CuSH: Cognitive ScHeduler for Heterogeneous High Performance Computing System

Giacomo Domeniconi
giacomo.domeniconi1@ibm.com
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA

Eun Kyung Lee
eunkyung.lee@us.ibm.com
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA

Alessandro Morari
amorari@us.ibm.com
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA

ABSTRACT

Heterogeneous computing systems deliver high performance and flexibility while increasing system management complexity. Resource management is particularly affected, because of the variety of computing, memory, and storage resources to handle. This paper describes Cognitive ScHeduler (CuSH), a resource management and job scheduling framework that leverages deep neural networks and reinforcement learning. To handle the complexity of heterogeneous resource scheduling, CuSH employs a two-step hierarchical solution. This solution decouples the job selection from the policy selection problem. CuSH has been evaluated using a simulator that is based on experiments executed on an IBM Power® system. Results show that CuSH outperforms traditional heuristic-based approaches on all the use cases, delivering significantly lower normalized turnaround time.

CCS CONCEPTS

• Theory of computation → Reinforcement learning; **• Computing methodologies → Policy iteration**; **• Software and its engineering → Scheduling**.

KEYWORDS

High Performance Computing, Scheduling, Reinforcement Learning, Policy Gradient

ACM Reference Format:
Giacomo Domeniconi, Eun Kyung Lee, and Alessandro Morari. 2019. CuSH: Cognitive ScHeduler for Heterogeneous High Performance Computing System. In *Proceedings of DRL4KDD 19: Workshop on Deep Reinforcement Learning for Knowledge Discovery (DRL4KDD).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The shift to heterogeneous architectures in High Performance Computing (HPC) has considerably increased the complexity of resource management algorithms. The Summit and Sierra supercomputers[1] are examples of systems that combine multiple types of computing (CPU, GPU) and memory (DDR, HBM, SSD) devices to maximize performance.

Application performance can vary depending on the compute and communication patterns of the workload and on the allocated hardware resources. When scheduling parallel distributed applications, the job scheduler could take into account several factors, including data locality, the levels of parallelism and the frequency of collective synchronizations. The implementation of an accurate cost model for a job scheduler can be an intractable problem due to the number and dimension of the modeling variables (e.g., job dimensions, queue size, execution time, available resources, locality, etc.). To simplify the problem, heuristics such as First Come First Serve, Shortest Job First, or Dominant Resource Fairness are often used. The heuristic-based approach achieves reasonable results, but performance may vary significantly depending on many factors. Moreover, a single heuristic is rarely optimal for all cases [11].

In this paper, we propose an AI-enhanced resource management solution called Cognitive ScHeduler for HPC system (CuSH). CuSH employs deep neural networks (NNs) and reinforcement learning (RL) to achieve optimal performance. Essentially, CuSH learns to make optimal scheduling choices training a NN on a dataset that contains jobs history and performance characteristics. Building such a dataset with multiple workloads and hardware configurations involves executing many jobs and it is very time-consuming. Because of this we use an cluster environment simulator based on real execution times to generate the dataset.

Using a novel approach, we formulate the job scheduling problem in two steps: (i) selecting a job to execute among the Q waiting jobs and (ii) selecting an allocation policy among the P allocation policies. This approach is able to reduce the action space from $Q \times P$ to $Q + P$, incurring in a modest performance cost. The implementation is composed of a job selector and a policy selector module each with its own NN trained in a RL configuration.

In the job selector module (JSM), parallel jobs are represented as images and serve as the input of a convolutional neural network (CNN), similarly to the DeepRM approach [8]. The convolutional layer extracts the features of the job states represented in images. The CNN is then used in a policy gradient configuration to learn job selection with Monte Carlo simulations. The reward function is designed to minimize the normalized turnaround time (NTAT). The JSM selects the best job to schedule, then communicates it to the policy selector module. The policy selector module (PSM) uses a fully connected NN to learn the optimal allocation policy for each job selected by the JSM. It is also trained in a RL configuration and the reward function is designed to minimize the jobs service time. Figure 1 depicts the proposed solution and its components.

[1]Summit and Sierra are currently the fastest supercomputers in the world https://www.top500.org/lists/2018/11/

DRL4KDD, 2019, Alaska - USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This paper presents a novel resource management solution for heterogeneous compute systems that leverages CNNs and RL. The major contributions of this paper are

- A resource management solution that leverages two separated NNs for job selection and policy allocation.
- A scheduling state representation for distributed and heterogeneous computing systems.
- A reward function that dynamically adjusts based on application profiling.

2 BACKGROUND

An effective job scheduling algorithm can significantly improve high performance computing systems efficiency and utilization. Traditionally, high performance computing systems included sophisticated and highly-configurable job management software to optimize the system depending on the hardware characteristics and the workload [7, 12]. The recent trends towards heterogeneous architectures has increased the complexity of the scheduling and resource management problem, adding additional resource types such as GPUs. Several works studied scheduling algorithms for heterogeneous systems, focusing on the optimization of various system metrics [5, 16, 17].

With the renewed interest in learning-based approaches, several authors have found that machine learning can be used effectively to obtain optimal scheduling. In particular, RL [15] has been shown to deliver better results with respect to traditional approaches [9, 18].

DeepRM [8], a multi-resource cluster scheduler based on RL, is the work that most inspired our approach. Its main idea is to model the cluster scheduling problem with an image-like state representation, enabling the use of CNN and RL. In a HPC system, a state can be seen as the current allocation of resources along with the request profiles of jobs waiting to be scheduled. DeepRM represents a state as a set of two-dimensional matrices that are equivalent to images, as shown in Figure 3. The y-axis of the image represents the time dimension (each row is one of the T timesteps) and the x-axis represents the hardware resources. A set of matrices represents the current resource allocation in the cluster, and another set of matrices represents the resource requirements by waiting jobs in the queue. The modeling of different resources is entrusted to separate matrices, having a matrix for each resource r (for instances CPU, GPU, memory, etc.) with the number of columns S^r as the availability of resources in the cluster. At each timestep, assuming M waiting jobs, the scheduling task is the definition of a subset of the jobs to be scheduled. This results in an intractable action space of size 2^M. DeepRM solves this issue (i) reducing the queue size to Q ($Q = 10$ in their experiments) while leaving the information of any other jobs in a backlog, and (ii) allowing the agent to execute multiple actions at each timestep. DeepRM uses a policy gradient network to train the agent. The network is composed of a shallow fully connected NN with one hidden layer of 20 neurons. In our implementation, we exploit the use of CNN instead, as in [2].

DeepRM demonstrates that RL achieves comparable and often better efficiency than other scheduling heuristics, however it has several limitations, that we will address in this paper. In particular, CuSH i) introduces the notion of resource locality, and this affects the actual computation time of the jobs; ii) distinguishes between jobs of various workload types, increasing the algorithm effectiveness and iii) introduces the notion of waiting time in the state representation.

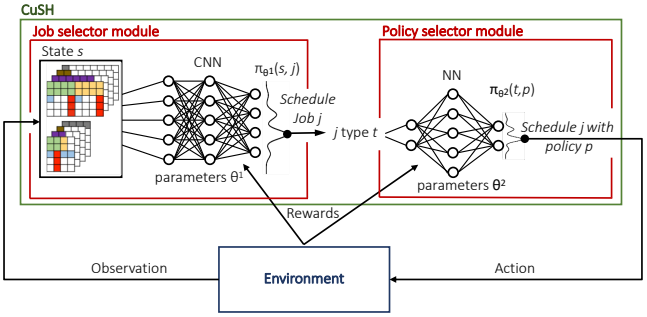

Figure 1: CuSH hierarchical agents trained via RL process.

3 PROPOSED SOLUTION

We model the RL environment as a cluster with N separate nodes, each having R resource types, a fixed number of available resource per node S^r (all nodes have the same size), and a queue of Q jobs waiting to be scheduled. Users submit jobs to the waiting queue. Then, the scheduler selects one or more of them to run at a given timestep. As it happens in traditional HPC job schedulers, such as IBM© Spectrum LSF[2], Slurm[3], or FLUX[4], job submission includes information regarding the required hardware resources (e.g., CPUs, GPUs, Memory, etc.) and the expected execution time (an upper bound of the actual completion time). The environment simulates an HPC cluster. The simulator returns the execution time of the jobs and takes into account how resource locality affects it. For instance, a job using 4 GPUs on the same node has higher locality than one using a single GPU in 4 different nodes. We collect from an actual HPC system the execution time of different workloads using different policies (Section 3.1). In summary, each job j is characterized by a profile containing a resources requirements vector $r_j = (r_{j,1}, ..., r_{j,R})$, the duration time t_j and the workload type w_j.

The CuSH agent is responsible for two tasks: select the next job in the queue and allocate the resources. In order to keep a reasonable small state space, we choose to separate these two tasks in two different modules that act in cascade. The first module is the JSM, its input is a representation of the current state of the cluster and the waiting jobs. Then, it either select the next job to schedule in the queue or does not select job (*do nothing*) as an output. Once the job selector picks a job, the PSM decides how to assign the resources to the job, based on its type. The two modules outputs are merged to make the scheduling action: schedule the job j with allocation policy p. The scheduling action will lead to a specific reward function for each module. Figure 1 depicts the workflow inside the CuSH agent. In the next sections, we will describe the details of each module.

[2]https://www.ibm.com/support/knowledgecenter/en/SSWRJV/product_welcome_spectrum_lsf.html
[3]https://slurm.schedmd.com/
[4]https://flux.ly/

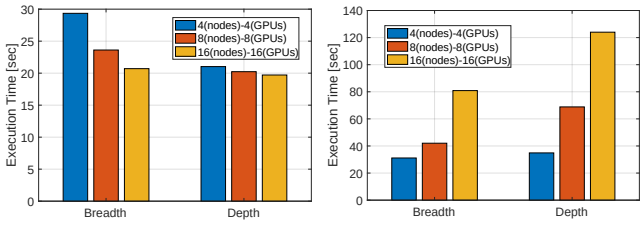

Figure 2: Execution time of compute intensive workload (dgemm, left) and network intensive workload (daxpy, right) with different resource allocation methods.

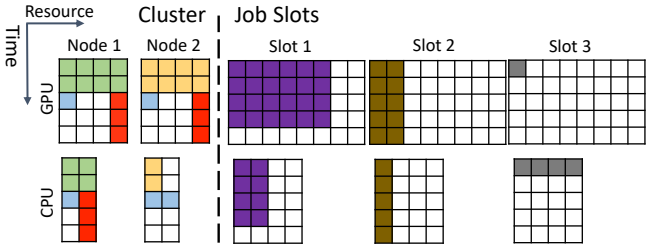

Figure 3: Example of environment (state) representation. This representation shows a cluster of 2 nodes (each one with 2 CPUs and 4 GPUs) and a queue (job slots) of 3 waiting jobs.

3.1 Environment simulator

We collected the execution time of different workloads to create a dataset for a realistic environment simulator. The initial dataset is based on the data gathered from a real HPC system[5]. Then it is extended generating additional samples with a linear extrapolation of the collected real data. Specifically, the environment simulator calculates the job service time based on the allocation policy (e.g., how we use resources) and the workload types (e.g., which workload we use and where is the bottleneck).

We considered two allocation policies: depth-first policy (assign requested resources utilizing as few nodes as possible) and a breadth-first policy (assign requested resources utilizing as many nodes as possible). Also, CuSH considers two different types of workloads: a compute intensive (dgemm, matrix multiplication) and a network intensive (daxpy, mixture of scalar multiplication and addition) workload.

Figure 2 shows the execution time for the dgemm and daxpy workloads with the two allocation policies. Each node used in the experiments contains 4 GPUs. In Figure 2 (a) the execution time of dgemm workload decreases (performance increases) when using the depth-first allocation policy and with a higher number of nodes. This is because using more resources reduces time to execute the same size of dgemm matrix and using localized GPUs (depth-first) always helps reduce time to copy matrices. Figure 2 (b) shows that the execution time of daxpy workload tends to decrease (performance increases) with the breadth-first policy, in this case a lower number of nodes leads to better performance. This is because daxpy is a network intensive workload and using a higher number of GPUs within a node (depth-first) adversely impact the performance creating a network bottleneck. Figure 2 (a) and (b) also show the performance impact when increasing the workload parallelism. The gathered data on the tested workloads are extended with a linear extrapolation. The RL environment uses this extended dataset to calculate the job service time during the simulations, adding to the time duration a penalty time depending on the job workload and how it is scheduled: $st_j = t_j + p_j$.

Knowing the workload type in advance may be difficult for some jobs. A simple solution is to add a new parameter to the job scheduler that would allow the user to specify the job type. Obviously, this solution is not optimal, and it would require pre-existing knowledge that is not always available to the user. A better approach is the use of dynamic scheduling. With dynamic scheduling, when the user does not explicitly specify the type of the job, the job is classified as "unknown". Then, after few executions, the job type can be automatically classified with a machine learning model [4].

[5]We gathered the workloads execution time data from a cluster of 26 nodes, each with 2 Power9® CPUs and 4 NVIDIA© Volta GPUs, all connected together with NVIDIA's high-speed NVLink. Nodes are connected using a dual-rail Mellanox EDR InfiniBand interconnect.

3.2 Job selector module (JSM)

JSM is the first part of the CuSH agent. JSM uses representation of current state as input, which includes scheduled jobs in the cluster and waiting jobs in the queue. Figure 3 shows a state representation of cluster nodes and of the waiting jobs as matrices, similarly to [8]. The left part of the figure shows the state of each node in the cluster, i.e. the resources of the node and how they are currently allocated for the running jobs. In this example the cluster contains two nodes with two types of resource: 2 CPUs and 4 GPUs each (the horizontal axis). In the figure, the state representation shows $T = 5$ timesteps (the vertical axis) and 4 jobs that are already scheduled (one per color). Different policies were used to allocate the resources for those 4 jobs: the yellow and green jobs have been allocated with a depth-first policy (assign requested resources utilizing as few nodes as possible), while the azure and red ones with a breadth-first policy (assign requested resources utilizing as many nodes as possible).

The right part of Figure 3 shows the waiting jobs in the queue (similarly to the cluster job representation). The total requested resources by a job cannot exceed the total cluster resources: 4 CPUs and 8 GPUs in this example. Each job in the queue is represented as the number of requested resources for a certain time duration t_j. For instance, the violet job in the first slot requires 2 CPUs and 6 GPUs for $4t$.

To reduce the size of the state representation, the maximum length of a job is reduced to T. Each job that is longer than T is represented by an equivalent job of length T. However, when it is scheduled it will run for its original duration. As mentioned, DeepRM has similar representation. However, DeepRM forces the job length to be contained in the state y-axis. This may cause an explosion of states when a longer (or fine-grained) job comes into the queue, leading to difficulties in training the scheduling agent. In addition, we keep the observable queue size Q fixed. New jobs are temporally stored in a backlog queue and moved to the observed queue once free slots are available in the observed queue. This shrewdness permits us to keep the state space of a reasonable size, even for a large environment.

To create an input for the NN, we merge the previously defined representations (cluster and queue). The cluster nodes are concatenated together along the x-axis, forming R matrices of $N \cdot S^r \times T$ size (in the example 4×5 for CPU and 8×5 for GPU), which is the same size of the representation of the waiting jobs. Cluster and queue are merged in the depth dimension (like channels of an image), creating a third dimension of size $Q + 1$, where the first channel is the cluster state and other channels are slots of the queue. Finally, to merge the different resources, we concatenate the resources along the x-axis[6]. In summary, the input of the job scheduler module is a three-dimensional matrix of size $(\sum_R N \cdot S^r) \times (T) \times (Q + 1)$, in the example of Figure 3 that would be $12 \times 5 \times 4$. This input representation allows the use of a CNN, where the third dimension corresponds to $(Q + 1)$ channels. DeepRM uses binary matrices to input whether a resource is in use/requested (1) or not (0), but we additionally code the current waiting time (instead of the 1s) for the waiting jobs in the queue. Considering that the RL agent treats the states with Markov property, without this information the agent cannot know if a job is waiting for a long time or it has just arrived in the queue.

The action space of the JSM is given by $\{\emptyset, 1, ..., Q\}$, where $a = i$ indicates scheduling the job at the i-th time slot; and $a = \emptyset$ indicates the *do nothing* action, meaning that the agent does not want to schedule any other job in the current timestep. While the output size of the NN is fixed to $Q + 1$, we discard the outputs related to unfeasible actions. The unfeasible actions are those related to empty queue slots or to jobs that do not fit in the cluster at any represented timestep (i.e. jobs with greater requests with respect to the resourced not yet allocated in the cluster). At each timestep, time is frozen until the JSM chooses the *do nothing* action (note that with an empty queue this is the only possible action). There is no limitation in the number of jobs that CuSH can schedule each timestep. After each scheduling action, the related queue slot is freed and the next job in the backlog queue (if any) is moved to that slot. Then the state is re-observed by the agent and a new action is performed. Once the agent picks the $a = \emptyset$ action, time goes to the next timestep: the cluster state image shifts up by one line and the waiting times of the waiting jobs in the queue increase. By decoupling the CuSH's actions from real time, the agent can schedule multiple jobs at each timestep while keeping fixed the action space.

The reward function uses a slightly different version of the common discounted future rewards [6]. In JSM, the return v_t represents the combination of discounted future local rewards r_t and the global reward r_g obtained at the end of the simulation, as in Eq. 1. This approach allows us to combine a local and immediate reward with a global one. The returns reflect how well the agent behaved in simulations.

$$v_t = (\sum_{i=t}^{L} \gamma^{i-t} r_t) r_g \tag{1}$$

The local reward r_t for each scheduling action is the sum of the ratio between waiting time and requested time duration of all the waiting job in the queue and in the backlog: $r_t = -\sum_{j=1}^{Q} \frac{wt_j}{l_j} - \sum_{j=1}^{B} \frac{wt_j}{l_j}$. The goal is to minimize the reward when scheduling the waiting jobs. The global reward r_g can be calculated using any evaluation metric of a system scheduler. We decided to use averaged normalized turnaround time (NTAT), as we believe it is the most suitable for the real scenarios (more details about the evaluation metrics are described in Section 4.1). In our experiments, we choose $\gamma = 0.9$ as the discount factor.

The JSM is a NN with a convolutional layer using 16 2x2 filters, stride=1 and without padding. The convolutional layer is followed by a ReLU activation and batch normalization. Finally, a softmax layer provides the predicted probability for each action.

3.3 Policy selector module (PSM)

The PSM has the goal of selecting which policy use to allocate a job that has to be scheduled. We designed the input as the workload type: daxpy or dgemm in the simulation. We fed this input to a shallow fully connected NN with a hidden layer of 10 neurons and a ReLU activation, followed by a softmax that returns as output the probability distribution for the two possible allocation policies (i.e. depth- or breadth-first).

The module is trained with policy gradient. The return v_t is only based on the local action a_t and its reward value r_t. The return is the locality penalty ($v_t = r_t = p_j$), that is calculated using the projected workloads data as in Section 3.1.

3.4 Training Process

CuSH is constituted by two NN modules. We train them in an episodic setting as described in Alg. 1. We denote θ^1 as the network parameters of the JSM, and θ^2 as those of the policy selector. We denote π_{θ^1} and π_{θ^2} as the JSM and PSM networks, respectively. Each episode consists of a fixed number of jobs (i.e., a jobset *js*) that are scheduled by CuSH. One episode terminates when all the jobs finish executing. At each discrete timestep, newly submitted jobs arrive and wait to be scheduled in a fixed-length queue. Exceeding jobs are saved in a backlog First Input First Out (FIFO) queue. When a slot in the queue turns available, a job from the backlog is moved to the queue using the first come first serve policy. We assume no preemption and a fixed resource allocation. All the requested resources must be allocated continuously from the job starting execution time until completion and are fixed once assigned to the job.

In each training epoch, we run M Monte Carlo simulations for each jobset, to explore the probabilistic space of different possible actions using the current models, updating the NNs once per all the jobsets. During the simulation of an episode, we keep track of the state, action, and reward information of both agents for each step (line 9 of Alg. 1). I.e. for each timestep i of the simulation we collect the state, action and local reward $\{s_i^1, a_i^1, r_i^1\}$ for the JSM, as well as those of the PSM $\{s_i^2, a_i^2, r_i^2\}$. The simulation of an episode also returns the global reward r_g. The local and global rewards are used to compute the discounted cumulative return of the JSM v_t^1 (line 10) and also for the PSM v_t^2 (line 11) for each timestep t. We then update the gradient information for the two networks (line 15 and 16) for each step taken in the simulation. Note that in line 15, we make use of a baseline value b_t to minimize the variance of the JSM returns [13]. We compute the baseline as the average of

[6]We also tried to keep the resources separated, with a NN for each, but we noticed worse results.

Algorithm 1 Training process

1: **for** each epoch e **do** in parallel:
2: **if** $e \mod K = 0$ **then**
3: $\theta^1 \leftarrow$allreduce(θ^1, AVG) *//synchronize JSM*
4: $\theta^2 \leftarrow$allreduce(θ^2, AVG) *//synchronize PSM*
5: $\Delta\theta^1 \leftarrow 0$ *//gradient reset*
6: $\Delta\theta^2 \leftarrow 0$ *//gradient reset*
7: **for** each jobset js **do**
8: **for** each Monte Carlo simulation s **do**
9: $\{s_i^1, a_i^1, r_i^1, s_i^2, a_i^2, r_i^2\}, r_g \leftarrow$ dosimulation(js) *//run simulation, each i is one of the L steps of the simulation.*
10: $v_t^1 \leftarrow (\sum_{i=t}^{L} \gamma^{i-t} r_i^1) r_g$ *//calculate JSM returns*
11: $v_t^2 \leftarrow r_t^2$ *//calculate PSM returns*
12: **for** each step in simulation i **do**
13: $b_t \leftarrow \frac{1}{L} \sum_{i=1}^{L} v_t^i$ *//baseline*
14: $b_t \leftarrow$allreduce(b_t, AVG) *//synchronize baseline*
15: $\Delta\theta^1 \leftarrow \Delta\theta^1 + \alpha^1 \sum_t \nabla_{\theta^1} \log \pi_{\theta^1}(s_t^1, a_t^1)(v_t^1 - b_t^1)$
16: $\Delta\theta^2 \leftarrow \Delta\theta^2 + \alpha^2 \sum_t \nabla_{\theta^2} \log \pi_{\theta^2}(s_t^2, a_t^2) v_t^2$
17: $\theta^1 \leftarrow \Delta\theta^1$ *//update JSM parameters*
18: $\theta^2 \leftarrow \Delta\theta^2$ *//update PSM parameters*

the returns obtained at the same time t in the different simulations (line 13).

Training a reinforcement agent is a compute-intensive task. Several approaches have been proposed to parallelize the training process [1, 10]. We adopted a P-learner K-step model averaging algorithm (KAVG) to explicitly manage the gradient staleness in parallel implementations [3, 19]. With KAVG, P workers run concurrently and average their parameters every K epochs (line 3 and 4 of Alg. 1). Note that the average at the very first epoch is an initial synchronization of the worker models. Other than the model parameters, another synchronization is needed in the distributed process: the computation of the baseline b_t among different simulations (line 14). Although this task introduces a lot of communication overhead, since the workers need to compute the baseline for each timestep of each jobset simulation, we noticed that it is crucial to keep control the variance of the returns and to avoid gradient divergence. All the synchronizations can be easily implemented with all reduce calls between the workers.

4 CUSH PERFORMANCE EVALUATION

The environment simulator simulates the behavior of an HPC cluster with $N = 128$ nodes. It considers $R = 2$ resource types (CPUs and GPUs) and it models two possible real-case configuration scenarios: i) $S^{CPU} = 2$ and $S^{GPU} = 4$ per node (similarly to the Sierra supercomputer) which is the default setting for our experiments; ii) $S^{CPU} = 2$ and $S^{GPU} = 6$ per node (similarly to the Summit supercomputer). As described in Section 3.1, we considered two workload types: compute intensive (dgemm) and network intensive (daxpy). The JSM sees a temporal horizon of $T = 5$. The size of the waiting jobs queue visible to the JSM is $Q = 5$.

In the default configuration, each training epoch is composed of 10 job sets. Each jobset contains 20 jobs of a maximum time length of $20t$. To simulate a real scenario, 75% of the jobs are randomly sampled to have a time duration less than or equal to $10t$ (short and medium short), while the other 25% have a length greater than

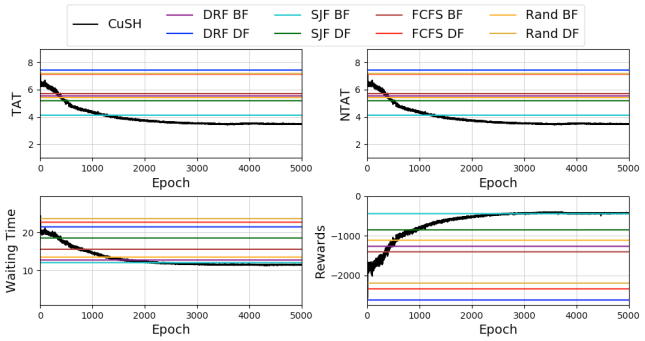

Figure 4: Learning curve in the environment of 128 nodes with 2 CPUs and 4 GPUs each (Sierra configuration).

$10t$. The job arrival time is randomly placed in the first $10t$ of an episode, then the episode runs until the completion of all the jobs. In each training epoch, we run $M = 24$ Monte Carlo simulations per jobset, divided among the distributed workers. We update both agent modules using rmsprop with a leaning rate $\alpha^1 = 1e^{-4}$ for the job selector agent and a learning rate $\alpha^2 = 1e^{-3}$ for the policy selector one. We run our training for 5000 epochs. Models are trained using a KAVG distributed training [19] on 12 GPUs with $K = 5$.

4.1 Performance Metrics and Scheduling Policies

To evaluate CusH we compare its performance against four traditional heuristic-based policies.

- First Come First Serve (FCFS) policy, which treats the waiting jobs as a FIFO queue, scheduling the first arrived one. In case of two jobs arrived at the same time, the shortest one is selected.
- Shortest Job First (SJF) policy, which schedules the shortest job in the queue. In case of multiple jobs with the same length, the first arrived one in the queue is scheduled.
- Dominant Resource Fairness [5] (DRF) policy, which selects the job with the smaller request of the dominant resource. The dominant resource of a job is the one with the higher ratio between request and environment availability.
- Random (Rand) policy, which schedules a random job in the queue.

All the policies are implemented with both a depth-first and a breadth-first version.

We use the averaged normalized turnaround time as the key performance metric [14]. The service time of a job is equivalent to the requested time plus the locality penalty given by the linear regression model, as described in Section 3.1. The turnaround time (TAT) of a job is given by its Waiting Time, i.e. the time spent in the queue, plus its service time. Finally, the normalized turnaround time (NTAT) is the ratio between the TAT and the job service time. Normalizing the TAT by the job's service time prevents biases towards large jobs.

Table 1: Results with up to 128 nodes using 2 CPUs and 4 GPUs (Sierra) and 2 CPUs and 6 GPUs (Summit) configurations.

	Sierra			Summit		
	NTAT	Wait time	Rewards	NTAT	Wait time	Rewards
CuSH	3.33	11.09	-379.92	3.8	11.79	-489.03
FCFS DF	7.13	22.68	-2342.47	6.54	20.90	-2014.06
FCFS BF	5.71	15.58	-1407.31	7.07	19.45	-2292.52
SJF DF	5.19	18.53	-853.70	5.23	18.37	-949.60
SJF BF	4.13	12.07	-444.83	5.14	15.33	-766.75
DRF DF	7.43	21.45	-2623.49	6.40	19.59	-1854.43
DRF BF	5.54	12.76	-1267.79	6.69	17.86	-2059.94
Rand DF	7.17	23.62	-2197.91	7.01	22.39	-2332.09
Rand BF	5.42	13.53	-1114.54	6.46	16.38	-1820.30

4.2 Scheduling Effectiveness

Figure 4 shows the averaged TAT, NTAT, Waiting Time and Rewards for CuSH training over the 5000 epochs in a Sierra environment (128 nodes with 2CPUs and 4GPUs each). The figures show the evolution of CuSH effectiveness (black lines) compared to the four policies in both depth-first (DF) and breadth-first (BF) allocation policies. It is clear that all the evaluation metrics are related to each other in the training phase. The training process ends with a best NTAT value of 3.33, that corresponds to an improvement of 19% with respect to the 4.13 of the best heuristic (SJF BF), and an improvement of 42% with respect to the FCFS policy, that is the one used for a default scheduling policy for IBM© Spectrum LSF. This results are also shown in Table 1.

Table 1 shows also the results with 6 GPUs per node. Also in this configuration the SJF is the best heuristic-base policy. CuSH achieves better results in all the metrics, with a NTAT improvement of 26% with respect to SJF and of 42% with respect to FCFS.

To understand why CuSH outperforms the other policies, we examined how the job selector behaves with jobs of different length. One can notice that the global reward r_g of the JSM is the NTAT: according to this metric it's more important to schedule short jobs as soon as possible. Figure 5 shows the NTAT averaged across all the jobsets and the waiting time for the jobs. Jobs are classified as: i) Short: 1-5t, ii) Medium short: 6-10t, iii) Medium long: 11-15t and iv) Long: 16-20t. It is interesting to observe how SJF treats the waiting time for the different jobs classes. The higher waiting time for long jobs with respect the other policies turns into a small loss in terms of NTAT, while the smaller waiting time for short jobs (that are the majority in our settings as well as in real scenarios) leads to a huge improvement in their NTAT with respect the other policies. Notably, in Figure 5, CuSH behavior is similar to SJF, but with less pronounced waiting time for the different classes. Interestingly, the waiting time for short jobs is slightly better for the SJF, but the NTAT is always better for CuSH scheduling. The reason of this gain is that the TAT is calculated with the waiting time plus the whole job service time (i.e. job duration+locality penalty), this remarks the importance of the PSM we introduced in CuSH.

4.2.1 Impact of PSM. As mentioned in Section 3.1, the scheduling policy can affect the job service time depending to the workload type. CuSH's PSM is responsible for selecting the best allocation policy for the specific workload type. Figure 6 shows how allocation policies are chosen by CuSH PSM per each workload type. The figure plots the ratio of each combination between policies and job types during training. In the beginning, the selection is quite random, but after the 1000th epoch the selection is almost perfect:

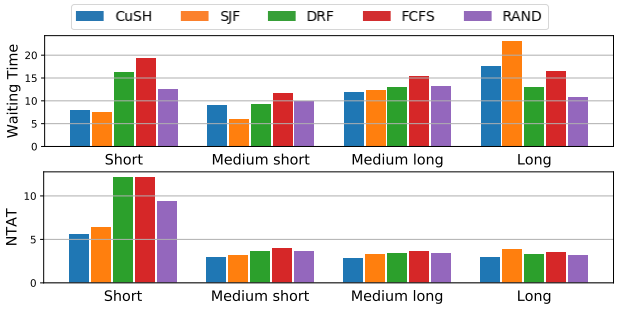

Figure 5: Averaged Waiting time and NTAT values classified by jobs length sets. Values for depth and breadth-first allocation policies are averaged.

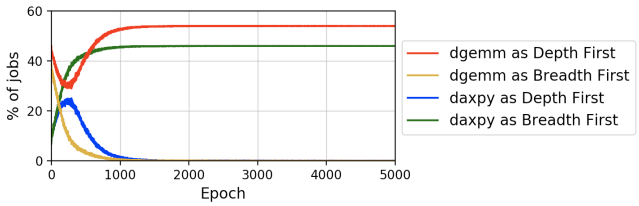

Figure 6: Ratio (averaged by all the job) of the allocation policies selected per each job type.

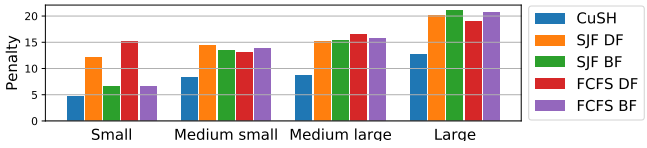

Figure 7: Averaged locality penalty of the different scheduling policies classified by the job requested resources. For sake of comprehension we show only SJF and FCFS, the omitted ones have similar behavior.

all dgemm jobs are scheduled with depth-first policy and the daxpy ones with breadth-first. The results show that CuSH PSM is able to allocate resources in a smart way. A better allocation leads to a shorter penalty time, and thus a better turnaround time.

Figure 7 shows the average locality penalty for jobs groped by the size. In the figure, jobs are classified as: Small: $r_m \leq 25\%$, Medium small: $25\% > r_m \leq 50\% r_m$, Medium large: $50\% > r_m \leq 75\%$ and Large: $r_m > 75\%$, where r_m is the maximum ratio between requested resources and size of the cluster (for instance, with a job request of 2 CPUs and 12 GPUs, and a cluster total capacity of 8 CPUs and 16 GPUs, $r_m = 75\%$ due to the requested GPUs). As expected, larger jobs receive higher locality penalty. The effectiveness of the PSM in providing the best allocation policy allows CuSH to always achieve a lower locality penalty with respect to all the other policies. This improvement is even more clear as the job sizes increase.

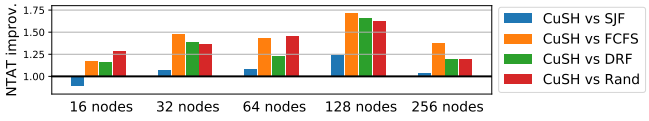

Figure 8: Improvement of CuSH NTAT compared to the heuristic-based policies, varying the number of nodes in the simulated cluster.

4.2.2 Impact of the environment Size. The reduced time horizon in the cluster and jobs representation (i.e., the y-axis of the images) reduces the JSM state space. We evaluated the default Sierra configuration with different time horizons, using $T = \{2, 5, 10, 20\}$. The CuSH NTAT results are very similar: 3.66, 3.33, 3.34, and 3.37, respectively.

We also evaluated the effect of the cluster size. Figure 8 shows the NTAT ratio between CuSH and the heuristic-based policies, with the Sierra configuration and the different number of nodes in the simulated cluster (for each configuration, here we show the best allocation policy per baseline). In a small environment of 16 nodes, where the locality of the resource allocation becomes less important, SJF is better than CuSH (ratio < 1). Increasing the size of the cluster, CuSH improves its performances, behaving always better than all the other policies.

5 CONCLUSIONS

This paper describes CuSH a resource management solution for heterogeneous high performance computing systems. CuSH observes the results of scheduling decisions and learns the optimal job scheduling strategy through the use of deep neural networks and reinforcement learning. It employs a two-step hierarchical approach to reduce the state-action space complexity, separating job selection from policy selection.

We evaluated CuSH against various heuristic-based policies including First Come First Serve, Shortest Job First, Dominant Resource Fairness, and Random. The evaluation is performed using a simulator based on real data. Results show that CuSH outperforms the best traditional heuristic-based approaches in all use cases, delivering up to 19% lower normalized turnaround time.

ACKNOWLEDGMENTS

We would like to acknowledge Nelson Mimura Gonzalez and Tonia Elengikal for the help in obtaining experimental results.

REFERENCES

[1] Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess, and Timothy Lillicrap. 2018. Distributional Policy Gradients. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SyZipzbCb

[2] Weijia Chen, Yuedong Xu, and Xiaofeng Wu. 2017. Deep Reinforcement Learning for Multi-Resource Multi-Machine Job Scheduling. *CoRR* abs/1711.07440 (2017). arXiv:1711.07440 http://arxiv.org/abs/1711.07440

[3] Guojing Cong, Giacomo Domeniconi, Joshua Shapiro, Fan Zhou, and Barry Chen. 2018. Accelerating deep neural network training for action recognition on a cluster of GPUs. In *First High Performance Machine Learning Workshop (HPML)*.

[4] Joseph Emeras, Sébastien Varrette, Mateusz Guzek, and Pascal Bouvry. 2015. Evalix: Classification and Prediction of Job Resource Consumption on HPC Platforms. In *JSSPP*.

[5] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.. In *Nsdi*, Vol. 11. 24–24.

[6] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[7] E. K. Lee, H. Viswanathan, and D. Pompili. 2017. Proactive Thermal-Aware Resource Management in Virtualized HPC Cloud Datacenters. *IEEE Transactions on Cloud Computing* 5, 2 (April 2017), 234–248.

[8] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 50–56.

[9] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 2430–2439. http://proceedings.mlr.press/v70/mirhoseini17a.html

[10] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. *CoRR* abs/1602.01783 (2016). arXiv:1602.01783 http://arxiv.org/abs/1602.01783

[11] Kalim Qureshi, Syed Munir Shah, and Paul Manuel. 2011. Empirical Performance Evaluation of Schedulers for Cluster of Workstations. *Cluster Computing* 14, 2 (jun 2011), 101–113. https://doi.org/10.1007/s10586-010-0128-5

[12] A. Reuther, C. Byun, W. Arcand, D. Bestor, B. Bergeron, M. Hubbell, M. Jones, P. Michaleas, A. Prout, A. Rosa, and J. Kepner. 2016. Scheduler technologies in support of high performance data analysis. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2016.7761604

[13] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *International Conference on Machine Learning*. 1889–1897.

[14] William. s Stallings. 2012. *Operating Systems Internals and Design Principle*. New Jersey: Prentice Hall.

[15] Richard S Sutton, Andrew G Barto, et al. 1998. *Reinforcement learning: An introduction*. MIT press.

[16] Xiaoyong Tang and Weizhen Tan. 2016. Energy-Efficient Reliability-Aware Scheduling Algorithm on Heterogeneous Systems. *Sci. Program.* 2016 (March 2016), 14–. https://doi.org/10.1155/2016/9823213

[17] Houssam-Eddine Zahaf, Abou El Hassen Benyamina, Richard Olejnik, and Giuseppe Lipari. 2017. Energy-efficient Scheduling for Moldable Real-time Tasks on Heterogeneous Computing Platforms. *J. Syst. Archit.* 74, C (March 2017), 46–60. https://doi.org/10.1016/j.sysarc.2017.01.002

[18] Wei Zhang and Thomas G. Dietterich. 1995. A Reinforcement Learning Approach to Job-shop Scheduling. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2 (IJCAI'95)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1114–1120. http://dl.acm.org/citation.cfm?id=1643031.1643044

[19] Fan Zhou and Guojing Cong. 2017. On the convergence properties of a K-step averaging stochastic gradient descent algorithm for nonconvex optimization. *arXiv preprint arXiv:1708.01012* (2017).