cannot effectively address this wide range of job duration. Instead, MRSch incorporates *a vector-based encoding mechanism* for effectively representing user jobs and system resources in a scalable way (§III-A). Second, a convolutional neural network (CNN) [19] is used in DFP for information processing. While CNNs are suitable for spatial data, user jobs and system states do not contain many spatial relationships. Therefore, we adopt the *multilayer perceptron (MLP)* [20] in the design of MRSch. Third, an essential input to drive DFP is *construction of a goal vector* that dynamically captures the relative priority of different resources. The design of MRSch uses a simple yet effective technique to automatically adjust the weight of each resource preference so as to pay more attention to the highly demanding resource by user jobs (§III-B). Fourth, given the unique characteristics of HPC workloads, advanced resource reservation and backfilling are common features required for HPC for preventing job starvation and improving resource utilization. MRSch incorporates these *HPC domain-specific techniques* by deploying a window-based reservation (§III-C). Finally, an efficient training strategy is leveraged by MRSch for fast convergence (§III-D).

Implemented in TensorFlow [21], we evaluate MRSch by extensive trace-based simulations with real-world job traces collected from the Theta machine at the Argonne Leadership Computing Facility (ALCF) [22]. To extensively evaluate MRSch under various resource confliction and saturation environments, we generate a series of workloads from these real traces that encompass a range of workloads. We compare MRSch with heuristic, classical optimization, and an extension of a single-objective RL.

We consider a setup where an HPC system has up to R schedulable resources. For simplicity, we initially restrict our focus to two schedulable resources: CPU and burst buffer. Following this setting, a case study is presented to show that MRSch can be readily extended to additional schedulable resources. Our experiments conclude that MRSch outperforms the existing methods by up to 48% with respect to overall scheduling performance.

II. RELATED WORK AND BACKGROUND

A. Related Work

On typical HPC clusters, cluster scheduling is responsible for allocating resources and determining the order in which jobs are executed. When submitting a job, a user is required to provide the resources required by the job and an estimate of job runtime. Submitted jobs are stored and sorted in a waiting queue based on the facility's prioritization policy. The scheduler then determines *when* and *where* to execute these queued jobs [24].

TABLE I: Comparison of MRSch with existing multi-resource cluster scheduling methods.

Methods / Features	Heuristics [8], [7], [2], [3], [6]	Classical optimization [9], [10], [11], [12], [13]	Existing RL-driven scheduling [16], [15], [23], [14]	MRSch
Long-term scheduling effect	×	×	√	√
Automatic policy tuning	×	×	√	√
Dynamic resource prioritizing	×	×	×	√
Training requirement	×	×	√	√

Unlike scheduling in data centers, HPC scheduling has several salient features. In particular, HPC is dominated by tightly-coupled parallel applications. Hence, *advanced job reservation* and *backfilling* are commonly used for preventing job starvation and improving resource utilization [8], [24]. Job reservation holds resources for the job at the head of the waiting queue to prevent starvation. Backfilling enables subsequent jobs to move ahead to utilize free resources appropriate for that job. A widely used strategy is EASY backfilling, which allows short jobs to skip ahead in the queue only if they do not delay the current job waiting at the head of the queue [8].

Considerable studies have been conducted to improve cluster scheduling by leveraging machine learning. For instance, one active topic is forecasting job characteristics or user behaviors to improve cluster scheduling, such as reported in [25] with a summary of the challenges and limitations of applying machine learning for job characteristic prediction. Distinguishing from this research, in recent years several pioneering studies explored reinforcement learning for HPC scheduling (i.e., sequential decision making). For example, RLScheduler deployed a new kernel-based neural network structure and trajectory filtering mechanism to stabilize the learning process [15]. MARS combined heuristics and a deep RL actor-critic algorithm to optimize HPC systems for legacy and complex workflows [23]. DRAS leveraged a hierarchical neural network that incorporates HPC-specific scheduling features [16]. These studies targeted CPU-only scheduling.

For multi-resource scheduling, heuristic methods are commonly used, such as co-scheduling CPUs and memory in data centers [2], [3], [26], [5]. Among them, dominant resource fairness (DRF) and Tetris are well-known methods [2], [3]. DRF adopts a max-min fairness algorithm for the dominant resources to ensure that no user is better off if the resources, such as CPU and memory, are equally partitioned among them [2]. Tetris presents a multi-dimensional bin packing method that improves the average job completion time by preferentially serving jobs that have less remaining work compared to other jobs [3]. These studies targeted typical workloads seen in data centers with jobs composed of multiple tasks and scheduling decisions designed to determine how many tasks for each job should be selected.

Unfortunately, these techniques are not suitable for multi-resource scheduling in HPC for two reasons. First, the scheduling objective in HPC is to optimally schedule jobs in the waiting queue (instead of tasks within the jobs, as in data centers). Second, large-sized, long-running rigid jobs are common in HPC, and preventing their starvation in the waiting queue is a crucial scheduling requirement.

Existing multi-resource scheduling approaches in HPC can be broadly classified as either heuristics- or optimization-based methods. In list scheduling [6], [7], jobs are first organized in a priority list and assigned in sequence to the earliest available resources. An extension of FCFS to multi-resource scheduling is an instance of list scheduling. Classical optimization methods have also been considered for multi-resource scheduling [9], [10], [11], [12]. Yuping et al. [13] developed a multi-resource scheduling algorithm to explore a Pareto set for decision-making. Heuristic and optimization methods are similar in that decisions are made for the best immediate effect, such as maximizing resource utilization at the decision-making moment. However, considering only immediate consequences may lead to suboptimal performance in the long term.

MRSch differs from these prior studies in multiple aspects, as summarized in Table I.

B. Direct Future Prediction

Direct future prediction (DFP) is an advanced MORL algorithm developed in 2017 [18]. Its foundational idea is to train an agent *to predict the effect of different actions on future measurements, conditioned by the present state input, measurements, and goal.* DFP inherits the long-term scheduling impact of traditional reinforcement learning. Distinct from conventional RL with feedback as a scalar reward, feedback in DFP is in the form of a measurement (*a vector*). Leveraging this extension, unlike traditional RL methods that learn a single objective according to a scalar reward, DFP can *switch* goals (i.e., the product of the measurement and goal vector) under various circumstances. This switching is performed by dynamically adjusting the goal vector.

DFP incorporates three input modules, each processing an image s (i.e., a perception module), measurement m, and goal g (i.e., reflecting the relative importance of each measurement) separately. The pursued objective can be expressed as a dot product of the predicted measurement change and goal vector. The outputs of these modules are concatenated into a joint representation j that is processed by two parallel streams, an expectation stream and a normalized action stream, inspired by the dueling architecture introduced by DeepMind [27]. These two streams are combined to produce a final prediction for each action. More details of DFP can be found in [18].

The DFP agent interacts with the environment to obtain the actual measurement change. The loss function between
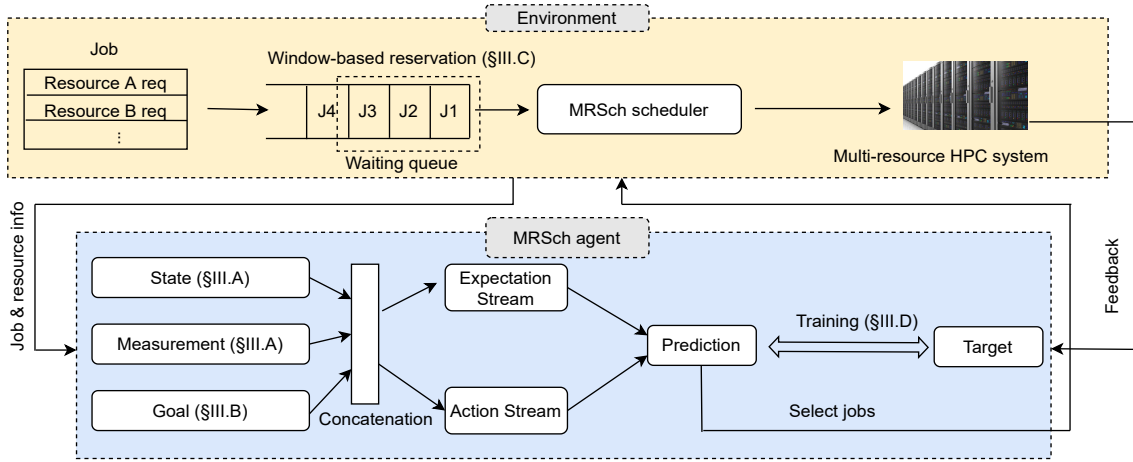
Fig. 2: Overview of MRSch. The environment (the top portion) denotes the HPC multi-resource scheduling system. The MRSch agent (the bottom portion) contains three input modules (state, measurement, and goal) and interacts with the environment by observing environmental changes and making scheduling decisions (i.e., selecting jobs for execution). The arrows between the agent and the environment indicate the information flows between them.

this measurement and the predicted measurement is used to train the neural network. During training, the agent follows an ϵ-greedy policy to avoid local optimums. During testing, the agent selects the action that yields the best-predicted outcome.

III. MRSCH DESIGN

MRSch represents the scheduler as an intelligent agent that makes decisions for when and which jobs should be allocated to available resources (Figure 2). *The environment* includes job and resource information, along with system measurements, such as resource utilization. The objective of the MRSch agent is to maximize the utilization of each resource by taking the actions of selecting jobs for scheduling. Because resource scarcity dynamically changes, the weight per resource, represented by the *goal* module, must adapt to dynamic environmental changes for optimizing job selection.

The MRSch agent interacts with the environment over discrete scheduling instances. At a given instance, the agent reads the job and resource information as input for the *state* and *measurement* modules. The input of the goal module represents the weights of each measurement from the measurement module. The outputs of these three modules are concatenated into a joint representation that is processed by the parallel *expectation stream* and *action stream*. The outputs of these streams are combined to produce a final *prediction* of future measurements for each action. The agent then takes an action by selecting jobs from the waiting queue and obtains the actual future measurement (the *target* module) fed back by the system. MRSch trains the neural networks to improve the prediction accuracy of future measurements for each action by minimizing a loss function between the prediction and target. Key techniques designed into MRSch are described below.

A. Input Modules

The foremost challenge is formulating the specific HPC multi-resource scheduling problem as MORL. In the follow-ing, we describe our representations of the input modules featured in Figure 2.

State. In the original DFP, the input of the state module is an image [18]. Encoding job and resource information as an image is not suitable for our case because it is difficult to capture critical job information (e.g., job waiting time) in images. Instead, we encode the job and resource information as *vectors*. Each waiting job is encoded as a vector of $(R+2)$ elements, where R is the number of resources requested by the job, and the additional elements correspond to the user-supplied estimated runtime and queued time of the job.

For system resources, we encode each resource unit as a vector of two elements. The first is a binary value representing resource availability (1 means available and 0 means not available). If the resource is occupied, then we take the user-supplied runtime estimate and job start time to calculate this unit's estimated available time. The second element is the time difference between the resource unit's estimated available time and the current time. If the resource is available, then we set this element to zero. The resource unit can be defined by the system administrator, e.g., a node for the CPU resource or a TB burst buffer as the unit for the burst buffer resource. Finally, we concatenate job information and resource information into a fixed-size vector as the input for the state module.

Rather than using CNN within the state module as deployed in the original DFP, we incorporate a multilayer perceptron (MLP). CNN works well on data with spatial relationships, such as image data [28]. However, the features of our state in-put are independent. We show experimental results comparing MLP and CNN architectures in Section §V-A.

We also use one neural network for all resources instead of one neural network per resource. This design choice is based on two reasons. First, more training parameters are available for the state module with a single neural network

TABLE II: Symbols and their descriptions.

Symbol	Description
N	number of jobs in the system.
R	number of resources in the system.
t_i	user-supplied runtime estimate of job i in waiting queue, remaining runtime estimate of job i running on system.
P_{ij}	percentage of requested resource j, (divided by the system resource j capacity) for job i.
r_j	goal vector value reflecting the contention fierceness of resource j by all jobs, including running and queued.

configuration compared to separate neural networks. Second, if using multiple neural networks, job information would be encoded multiple times in the final joint representation, resulting in an inefficient redundancy.

Our state neural network consists of four layers, including the input layer, two fully-connected layers, and output layer. The input layer is connected to two fully-connected layers activated by a leaky rectifier [29], and the second fully-connected layer is connected to the output layer.

Measurement. The inputs to the measurement module are the metrics of the scheduling objective. Different HPC facilities may have different scheduling objectives. A typical objective is to maximize the utilization for all schedulable resources. Suppose two types of resources, Resource A and B, are available, and the site objective is to maximize the utilization of both resources. A measurement vector is defined as <*Resource A util*, *Resource B util*>, and a three-layer fully-connected network parses the measurement module.

Goal. The values of the goal vector determine the weights of each measurement in the overall scheduling objective. Positive values correspond to maximizing the particular measurement, and negative values correspond to its minimization. Configuring the goal vector is described in the next subsection.

Action. MRSch deploys a window to specify a range of jobs to select from within the waiting queue. Intuitively, the scheduler can select multiple jobs within this window simultaneously, but this could result in an explosive number of actions. Instead, MRSch decomposes a scheduling decision that includes several jobs in one action into a series of individual job selections.

B. Dynamic Resource Prioritizing

The fierceness of contention for each resource changes during multi-resource scheduling, so more consideration should be assigned to the more contentious resource. Therefore, dynamically adjusting the resource priority is essential.

In MRSch, dynamic resource priority is achieved by adjusting the *goal vector* input to the goal module, g, that represents the weights of each measurement in the overall scheduling objective. A larger value of the goal vector means the corresponding measurement plays a more important role in the scheduling objective. MRSch gives preference to the resource with more fierce contentions.

Suppose there are R schedulable resources and the scheduling objective is to maximize resource utilization (Table II lists all symbols and their corresponding meanings). MRSch sets the values in the goal vector as follows:

$$r_j = \frac{\sum_{i=1}^{N} P_{ij} t_i}{\sum_{j=1}^{R} \sum_{i=1}^{N} P_{ij} t_i} \quad (1)$$

Equation (1) describes how long (normalized) it will take to complete all the jobs' resource j demands in the ideal situation where resource j is fully utilized. A longer time represents a more fierce resource contention.

C. Avoid Job Starvation

HPC job sizes and runtimes can span broad scales in practice. A job size ranges from a single node to the entire HPC system comprised of thousands of compute nodes, and its runtime may vary from seconds to days. Such a variety in job characteristics presents a unique challenge for HPC scheduling: queued jobs, especially large-sized jobs, tend to be starved when small-sized jobs continue arriving into the queue and skip to the front while insufficient resources are available for the larger job. Directly applying DFP to the multi-resource scheduling problem results in severe job starvation.

MRSch adopts two techniques to overcome this challenge. First, a window-based design alleviates job starvation by providing higher priority to older jobs in the queue. Second, MRSch inherits the reservation strategy. At a given scheduling instance, the scheduler enforces a window at the front of the waiting queue. When MRSch selects a job from this window, if its requested resources are available, then it is marked as ready and sent for immediate execution on the system. This process repeats until the system no longer has sufficient available resources for the next job selected by the agent. This next job is then marked as reserved so that its requested resources will be held for its execution on the system at the earliest available time. In addition, EASY backfilling is leveraged to improve resource utilization.

D. Training Strategy

To obtain a converged and accurate model for scheduling, the MRSch agent must gain experience through training from a large quantity of jobs with various job arrival patterns and diverse job characteristics. We train our MRSch agent with real workloads, along with sampled and synthetic workloads, to increase its robustness toward workload changes.

We follow the principle of gradual improvement to learn a robust model. MRSch begins with common represented cases and incrementally improves its capability with unseen rare cases. In particular, three types of job sets and a three-phase training process are employed to train MRSch in the following order: a set of sampled jobs from real job traces, real job traces, and synthetic jobs generated to represent previously unseen patterns. The sampled job sets have controlled job arrival rates that provide the easiest learning environment for MRSch to learn good scheduling decisions within a controlled environment. Subsequent training on real job traces with varying job arrival patterns enables MRSch to learn more complex scenarios. The final phase includes synthetic job sets to tune MRSch with experiences from a broader variety of

TABLE III: Workloads based on the production traces, representing light to heavy contention for the burst buffer.

Workload	Number of requested nodes	Percentage of jobs requesting burst buffer	Burst buffer size range
S1	number of requested nodes in the trace	50%	[5 TB, 285 TB]
S2	number of requested nodes in the trace	75%	[5 TB, 285 TB]
S3	number of requested nodes in the trace	50%	[20 TB, 285 TB]
S4	number of requested nodes in the trace	75%	[20 TB, 285 TB]
S5	half of number of requested nodes in the trace	75%	[20 TB, 285 TB]

potential states that may have not been seen during the first two sets. Results comparing different training strategies are presented in §V-B.

IV. IMPLEMENTATION AND EVALUATION

MRSch is implemented in TensorFlow [21]. We evaluate MRSch through trace-based simulation using real workloads collected from a production system. In our experiments, MRSch interacts with CQSim, a trace-based HPC job scheduling simulator that has been used in various scheduling studies for a decade [30]. A real system processes jobs from user submissions, while CQSim imports jobs by reading the job arrival information from a trace. The simulator emulates system execution by advancing the simulation clock according to the job runtime recorded in the trace. Changes in the job wait queue or the system trigger the simulator to send scheduling requests to the MRSch agent. Typical triggers include the submission of a new job to the queue or a running job leaving the system.

For simplicity of presentation, we first confine our attention to two resources and later present a case study featuring more resources in §V-E.

A. Workload Traces

A variety of resources beyond CPUs may be considered as schedulable resources. Given that the burst buffer is widely deployed in production supercomputers [31], [32], we evaluate MRSch with the scheduling of CPU and burst buffer.

Our workload trace is a five-month historical job trace in 2018 from Theta at ALCF [33]. This trace only contains CPU request information, so we extend the data with burst buffer (BB) requests, assuming a shared burst buffer of 1.26 PB. To compensate for this lack of burst buffer information in the trace, a corresponding Darshan [34] trace extracts the amount of data moved between compute nodes and the parallel file system, which is then considered as the potential burst buffer request for each job. During the five months, 40% of the jobs have Darshan I/O records, and 17.18% have more than 1 GB of data transferred. The amount of transferred data is assigned as the corresponding job's burst buffer request, with a range of requested burst buffer sizes between 1 GB to 285 TB. A limitation is that the burst buffer was not heavily utilized during the time of this historical trace because the burst buffer was a relatively new resource, and not all applications had been refactored to benefit from this new feature. Also, some jobs did not include Darshan I/O recordings.

We extensively evaluated MRSch under various configurations, including cases of resource contention for either the CPU or burst buffer, by generating five synthetic workloads

from the original trace (Table III). These designed workloads represent *light to heavy contentions* for the burst buffer. The assigned burst buffer request is randomly selected from the original requests within a certain range. Those greater than 5 TB are randomly assigned to S1 or S2, while S3 and S4 select from requests greater than 20 TB. Compared to S1 and S2, S3 and S4 have larger burst buffer requests. S1 and S2 have similar distributions, but more jobs in S2 include burst buffer requests. A similar pattern is observed in S3 and S4. The S5 workload is generated by reducing the requested number of nodes from S4 by half to represent workloads with less CPU resource contention.

We split the five-month log into three parts: the first three and a half months of the workload for *agent training*, a subsequent two weeks of the workload for *model validation*, and the remaining data for *inference/testing*.

B. Evaluation Metrics

The quality of the scheduling method must be evaluated by multiple metrics, including both system-level and user-level metrics. Four well-established metrics are used to evaluate MRSch, where the first two are system-level metrics and the last two are user-level metrics.

1) *Node utilization*: the ratio of the used node-hours during useful job execution to the elapsed node-hours.
2) *Burst buffer utilization*: the ratio of the used burst buffer hours to the elapsed burst buffer hours.
3) *Average job wait time*: the average interval between job submission to job start time.
4) *Average job slowdown*: the average ratio of job response time (job runtime plus wait time) to the actual runtime, representing the responsiveness of a system.

C. Network Architecture

The input of the state neural network is a vector of size $[4W+2N_1+2N_2, 1]$, where W is the window size (10 in our experiment), N_1 is the number of compute nodes, and N_2 is the number of burst buffer units in the system. For the Theta machine, the input size of the state neural network is [11410, 1]. We use two fully-connected layers with 4,000 and 1,000 neurons, respectively, with an output layer of 512 nodes. A three-layer fully-connected network with 128 neurons parses the measurement and goal modules. The action space includes the waiting jobs in the window. MRSch selects the jobs from this window for job allocation to optimize the goal. The MRSch agent follows an ϵ-greedy policy to select jobs in the training time by acting greedily according to the current goal with probability $(1 - \epsilon)$ and selects a random action with probability ϵ. We set $\epsilon = 1.0$ at the beginning of the training,

(a) Node utilization   (b) Burst buffer utilization   (c) Average job wait time   (d) Average job slowdown
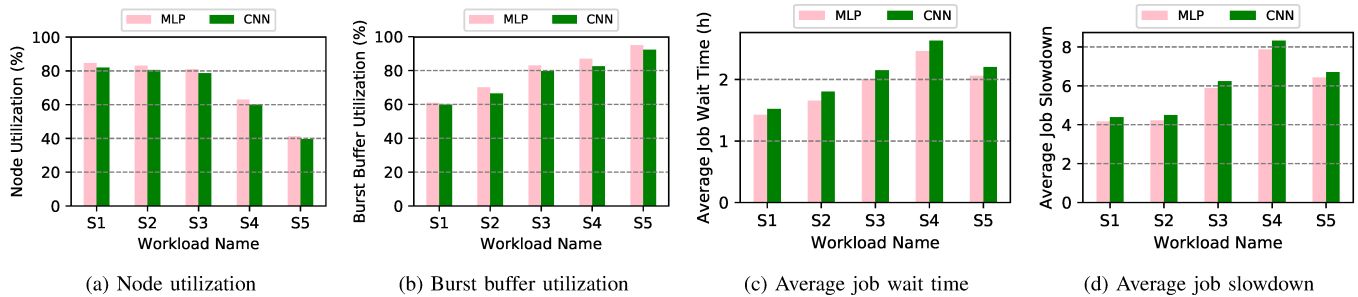
Fig. 3: Comparison of MRSch scheduling performance by using different state modules (MLP vs CNN) that indicates the use of MLP is more beneficial for multi-resource scheduling.
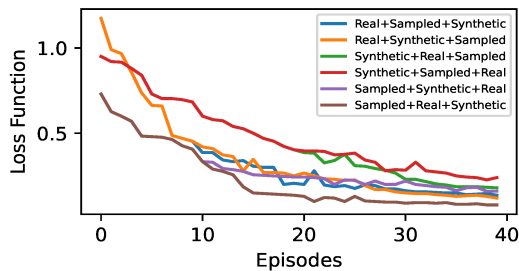

Fig. 4: Comparison of the quality and convergence of MRSch by training with different jobset orderings. The loss function is expressed by the mean squared error.

which then decays at a rate of $\alpha = 0.995$. During testing, the agent observes the environment and dynamically changes the weights in the goal vector according to the scarcity of resources calculated with Equation (1).

D. Comparison Methods

We compare MRSch with three scheduling methods:

- **Heuristic** is an extension of FCFS, belonging to the list scheduling family [7], for multi-resource scheduling where jobs are scheduled according to the arrival order into the waiting queue.
- **Optimization** denotes the method that formulates the multi-resource scheduling problem into a multi-objective optimization problem and solves the problem using a genetic algorithm [13]. For a fair comparison, we apply the same window size as in MRSch.
- **Scalar RL** represents a group of reinforcement learning methods that formulates multi-resource requirements into a scalar reward with a fixed weight. In our experiments, we use a policy gradient method [35] and the scalar reward is set to $(0.5 \times CPU\_util + 0.5 \times burst\ buffer\_util)$.

In addition, EASY backfilling is adopted in each of these methods to mitigate resource fragmentation [8]. The comparison study is performed with the trace-based, event-driven scheduling simulator CQSim [30].

V. RESULTS

We examine the scheduling performance of MRSch under different state module representations and various training strategies in §V-A and §V-B, and compare MRSch with existing multi-resource scheduling methods in §V-C. We also assess if MRSch can adapt to workload changes in §V-D. A case study with more schedulable resources is presented in §V-E. Finally, we list runtime overhead in §V-F.
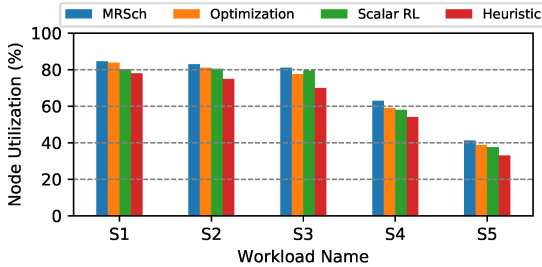
A. State Module: MLP vs CNN

For the state module described in §III-A, we use MLP instead of the CNN adopted in the original DFP. This set of experiments compares the scheduling performance under different state modules (MLP vs CNN), with results presented in Figure 3. The use of the MLP network achieves a better scheduling performance by up to 7% across the system-level and user-level metrics. CNN is good at spatially extracting local correlations present in the input data, e.g., local filters over the input [28]. However, the features of the MRSch state module input (e.g., the waiting job and running job information) are not spatially related. For processing input with independent features, MLP often provides a better solution [29], as we observe in this experimental scenario.
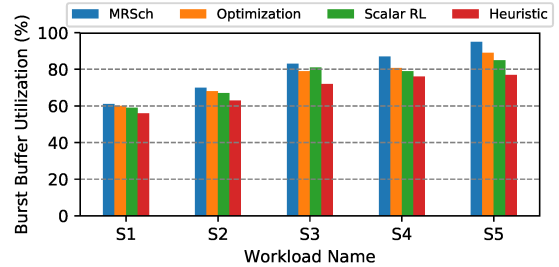
B. Training Strategy

We separate the training data into ten job sets and collect another ten job sets by randomly sampling jobs from the original training trace. The arrival times for these jobs are modeled as a Poisson distribution following the average inter-arrival time of the original trace. Also, we generate 20 synthetic job sets that mimic Theta workload patterns in terms of hourly and daily job arrivals, distributions of resource requests, and job runtimes. In total, we train our model with 40 job sets containing 200,000 jobs.

Figure 4 compares the convergence rates for the loss function by varying the order of the job sets during the training of MRSch. This set of experiments demonstrates that training with a sequence of sampled, real, and synthetic job sets (the brown curve in the figure) achieves the fastest convergence speed and the smallest mean squared error compared to the other trace orderings. This result confirms our initial intuition that it is advantageous for MRSch to first learn from simple, averaged cases (i.e., the sampled job sets) and then subsequently advance through more complex special cases (i.e., the real and synthetic job sets) to generate a converged and high-quality model.
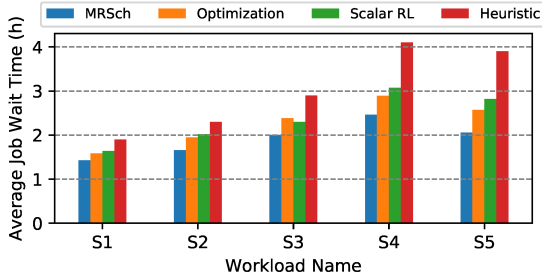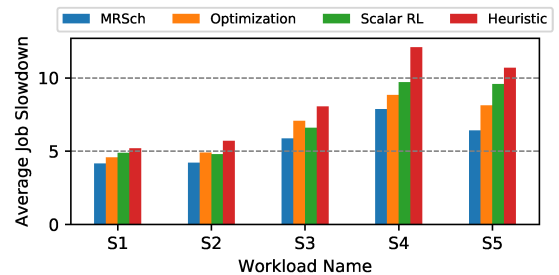
(a) Node utilization

(b) Burst buffer utilization

Fig. 5: Scheduling performance in terms of system-level metrics.

(a) Average job wait time

(b) Average job slowdown

Fig. 6: Scheduling performance in terms of user-level metrics.

C. Scheduling Performance

Figure 5 compares different scheduling methods in terms of the system-level metrics. MRSch yields the highest node and burst buffer utilization across the various workloads, whereas FCFS leads to the worst system performance. Scalar RL achieves better performance than the optimization method on S3, which we attribute to two reasons. First, compared to the other workloads, the CPU and burst buffer demands in S3 are relatively balanced. Second, the RL method offers better long-term effects due to its learning capability [16].

Among the five workloads, MRSch achieves a larger increase in resource utilization on S4 and S5. The contention fierceness for the burst buffer increases from S1 to S5, which indicates that MRSch attains higher performance gains when the resource demands are unbalanced and fierce.

Figure 6 compares different scheduling methods in terms of the user-level metrics. For all cases, MRSch achieves the best performance. We notice that average job wait time and slow down increase dramatically as the burst buffer requests increase. The most noticeable average wait time and slow-down reductions obtained by MRSch occur in the heavily unbalanced S4 and S5 workloads. In contrast, the scalar RL method does not perform well in these workloads, suggesting the importance of dynamic resource prioritization in response to the scarcity of the resources. These results also indicate that MRSch achieves better scheduling performance, especially in the cases of high demand from user jobs for a resource. Overall, MRSch delivers the best performance among all compared methods, highlighted by shortening the average job wait time by up to 48% and decreasing job slowdown by up to 41%.

Figure 7 presents Kiviat charts of the overall scheduling performance for each workload obtained by different scheduling methods. We plot the reciprocal of the average job wait time and the reciprocal of the job slowdown in these charts. All metrics are normalized within the range of $[0, 1]$, where 1 corresponds to a method that achieves the best performance among all others. In other words, a larger area outlined in the figure indicates a better overall scheduling performance for that method. MRSch consistently yields the best results, whereas FCFS delivers the worst performance across all workloads.

MRSch demonstrates its best improvements over the other methods in S5. We attribute this outcome to the heavily unbalanced contention for each resource compared to the S1–S4 workloads. Specifically, the burst buffer resource contention in S5 is more fierce than the CPU resource contention. MRSch delivering its strongest performance in this scenario suggests its robust capability to automatically change objectives within an unbalanced resource contention environment.

D. Adaption to Workload Change

To validate our observations, we examine r_{BB}, the goal vector value for the burst buffer as calculated by Equation (1). This value reflects the relative importance of the burst buffer to CPU during multi-resource scheduling. Figure 8 plots the dynamic changes of r_{BB} in the range of 0.6 to 0.9 when using MRSch in S5 during a randomly selected 12 hours.

Figure 9 presents box plots of r_{BB} in the S1–S5 workloads, suggesting that (1) r_{BB} dynamically changes compared to the fixed value of 0.5 in the scalar RL method, and (2) the

(a) S1 workload.   (b) S2 workload.   (c) S3 workload.

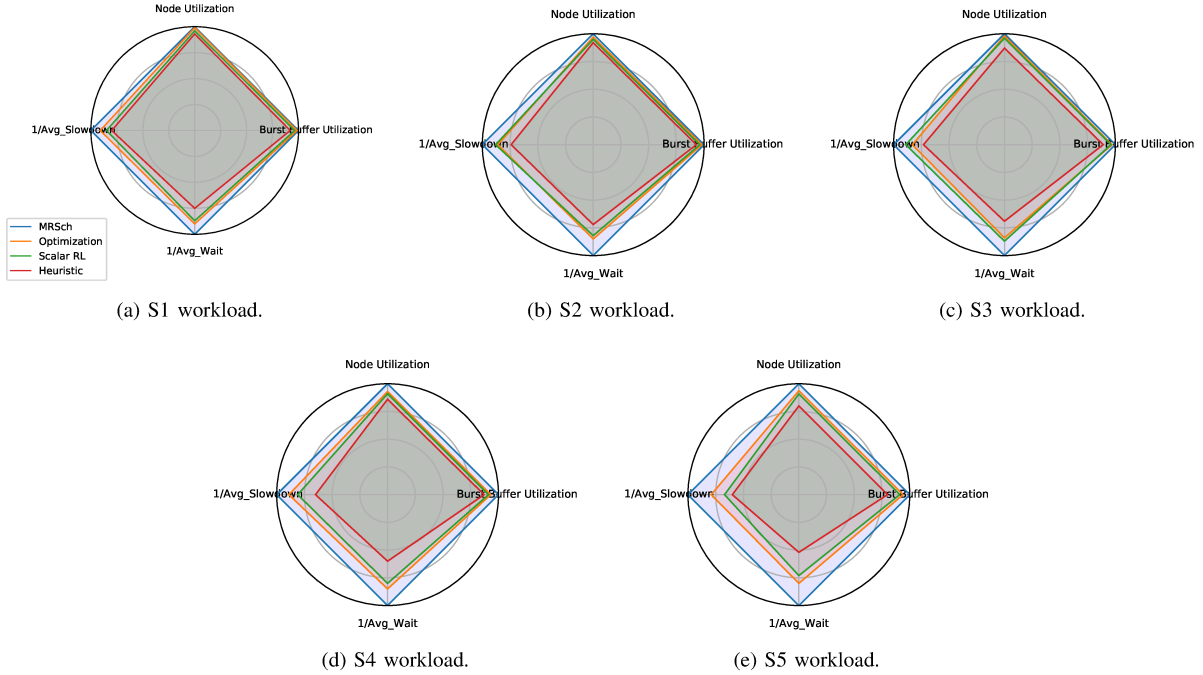(d) S4 workload.   (e) S5 workload.

Fig. 7: Comparison of MRSch with existing methods using Kiviat charts. The larger the area outlined by a curve, the better the overall performance of the corresponding method.
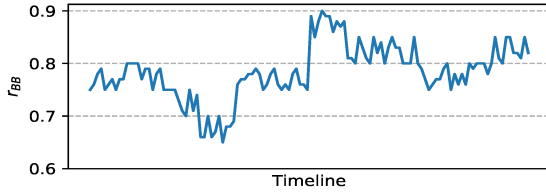

Fig. 8: The fluctuation of r_{BB} under the S5 workload within a randomly selected 12 hours, where r_{BB} is calculated by Equation (1). This value reflects the relative importance of burst buffer compared to CPU during multi-resource scheduling.
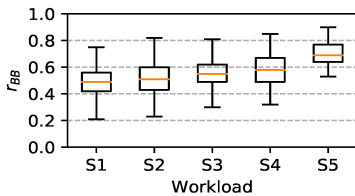

Fig. 9: Box plot of r_{BB} (i.e., the goal vector value corresponding to the burst buffer utilization) for the S1–S5 workloads.

minimum value, first quartile, mean value, third quartile, and maximum value are the largest for S5. These results validate that the MRSch agent automatically assigns more preference to the relatively scarce resources when it detects an unbalanced contention for each resource. However, in such a situation, the scalar RL method treats the CPU and burst buffer equally, which leads to poor scheduling performance.

E. Case Study: More Resources

MRSch is generally applicable for multiple schedulable resources. For instance, another schedulable resource could be power because the power consumption of supercomputers increases significantly. Aurora, the planned exascale supercomputer, anticipates a power budget of 60 MW [36]. Therefore, approaches for improving energy efficiency are attracting more attention in the HPC field, including several studies that explored power-aware scheduling [9], [37]. As a case study, we incorporate power as a third resource in addition to the CPU and burst buffer resources to illustrate how MRSch can be easily extended to incorporate more schedulable resources.

Consider a system with three schedulable resources of CPU, burst buffer, and power. A fixed power budget exists for the entire system, making power another resource for which jobs must contend. Each submitted job includes four pieces of information: walltime, requested number of nodes, requested volume of burst buffer, and a power profile (i.e., peak power consumption). This case study considers three objectives of maximizing the CPU/node utilization, maximizing the burst buffer utilization, and maximizing the total power consumption of running jobs within a power budget.

We generate five new workloads S6–S10 by creating power profiles for the jobs from S1–S5. For each job, its power consumption per node is randomly assigned between (100–215 W). The Theta computing nodes (Intel KNL 7230) have a 215 W thermal design power (TDP) [38], and 100 W is selected as the lower bound based on previous work [39]. The power consumption of an idle node is set to 60 W [40], and
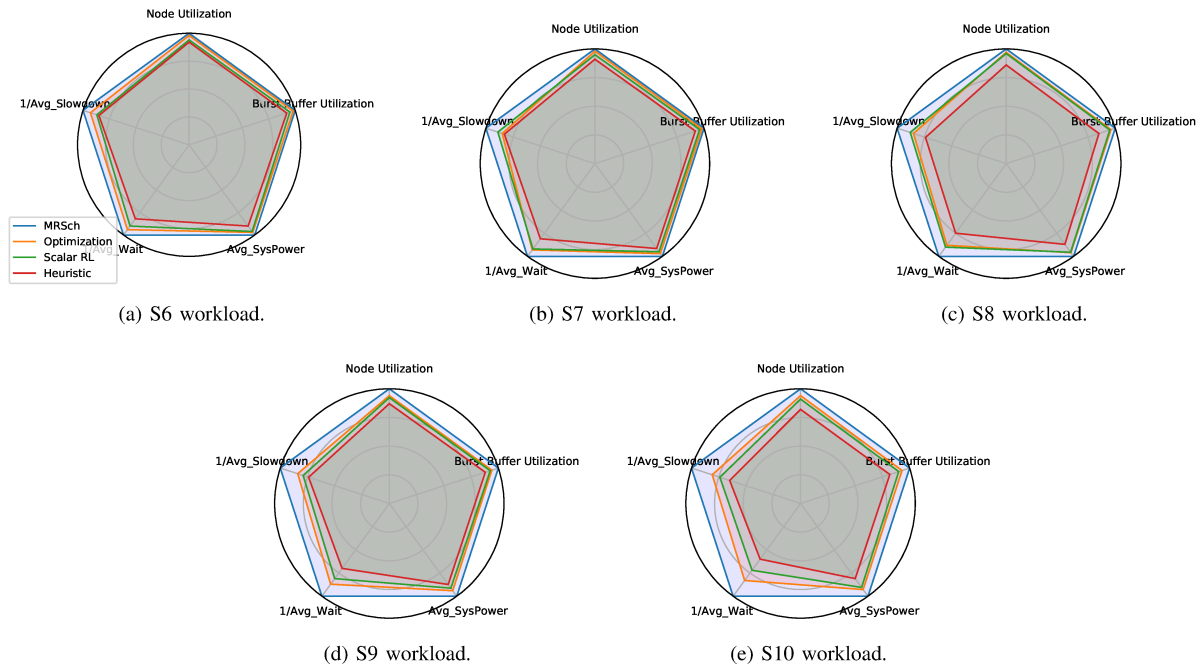
Fig. 10: Scheduling performance under more schedulable resources. The larger the area outlined by the curve, the better the overall performance of the corresponding method.

the power budget for the entire system is restricted to 500 kW to ensure a contentious environment.

Figure 10 presents a holistic view of the scheduling performance. We observe that MRSch achieves the best overall performance for all workloads, while the FCFS heuristic results in the worst overall performance on all workloads. Compared to the other methods, MRSch improves resource utilization by up to 18%, reduces the average wait time by up to 39%, and reduces the average slowdown by up to 34%, which demonstrate that MRSch can be generally applied to scheduling multiple resources. In summary, this case study demonstrates the effectiveness of MRSch for multiple schedulable resources.

F. Runtime Overhead

In our experiments, MRSch required less than two seconds to make scheduling decisions during the two-resource scheduling experiments and less than three seconds during the three-resource scheduling during testing. All experiments were performed on a personal computer configured with an Intel 2 GHz quad-core CPU and 16 GB memory. Current HPC systems typically require the scheduler to respond within 15–30 seconds [10]. Therefore, the MRSch agent imposes negligible overhead and is a feasible solution for online deployment in production systems.

VI. Conclusion

Motivated by the increasing need for multi-resource scheduling in HPC, we present MRSch, an intelligent multi-resource scheduling agent that leverages an advanced multi-objective reinforcement learning algorithm called DFP. While DFP features an inherent advantage for pursuing dynamically

changing objectives, it was initially designed for gaming and never previously applied to HPC scheduling. In this work, we describe our problem formulation and several key techniques are developed into MRSch for incorporating HPC-specific scheduling requirements. These techniques enable MRSch to automatically observe the HPC scheduling environment and adapt its policy to continuous workload and resource changes. Our experimental results show that MRSch outperforms existing scheduling approaches—heuristic, optimization, and scalar-based reinforcement learning methods—by up to 48% in terms of user-level and system-level metrics.

While MRSch demonstrates promising performance compared with conventional heuristic and optimization methods, a significant gap remains in deploying RL-based scheduling in production systems. One key hurdle is the lack of model interpretability. Because the scheduling agent is constructed on deep neural networks with millions or more parameters, it appears as a black box model to system managers, so is incomprehensible to debug, deploy, and adjust in practice [41]. Our future work includes investigating how to provide practical RL-driven scheduling systems with interpretable models.

References

[1] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X.-H. Sun, "Leveraging burst buffer coordination to prevent i/o interference," in *12th International Conference on e-Science*. IEEE, 2016.

[2] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *8th Symposium on Networked Systems Design And Implementation*, 2011.

[3] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Computer Communication Review*, 2014.

[4] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, 2014.

[5] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan, "Altruistic scheduling in multi-resource clusters," in *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[6] M. R. Garey and R. L. Graham, "Bounds for multiprocessor scheduling with resource constraints," *SIAM Journal on Computing*, 1975.

[7] H. Sun, R. Elghazi, A. Gainaru, G. Aupy, and P. Raghavan, "Scheduling parallel tasks under multiple resources: List scheduling vs. pack scheduling," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018.

[8] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 6, pp. 529–543, 2001.

[9] S. Wallace, X. Yang, V. Vishwanath, W. E. Allcock, S. Coghlan, M. E. Papka, and Z. Lan, "A data driven scheduling approach for power management on HPC systems," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE/ACM, 2016.

[10] X. Yang, Z. Zhou, S. Wallace, Z. Lan, W. Tang, S. Coghlan, and M. E. Papka, "Integrating dynamic pricing of electricity into energy aware scheduling for HPC systems," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE/ACM, 2013.

[11] A. Wierman, L. L. Andrew, and A. Tang, "Stochastic analysis of power-aware scheduling," in *2008 46th Annual Allerton Conference on Communication, Control, and Computing*. IEEE, 2008.

[12] S. Ren, Y. He, and F. Xu, "Provably-efficient job scheduling for energy and fairness in geographically distributed data centers," in *IEEE 32nd International Conference on Distributed Computing Systems*. IEEE, 2012.

[13] Y. Fan, Z. Lan, P. Rich, W. E. Allcock, M. E. Papka, B. Austin, and D. Paul, "Scheduling beyond cpus for HPC," in *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2019.

[14] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM workshop on hot topics in networks*, 2016.

[15] D. Zhang, D. Dai, Y. He, F. S. Bao, and B. Xie, "RLScheduler: an automated HPC batch job scheduler using reinforcement learning," in *SC'20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE/ACM, 2020.

[16] Y. Fan, Z. Lan, T. Childers, P. Rich, W. Allcock, and M. E. Papka, "Deep reinforcement agent for scheduling in HPC," in *Proceedings of the 35th International Parallel and Distributed Processing Symposium*. IEEE, 2021.

[17] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[18] A. Dosovitskiy and V. Koltun, "Learning to act by predicting the future," in *5th International Conference on Learning Representations*, 2017.

[19] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[21] MRSch on GitHub. [Online]. Available: https://github.com/SPEAR-IIT/MRSch

[22] Argonne Leadership Computing Facility (ALCF). [Online]. Available: https://www.alcf.anl.gov

[23] B. Baheri and Q. Guan, "Mars: Multi-scalable actor-critic reinforcement learning scheduler," *arXiv preprint arXiv:2005.01584*, 2020.

[24] W. Allcock, P. Rich, Y. Fan, and Z. Lan, "Experience and practice of batch scheduling on leadership supercomputers at argonne," in *Workshop on Job Scheduling Strategies for Parallel Processing*. IEEE, 2017.

[25] M. Kuchnik, J. W. Park, C. D. Cranor, E. Moore, N. Debardeleben, and G. Amvrosiadis, "This is why ml-driven cluster scheduling remains widely impractical," Carnegie Mellon University, CMU-PDL-19-103, Tech. Rep., 2019.

[26] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[27] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2016.

[28] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions," *Journal of big Data*, vol. 8, no. 1, pp. 1–74, 2021.

[29] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press, 2016.

[30] CQSim. [Online]. Available: https://github.com/SPEAR-IIT/CQSim

[31] Trinity. [Online]. Available: https://www.lanl.gov/projects/trinity/

[32] Cori. [Online]. Available: http://www.nersc.gov/users/computational-systems/cori/

[33] Theta. [Online]. Available: https://www.alcf.anl.gov/theta

[34] Darshan. [Online]. Available: https://www.mcs.anl.gov/research/projects/darshan/

[35] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International Conference on Machine Learning*. PMLR, 2014.

[36] Aurora. [Online]. Available: https://www.hpcwire.com/2021/09/08/how-argonne-is-preparing-for-exascale-in-2022/

[37] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini, "Scheduling-based power capping in high performance computing systems," *Sustainable Computing: Informatics and Systems*, vol. 19, pp. 1–13, 2018.

[38] TDP of KNL 7230. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/94034/intel-xeon-phi-processor-7230-16gb-1-30-ghz-64-core.html

[39] S. Sharma, Z. Lan, X. Wu, and V. Taylor, "A dynamic power capping library for HPC applications," in *Cluster Conference (2-page extended poster)*. IEEE, 2021.

[40] I. Marincic, V. Vishwanath, and H. Hoffmann, "PoLiMEr: An energy monitoring and power limiting interface for HPC applications," in *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, 2017.

[41] Z. Meng, M. Wang, J. Bai, M. Xu, H. Mao, and H. Hu, "Interpreting deep learning-based networking systems," in *ACM SIGCOMM*, 2020.