A HPC Co-Scheduler with Reinforcement Learning

Abel Souza[1], Kristiaan Pelckmans[2], and Johan Tordsson[1]

[1] Department of Computing Science, Umeå University, Sweden
{abel.souza,tordsson}@cs.umu.se,
[2] Department of Information Technology, Uppsala University, Sweden
kristiaan.pelckmans@it.uu.se

Abstract. Although High Performance Computing (HPC) users understand basic resource requirements such as the number of CPUs and memory limits, internal infrastructural utilization data is leveraged only by cluster operators, who use it to configure batch schedulers. This task is challenging and increasingly complex due to ever larger cluster scales and heterogeneity of modern scientific workflows. As a result, HPC systems achieve low utilization with long job completion times (makespans). To tackle these challenges, we propose a co-scheduling algorithm based on an adaptive reinforcement learning algorithm, where application profiling is combined with cluster monitoring. The resulting cluster scheduler matches resource utilization to application performance in a fine-grained manner (i.e., operating system level). As opposed to nominal allocations, we apply decision trees to model applications' actual resource usage, which are used to estimate how much resource capacity from one allocation can be co-allocated to additional applications. Our algorithm learns from incorrect co-scheduling decisions and adapts from changing environment conditions, and evaluates when such changes cause resource contention that impacts quality of service metrics such as jobs slowdowns. We integrate our algorithm in an HPC resource manager that combines Slurm and Mesos for job scheduling and co-allocation, respectively. Our experimental evaluation performed in a dedicated cluster executing a mix of four real different scientific workflows demonstrates improvements on cluster utilization of up to 51% even in high load scenarios, with 55% average queue makespan reductions under low loads.

Keywords: Datacenters, co-scheduling, high performance computing, adaptive reinforcement learning

1 Introduction

High Performance Computing (HPC) datacenters process thousands of applications supporting scientific and business endeavours across all sectors of society. Modern applications are commonly characterized as data-intensive, demanding processing power and scheduling capabilities that are not well supported by

large HPC systems [33]. Data-intensive workflows can quickly change computational patterns, e.g., amount of input data at runtime. Unfortunately, traditional HPC schedulers like Slurm [18] do not offer Application Programming Interfaces (APIs) that allow users and operators to express these requirements. Current resource provisioning capabilities barely satisfy today's traditional non-malleable workloads, and as a result, most HPC centers report long queue waiting times [1], and low utilization [4]. These problems delay scientific outputs, besides triggering concerns related to HPC infrastructure usage, particularly in energy efficiency and in the gap between resource capacity and utilization [37].

In contrast to HPC infrastructures that use batch systems and prioritize the overall job throughput at the expense of latency, cloud datacenters favor response time. On the one hand, cloud resource managers such as Kubernetes/Borg [6] and Mesos [16] assume workloads that change resource usage over time and that can scale up, down, or be migrated at runtime as needed [25]. This model allows cloud operators to reduce datacenter fragmentation and enable low latency scheduling while improving cluster capacity utilization. On the other hand, HPC resource managers such as Slurm and Torque [36], assume workloads with fixed makespan and constant resource demands throughout applications' lifespan, which must be acquired before jobs can start execution [12, 31]. Consequently, to improve datacenter efficiency and to adapt to quick workload variations, resources should be instantly re-provisioned, pushing for new scheduling techniques.

Thus, this paper proposes an algorithm with strong theoretical guarantees for co-scheduling batch jobs and resource sharing. Considering the problem of a HPC datacenter where jobs need to be scheduled and cluster utilization needs to be optimized, we develop a Reinforcement Learning (RL) algorithm that models a co-scheduling policy that minimizes idle processing capacity and also respects jobs' Quality of Service (QoS) constraints, such as total runtimes and deadlines (Section 3.1). Our co-scheduling algorithm is implemented by combining the Slurm batch scheduler and the Mesos dynamic job management framework. The potential benefit of more assertive co-allocation schemes is substantial, considering that – in terms of actual resource utilization, as opposed to nominal allocations – the current HPC practice often leaves computational units idling for about 40% [30, 39].

We evaluate our solution by experiments in a real cluster configured with three different sizes and four real scientific workflows with different compute, memory, and I/O patterns. We compare our co-scheduling strategy with traditional space-sharing strategies that do not allow workload consolidation and with a static time-sharing strategy that equally multiplexes the access to resources. Our RL co-scheduler matches applications QoS guarantees by using a practical algorithm that improves cluster utilization by up to 51%, with 55% reductions in queue makespans and low performance degradation (Section 4).

2 Background and Challenges

The emergence of cloud computing and data-intensive processing created a new class of complex and dynamic applications that require low-latency scheduling and which are increasingly being deployed in HPC datacenters. Low latency scheduling requires a different environment than batch processing that is dominant in HPC environments. HPC systems are usually managed by centralized batch schedulers that require users to describe allocations by the total run time (makespan) and amounts of resources, such as the number of CPUs, memory, and accelerators, e.g., GPUs [13,18,31]. As depicted in Figure 1(a), in the traditional HPC resource reservation model, known as space sharing, each job arrives and waits to be scheduled in a queue until enough resources that match the job's needs are available for use. In this scenario, the batch system uses backfilling [23] to keep resources allocated and maximize throughput, and users benefit from having jobs buffered up (queued) and scheduled together, a technique known as gang scheduling [11].

The main scheduling objectives in traditional HPC is performance predictability for parallel jobs, achieved at the expense of potential high cluster fragmentation, scalability and support for low latency jobs. In addition, this model assumes that the capacity of allocated computing resources is fully used, which is rarely the case [39], especially at early stages of development. This static configuration can be enhanced through consolidation, where jobs that require complementary resources share the same (or parts of) physical servers, which ultimately increases cluster utilization. In HPC, consolidation happens mostly in non-dedicated resource components, such as the shared file systems and the network. Resource managers, such as Slurm, allow nodes to be shared among multiple jobs, but do not dynamically adjust the preemption time slices.

2.1 Resource Management with Reinforcement Learning

Reinforcement learning (RL) is a mathematical optimization problem with states, actions, and rewards, where the objective is to maximize rewards [27]. In HPC, this can be formulated as a set of cluster resources (i.e., the environment) to where jobs need to be assigned by following an objective function (i.e., the rewards). Three common objective functions in HPC are minimizing expected queue waiting times, guaranteeing fairness among users, and increasing the cluster utilization and density. The scheduler's role is to model the cluster – i.e., to map applications behaviour to provisioned resources – through actionable interactions with the runtime system, such as adjusting cgroup limits [26]. Finally, to evaluate actions, the scheduler can observe state transitions in the cluster and subsequently calculate their outcomes to obtain the environment's reward. One example of action used extensively in this work is to co-schedule two applications A and B onto the same server and measure the reward (or loss) in terms of runtime performance. The latter compared to a scenario with exclusive node access for A and B. In this RL framework, the scheduler is the learner and interactions allow it to model the environment, i.e., the quantitative learning of

the mapping of actions to outcomes observed by co-scheduling applications. A reward (or objective) function describes how an agent (i.e., a scheduler) should behave, and works as a normative evaluation of what it has to accomplish. There are no strong restrictions regarding the characteristics of the reward function, but if it quantifies observed behaviors well, the agent learns fast and accurately.

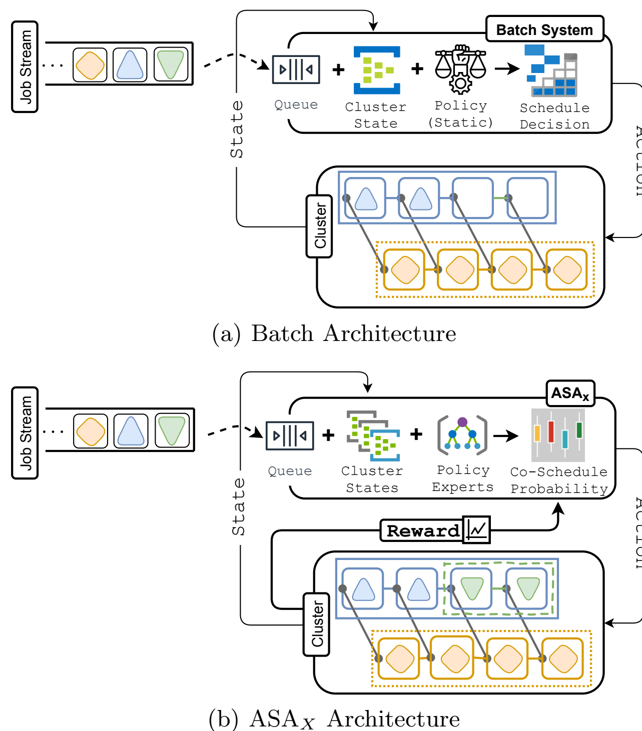
2.2 Challenges

Common HPC resource managers do not profile jobs nor consider job performance models during scheduling decisions. Initiatives to use more flexible policies are commonly motivated by highly dynamic future exascale applications, where runtime systems need to handle orchestration due to the large number of application tasks spawned at execution time [8,42]. Currently, dynamic scheduling in HPC is hindered as jobs come with several constraints due to their tight coupling, including the need for periodic message passing for synchronization and checkpointing for fault-tolerance [14]. Characteristics such as low capacity utilization [4] show a potential to improve cluster efficiency, where idle resources – ineffectively used otherwise – can be allocated to other applications with opposite profile characteristics.

Thus, different extensions to the main scheduling scheme are of direct relevance in practical scenarios. Server *capacity* can be viewed as a single-dimensional variable, though in practice it is multivariate and includes CPU, memory, network, I/O, etc. As such, a practical and natural extension to this problem is to formulate it as a multi-dimensional metric [22]. However, co-scheduling one or more jobs affects their performance as a whole [17], meaning that the actual capacity utilization of a job depends on how collocated jobs behave, making this capacity problem even more complex.

2.3 The Adaptive Scheduling Architecture

ASA – The Adaptive Scheduling Architecture is an architecture and RL algorithm that reduces the user-perceived waiting times by accurately estimating queue waiting times, as well as optimizes scientific workflows resource and cluster usage [35]. ASA encapsulates application processes into containers and enable fine-grained control of resources through and across job allocations. From user-space, ASA enables novel workflow scheduling strategies, with support for placement, resource isolation and control, fault tolerance, and elasticity. As a scheduling algorithm, ASA performs best in settings where similar jobs are queued repeatedly as is common setting in HPC. However, ASA's inability to handle states limits its use in broader scheduling settings. As this paper shows, embedding states such as application performance, users' fair-share priorities, and the cluster capacity directly into the model enables more accurate scheduling decisions.



(a) Batch Architecture

(b) ASA$_X$ Architecture

Fig. 1. (a) Batch and (b) ASA$_X$ Architectures: (a) A traditional batch system such as Slurm. The Upside down triangle (green) job waits for resources although the cluster is not fully utilized; (b) ASA$_X$, where rewards follow co-scheduling decision actions, steered through *Policy Experts*. In this example, the upside down triangle job (green) is co-allocated with the triangle job (blue).

3 A Co-Scheduler Architecture and Algorithm

In this section we introduce ASA$_X$: the HPC co-scheduler architecture and its algorithm. The goals of ASA$_X$ are higher cluster throughput and utilization, while also controlling performance degradation as measured by application completion time. Additionally, ASA$_X$ introduces the concept of cluster and application states (Section 3.1). When measured, these concepts influence how each co-scheduling action is taken. Handling states is the main difference to ASA and enables the creation of scheduling strategies that achieve an intelligent exploitation (finding the Pareto frontier) of the space spanned by the application QoS requirements and the available compute capacity. As such, evaluating ASA$_X$'s performance depends on how time-sharing strategies compare to space-sharing strategies (the default policy of most HPC clusters) and that guarantees predictable performance. We handle the problem of growing number of actions by incorporating expert decision tree structures that are efficiently computed. To-

gether with the loss functions, decision trees can be efficiently computed and evaluated to update the probabilities that affect how co-scheduling actions are chosen and how resources are used.

3.1 Architecture and Algorithm Overview

Fig. 1(b) shows ASA_X's architecture, where jobs are queued normally and share allocated resources with other jobs. The main differences from the regular batch scheduling (illustrated in Fig. 1(a)) is that a single deterministic cluster policy is replaced with a probabilistic policy that evolves as co-scheduling decision outcomes are evaluated. The job co-scheduling is implemented by an algorithm (see Section 3.2) that creates a collocation probability distribution by combining the cluster and application states, the cluster policy, and the accumulated rewards. Note that the cluster 'Policy' is analytically evaluated through decision trees (DTs) (Fig. 2), which correspond to the (human) *experts* role in RL. Experts map the applications and system online features, represented by internal cluster metrics such as CPU and memory utilization, into collocation decisions. Combined through multiple DTs, these metrics form the cluster *state*. Each leaf in the DT outputs a multi-variate distribution $\mathrm{p}_{i,j}$ that represents the likelihood of taking a specific action, and combined they affect how co-schedule decisions ought to be made. In here, actions are defined by how much CPU cgroup [26] quota ratios each job receives from the resources allocated to them, which influence how the operating system scheduler schedules application processes at runtime. Another difference with traditional HPC architectures regards the addition of a feedback mechanism, where *rewards* are accumulated after each co-scheduling action. This mechanism allows the architecture to asses the outcome of a co-scheduling decision to improve future decisions. To enable a job-aware scheduler, decisions are implemented in a per-job basis (see Section 3.2). Thus, ASA_X aims at discovering which expert is the best to minimize the performance degradation faced by a job due to co-schedule decisions.

There can be an arbitrary number of experts, and they can be described by anything that is reasonably lean to compute, from functions to decision trees. However, the combination of all experts needs to approximate the current cluster state as precisely as possible. To define and compute experts, in this paper we use metrics such as CPU (`CPU%`) and memory utilization (`Mem%`), workflow stage `type` (i.e. *sequential*, where only one core is utilized, or *parallel*, where all cores are utilized), time `interval` since the job started execution related to its duration (e.g. 25%, 50%, 75%). In addition to these metrics, we also define `Hp(t)`, a `happiness` metric that is defined at time t for a given job as

$$Hp(t) = \frac{|t_{Walltime} - t| * \#RemainingTasks}{\#Tasks/s}. \tag{1}$$

Devised from a similar concept [20], the happiness metric relates the job's remaining execution time with its remaining amount of tasks. It enables the analytical evaluation of a running job in regards to the allocated resource capacity and its overall throughput. Given the remaining time ($|t_{Walltime} - t|$) and per-

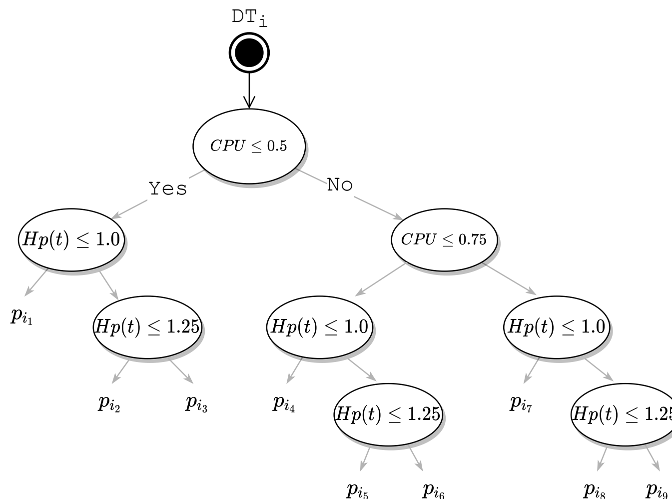


Fig. 2. A Decision tree (DT) expert structure illustrating the evaluation of the 'CPU' state in an allocation. For each DT 'Metric (`CPU%`, `Mem%`, `type`, and `interval`), an action strategy is devised by combining four different distributions p_i. Then, depending on the state for each 'Metric', one distribution among nine ($p_{i_1}, ..., p_{i_9}$) is returned.

formance ($\#Tasks/s$), if `Hp(t)` is near to, or greater than 1.0, then it can be inferred that the job is likely to complete execution within the wall-time limit; else (if `Hp(t)` < 1.0), the job is likely to not complete by the wall-time. It is important to note that the `Hp(t)` metric best describes jobs that encapsulate one application process. This is achieved by encapsulating each workflow stage into a single job, as is common in scientific workflows (see Section 4).

Cluster Policy We combine states through DTs for each metric with the `happiness` metric. Figure 2 illustrates one possible expert for the CPU state. It evaluates if the `CPU%` is high (e.g. > 0.75), and if so, and if `Hp(t)` is near to 1.0, the policy expects high performance degradation due to the job collocation. Conversely, if the `CPU%` is low, and the `Hp(t)` is greater than 1.1, the performance degradation is likely to be small, and the expert increases the probability for jobs to be collocated. Rather than describing these relations explicitly, we work with a mixture of different decision trees. Hence, $\mathbf{p}_{i,j}(\mathbf{x})$ represents how likely an action a_j is to be taken given the evaluated state x_t, according to the i-th expert in DT_i. For instance, if there are four CPU quota co-allocations e.g. (0%, 25%, 50%, 75%), each degrading `Hp(t)` differently, `DT1` may evaluate how state x_t is mapped into a distribution of performance degradation $\mathbf{p}_{i,j}(\mathbf{x})$ for a situation where `CPU% > 0.75`, and `Hp(t) ≈ 1.0`. This scenario could return $\mathbf{p}_1(\mathbf{x}) = (0.6, 0.3, 0.05, 0.05)$, meaning action a_0 ($= 0\%$), i.e. no co-allocation, is likely the best co-schedule decision. Another DT2 could evaluate another state, e.g. related to memory or to how a given special resource behaves, etc, returning

a different $\mathbf{p}_{i,j}(\mathbf{x})$, with a different co-allocation action, impacting the final $\mathbf{p}$ accordingly (line 6 in Algorithm 1). In general, we design our decision trees to suggest more collocation when $Hp(t) \geq 1.0$, and ASA_X looks for other ongoing allocations whenever a 0% co-allocation is the best decision for a certain running job.

3.2 Algorithm Details

Table 1. Algorithm variables and their descriptions.

Feature	Name	Description
$\{a\}$	Action set	Actions that users or operators can take while scheduling or allocating resources to applications.
DT_i	Decision Tree	Used to assess the weight a given application or resource metric value ought to influence actions.
R_i	Risk	Risk for using DT_i.
r_i	Accumulated Risk	Accumulation of all risks incurred by DT_i.
$\ell(a)$	Loss function	The performance degradation due to collocation.

In here we describe the internal details of how ASA_X is implemented, with variable and details presented in Table 1. A co-scheduling action depends on the associated state $\mathbf{x} \in \mathbb{R}^d$, where d depends on the number of selected cluster metrics as shown in Fig. 2. A state $\mathbf{x}$ is a vector that combines the assessment of all distributions output by the DTs. At any time t that a co-scheduling decision is to be made, one action a out of m is taken with the system in state $\mathbf{x}$. However, while in state $\mathbf{x}$, co-allocating resources from a running Job_a to a Job_b may degrade Job_a's performance, and this incurs in an associated loss $\ell(a)$. This overall idea is represented in Algorithm 1, where we have a nested loop. The outer loop (starting in line 1) initiates and updates the parameters for each scheduling decision that can be made. Once an ongoing job allocation candidate is presented, the inner loop (line 3) co-allocates resources to other jobs until the cumulative loss exceeds a threshold, in which case learning needs to be updated. In mathematical terms, let $\mathbf{p} : \mathbb{R}^d \rightarrow \mathbb{R}^m$ be a function of states $\mathbf{x} \in \mathbb{R}^d$ to co-allocation actions, and the goal of learning is to optimally approximate this mathematical relationship. The central quantity steering $\mathbf{p}$ is the *risk* $\mathbf{R}$, here defined for each DT_i as a vector in $\mathbb{R}^n$ where

$$\mathbf{R}_i = \ell(a)\mathbf{p}_i(a). \tag{2}$$

To quantify the performance degradation due to co-scheduling at iteration t, we maintain a vector $\mathbf{r}_t$ of the total accumulated risk (line 10). At each co-scheduling decision at time t, we let $\mathbf{r}_{t,i}$ denote the accumulated risk of the ith expert. ASA_X then creates a strategy to minimize $\mathbf{r}_i$:

$$\min_{*} \sum_{s=1}^{t} \sum_{j=1}^{n_s} \mathbf{p}_*(\mathbf{x}_{sj}) \ell\left(a_{sj}\right). \quad (3)$$

Note that the strategy minimising $\mathbf{r}_t$ corresponds to the maximum likelihood estimate, meaning we want to follow the expert that assigns the highest probability to the actions with the lowest loss. In here m actions $a = \{a_0, a_1, \ldots, a_{m-1}\}$ correspond to a discretization of resource allocations, and are expressed in terms of CPU% quota allocation ratios of m intervals, e.g. if $m = 10$ then $a = \{a_0 = 0\%, a_1 = 10\%, \ldots, a_9 = 90\%\}$. This means a fraction a_m of an ongoing Job_a allocation is re-allocated to a Job_b. When co-scheduling a_m from an allocation to other job, we calculate the performance degradation loss $\ell(a)$, limiting it to 1.0 and proportionally to the actual job submission execution time, normalized by its wall-time. For example, if an user requests a wall-time of 100s, and the submission finishes in $140s$, the loss would be proportional to $min(1.0, |140-100|/100)$. On the other hand, if the submission finishes in 70s, then the loss is 0, because it successfully completed within its wall-time limits. ASA$_X$ then optimizes $\mathbf{p}$ over the m quota allocations (actions) according to the loss $\ell(a)$ and the accumulated risks $\mathbf{r}_{ti}$ for Job_a. Accumulated risks help in cases where the co-scheduling strategy incurs in high performance degradation. Therefore, in Algorithm 1 we set a threshold $\mathbf{r}_t$ (line 3) bounding the accumulated risk for a given Job_a, and reset it in the situation when the co-location actions resulted in large performance degradation losses (line 12). This allows Algorithm 1 to update its knowledge about Job_a and bound the accumulated mistakes. Notably, we prove that the *excess risk* E_t after t co-scheduling decisions is bound, meaning the algorithm converges to an optimal point in a finite time after a few iterations (see Appendix A).

Finally, rigid jobs can only start execution when all requested resource are provisioned (see Section 2). To simultaneously achieve this and to improve cluster utilization, HPC infrastructures use *gang scheduling* [11]. By running on top of Mesos [16] (see Section 2.3), ASA$_X$ can scale to manage thousands of distributed resources. These features include fine-grained resource isolation and control (through cgroup quotas), fault tolerance, elasticity, and support for new scheduling strategies. However, although Mesos supports resource negotiation, it does not support gang scheduling, nor timely reservation scheduling. Therefore, ASA$_X$ also works as a framework to negotiate resource offers from Mesos in a way that satisfies queued jobs' requirements and constraints. ASA$_X$ withholds offers, allocating them when enough resources fulfill the job's requirements, and releasing the remaining ones for use by other allocations. This is a basic version of the *First fit* capacity based algorithm [11]. Our focus is towards improving cluster throughput and utilization, which may have direct impacts on reducing queue waiting times and jobs response times. Although in general many other aspects such as data movement and locality also influence parallel job scheduling, we do not (explicitly) address them.

Algorithm 1 ASA_X

Require:

Queued Job_b

Job_a allocation satisfying Job_b `#In terms of resource`

m co-allocation actions a, e.g. $m = 10$ and $a = \{a_0 = 0\%, ..., a_{m-1} = 90\%\}$

Initialise $\alpha_{0i} = \frac{1}{n}$ for $i = 1, \ldots, n$ `#metrics in state x, e.g. CPU, Mem.`

1: **for** $t = 1, 2, \ldots$ **do**
2: Initialise co-allocation risk $\mathbf{r}_{ti} = 0$ for each i-th metric
`#Co-allocation assessment:`
3: **while** $\max_i \mathbf{r}_{ti} \leq 1$ **do**
4: Evaluate Job_a's state $\mathbf{x}$:
5: Compute each ith-DT metric expert $\mathbf{p}_i(\mathbf{x}) \in \mathbb{R}^m$
6: Aggregate Job_a's co-schedule probability as $\mathbf{p} = \sum_{i=1}^{n} \alpha_{t-1,i} \mathbf{p}_i(\mathbf{x}) \in \mathbb{R}^m$
7: j = sample one action from a according to $\mathbf{p}$
8: Allocate a_j from Job_a's to co-schedule Job_b
9: Compute Job_a's performance loss $|\ell(a_j)| \leq 1$ due to co-allocation
10: For all i, update Job_a's risk $\mathbf{r}_{ti} = \mathbf{r}_{ti} + \mathbf{p}_i(a_j)\ell(a_j)$
11: **end while**
12: For Job_a and for all i, update $\alpha_{t,i}$ as

$$\alpha_{t,i} = \frac{\alpha_{t-1,i}}{N_t} \times e^{-\gamma_t \mathbf{r}_{ti}}$$

where N_t is a normalising factor such that $\sum_{i=1}^{n} \alpha_{t,i} = 1$.
13: **end for**=0

4 Evaluation

In this section we evaluate ASA_X with respect to workloads' total makespan, cluster resource usage, and workflows total runtime. We compare ASA_X against a default backfilling setting of Slurm and a static co-allocation setting.

4.1 Computing System

The experimental evaluation is performed on a system with high performance networks, namely a NumaScale system [34] with 6 nodes, each having two AMD Opteron Processor (6380) 24-cores and 185 GB memory. The NumaConnect hardware interconnects these nodes and appears to users as one large *single* server, with a total of 288 cores and 1.11 TB of memory. Memory coherence is guaranteed at the hardware level and totally transparent to users, applications, and the OS itself. Servers are interconnected through a switcher fabric 2D Torus network which supports sub microsecond latency accesses. The NumaScale storage uses a XFS file system, providing 512 GB of storage. The system runs a CentOS 7 (Kernel 4.18) and jobs are managed by Slurm (18.08) with its default backfilling setting enabled. When managing resources in the static and ASA_X settings, jobs are managed by Mesos 1.9 that uses our own framework/co-scheduler on top of it (see Fig. 1(b) and Section 3.1).

4.2 Applications

Four scientific workflows with different characteristics were selected for our evaluation Montage, BLAST, Statistics, and Synthetic.

Montage [5] is an I/O intensive application that constructs the mosaic of a sky survey. The workflow has nine stages, grouped into two parallel and two sequential stages. All runs of Montage construct an image survey from the `2mass` Atlas images.

BLAST [3] is a compute intensive applications comparing DNA strips against a large database ($>$ 6 GB). It maps an input file into many smaller files and then reduces the tasks to compare the input against the large sequence database. BLAST is composed of two main stages: one parallel followed by one sequential.

Statistics is an I/O and network intensive application which calculates various statistical metrics from a dataset with measurements of electric power consumption in a household with an one-minute sampling rate over a period of almost 4 years [19]. The statistics workflow is composed mainly of two stages composed of two sequential and two parallel sub-stages. The majority of its execution time is spent exchanging and processing messages among parallel tasks.

Synthetic is a two stages workflow composed of a sequential and a parallel task. This workflow is both data and compute intensive. It first loads the memory with over one billion floating point numbers (sequential stage), and then performs additions and multiplications on them (parallel stage).

4.3 Metrics

The following metrics are used in the evaluation. The *total runtime* is measured by summing up the execution times for each workflow stage, submitted as separate jobs. Equally important, *the response time* (also known as makespan, or flow time) is defined as the time elapsed between the submission and the time when the job finishes execution. A related metric is the waiting time, which is the time one job waits in the queue before starting execution. Additional metrics are CPU and memory utilization as measured by the Linux kernel. These latter two capture the overall resource utilization, and aids understanding of how well co-schedulers such as ASA_X model application performance.

4.4 Workloads

To evaluate ASA_X co-scheduling we compare it against a Static CPU quota configuration and a default (space-sharing) Slurm setting with backfilling enabled. We execute the same set of 15 workflow jobs (4 Montage, 4 BLAST and 4 Statistics [8, 16, 32, 64 cores], and 3 Synthetic [16, 32, 64 cores]) three times, each with three size configurations: namely 64 (x2), 128 (x4), and 256 (x8) cores. The workflows have different job geometry scaling requests ranging from 8 cores to up to 64 cores (smallest cluster size, x2), totalling 512 cores and 45 job sub-

missions for each cluster size. This workload selection demonstrates how the different scheduling strategies handle high (cluster size x8 = 256 cores), medium (x4 = 128 cores), and low loads (x2 = 64 cores), respectively. When comparing the Static setting and ASA_X against Slurm, neither of them get access to more resources (i.e. cores) than the cluster size for each experimental run. In all experiments, Slurm statically allocates resources for the whole job duration. Moreover, when scheduling jobs in the Static configuration, collocations of two jobs in a same server are allowed and the time-sharing CPU quota ratio is set to 50% for each job (through cgroups [26]). Collocation is also done for ASA_X, but the CPU quotas are dynamically set and updated once the rewards are collected according to Algorithm 1. Notably, to reduce scheduling complexity, our First Fit algorithm does not use Mesos resource offers coming from multiple job allocations. Finally, as mentioned in the previous section, the loss function $\ell(a)$ to optimize the co-schedule actions is calculated proportionally to the user requested wall-time and to the actual workflow runtime, similarly to the `Hp(t)` metric. The $\ell(a)$ values span 0.0 and 1.0, where 1.0 means performance degraded at least 50%, and 0.0 means no performance degradation.

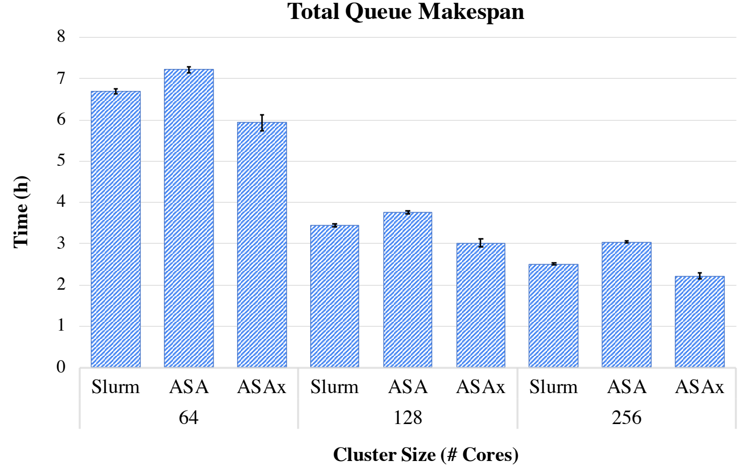
4.5 Results

Figures 3(a) and 3(b), and Tables 2 and 3 summarize all experimental evaluation, which is discussed below.

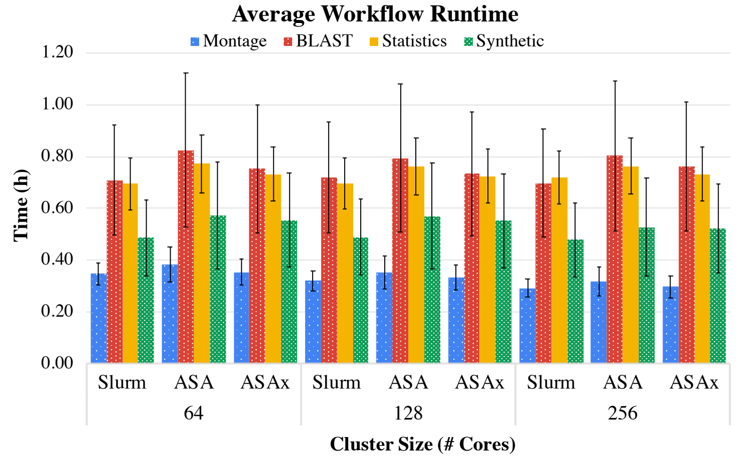
Makespan and Runtimes. Figures 3(a) and 3(b) show respectively queue workload makespans and workflow runtimes, both in hours. We note that the total makespan reduces as the cluster size is increased. As expected, Slurm experiments have small standard deviations as they use the space-sharing policy and isolate jobs with exclusive resource access throughout their lifespan. Static allocation yields longer makespans as this method does not consider actual resource usage by applications, but rather takes deterministic decisions about co-scheduling decisions, i.e. allocates half the CPU capacity to each collocated application. In contrast, ASA_X, reduces the overall workload makespan by up to 12% (64 cores) as it learns overtime which jobs do not compete for resources when co-allocated. For this reason, some initial ASA_X scheduling decisions are incorrect, which explains the higher standard deviations shown in Fig. 3(a).

Table 2 and Figure 3(b) show the aggregated average runtimes for each workflow, with their respective standard deviations. Notably, the high standard deviations illustrate the scalability of each workflow, i.e. the higher standard deviation, the more scalable the workflow is (given the same input, as it is the case in the experiments). Figure 3(b) shows different runs for the same application with different number of cores. If an application is scalable, the more cores the allocation gets, the faster the application completes execution, and the larger the standard deviation. Conversely, when the application is not scalable, performance is not improved when more cores are added to the allocation. The application cannot take advantage of extra resources, resulting in more idle resources, thus reducing the standard deviation between the runs. Finally, BLAST

and Synthetic are two very scalable, CPU intensive workloads, which do not depend on I/O and network as Montage and Statistics do. The Static strategy has the highest standard deviations overall because its workload experiences more performance degradation when compared to both Slurm and ASA_X, which this is due to its static capacity allocation.



(a) Total workload queue makespan per strategy



(b) Average total runtime per strategy

Fig. 3. Slurm, Static, and ASA_X strategy results - Total (a) Queue makespan and (b) runtime of each cluster size (64, 128, and 256 cores) and scheduling strategy.

Table 2 shows workflows runtime as well as CPU and memory resource utilization, all normalized against Slurm. It can be seen that ASA_X is close to Slurm regarding average total runtimes, with low overall overheads up to 10%, reaching as low as 3% increase for Montage. A predictable, but notable achievement for both the Static setting and ASA_X is the improved CPU utilization. As only a specific fraction of resources are actually allocated in both strategies, the utilization ratio increases considerably. For instance, when a job requests 1 CPU, the Static setting allocates 50% of one CPU to one job and the other 50% is co-allocated to other job. For non CPU intensive workloads (Montage and Statistics), this strategy results in higher utilization ratios because such workloads consume less than the upper bound capacity, which is noticeable when comparing to Slurm. For CPU intensive workloads such as BLAST, this strategy hurts performance, extending the workflow total runtime, most noticeably in the Static results. In contrast, as ASA_X learns overtime that co-scheduling jobs with either BLAST or Synthetic results in poor performance, its scheduling decisions become biased towards co-allocating Montage and Statistics workflows, but not BLAST. Also notable is the increased CPU utilization for the Statistics workflow, a non CPU intensive workload. The only workload capable of utilizing most of the memory available in the system is the Synthetic workflow, which is both memory and CPU intensive. This property makes ASA_X avoid co-placing any other workload with Synthetic jobs, as memory is one of the key metrics in our decision trees (see previous section).

Aggregated Queue and Cluster Metrics Although total runtime results are important for the individual users, so are aggregated cluster metrics such as queue waiting times for cluster operators. These metrics are summarized in Table 3, which shows average waiting times (h), cluster CPU utilization (%), and response times (h). The key point in Table 3 relates to both average response time and queue waiting time. Queue waiting times are reduced by as much as 50% in both Static and ASA_X compared to Slurm. ASA_X has 55% lower response time (256 cores) than Slurm or Static, which shows that it makes co-scheduling decisions that – for the proposed workload – result in fast executions. Similarly to the Static setting, ASA_X also increases cluster (CPU) utilization by up to 59%, but with the advantage of low performance degradation while also reducing queue waiting times considerably. This also happens for low cluster loads, as can be seen in the x2 workload case, with 51% average waiting time reductions. Conversely, Slurm decreases CPU utilization as the load decreases, and it is thus not able to improve response time.

5 Discussion

The evaluation demonstrates how ASA_X combines application profiling with an assertive learning algorithm to simultaneously improve resource usage and cluster throughput, with direct reductions on the workload makespan. By combining an intuitive but powerful abstraction (decision tree experts) and an application agnostic (happiness) performance, ASA_X efficiently co-schedules jobs

Table 2. Workflows summary for Slurm, Static, and ASA_X in three different cluster sizes. CPU Util. and Mem Util. represent resource utilization (%) proportionally to total allocated capacity. Normalized averages over Slurm are shown below results for each cluster size. Acronyms: WF (Workflow), Stats (Statistics), and Synth. (Synthetic).

		Slurm			**Static**			**ASA_X**		
Cluster Size	**WF**	**Runtime (h)**	**CPU Util. (%)**	**Mem Util. (%)**	**Runtime (h)**	**CPU Util. (%)**	**Mem Util. (%)**	**Runtime (h)**	**CPU Util. (%)**	**Mem Util. (%)**
64	Montage	0.35 ±4%	45 ±5	10 ±5	0.38 ±7%	94 ±5	10 ±5	0.35 ±5%	95 ±4	10 ±4
	BLAST	0.71 ±21%	96 ±4	44 ±5	0.82 +30%	99 ±1	44 ±5	0.75 ±25%	99 ±1	44 ±5
	Stats	0.69 ±10%	26 ±2	5 ±1	0.77 ±30%	51 ±3	6 ±1	0.73 ±11%	61 ±4	5 ±1
	Synth.	0.49 ±15%	99 ±1	95 ±3	0.57 ±21%	99 ±1	95 ±1	0.56 ±18%	99 ±1	95 ±1
Normalized Average		-	-	-	*+10%*	*+230%*	*0%*	*+3%*	*+227%*	*0%*
128	Montage	0.32 ±4%	33 ±5	7 ±3	0.35 ±6%	73 ±5	7 ±3	0.33 ±5%	75 ±5	7 ±3
	BLAST	0.72 ±22%	95 ±3	45 ±5	0.79 ±29%	99 ±1	44 ±5	0.73 ±24%	99 ±1	44 ±5
	Stats	0.70 ±10%	16 ±2	1 ±1	0.76 ±11%	36 ±6	2 ±1	0.72 ±11%	43 ±7	1 ±1
	Synth.	0.50 ±15%	99 ±1	93 ±4	0.57 ±21%	99 ±1	93 ±3	0.55 ±18%	99 ±1	93 ±4
Normalized Average		-	-	-	*+19%*	*+5%*	*0%*	*+9%*	*+5%*	*0%*
256	Montage	0.29 ±4%	21 ±5	5 ±3	0.32 ±6%	55 ±5	5 ±3	0.30 ±4%	51 ±5	5 ±3
	BLAST	0.70 ±21%	91 ±3	42 ±6	0.80 ±29%	99 ±1	43 ±6	0.76 ±25%	99 ±1	43 ±6
	Stats	0.72 ±10%	10 ±2	1 ±1	0.76 ±11%	26 ±2	1 ±1	0.73 ±11%	86 ±6	1 ±1
	Synth.	0.48 ±14%	99 ±1	92 ±2	0.53 ±19%	99 ±1	91 ±2	0.52 ±17%	99 ±1	93 ±1
Normalized Average		-	-	-	*+10%*	*+36%*	*0%*	*+5%*	*+50%*	*+1%*

and improves cluster throughput. The overall performance degradation experienced by ASA_X (10% average runtime slowdown) is negligible when compared to its benefits, in particular as HPC users are known to overestimate walltimes [9, 32, 40]. When compared to Slurm, a very common batch system used in HPC infrastructures, ASA_X reduces average job response times by up to 10%, enabling HPC users to achieve faster time to results. Notably, these improvements are achieved also for lightly loaded clusters, where the co-scheduling actions optimize cluster resources and allows them to be shared with other jobs.

It is important to note that current HPC scheduling strategies, specially those based on backfilling, aim to maximize resource allocation at a coarse granularity level. It is generally assumed this is good policy, as it guarantees high allocation ratios without interfering with users' workflows and thus achieves predictable job performance. However, such strategies do not take advantage of the characteristics of modern dynamic workflows, namely adaptability to changed resource allocation, faults, and even on input accuracy. Nor does backfilling strategies take advantage of modern operating system capabilities such as fine-grained processes scheduling and access control such as Linux [26], available in

Table 3. Slurm, Static, and ASA_X - Average results for three strategies in each cluster size.

	Cluster Size	Cluster Load	Waiting Time (h)	CPU Util. (%)	Response Time (h)
Slurm	64	8x	3.5±1%	53±5	4.4±1%
	128	4x	1.5±1%	45±5	2.4±1%
	256	2x	0.5±1%	32±6	1.4±1%
Static	64	8x	1.7±1%	90±5	4.8±1%
	128	4x	0.8±1%	92±3	2.8±1%
	256	2x	0.3±1%	89±5	2.0±1%
ASA_X	64	8x	1.8±5%	82±7	3.5±3%
	128	4x	1.0±8%	84±5	2.0±2%
	256	2x	0.3±7%	83±4	0.9±2%

userspace. Notably in HPC, even classical rigid jobs may suffer from traditional space-sharing policies, as the reservation based model does not take into consideration how resources are utilized in runtime. This is due to historical reasons, mostly related to CPU scarcity and the need for application performance consistency and SLO guarantees. In contrast, by leveraging the dynamic nature of modern workflows, and by using well fine-grained resource control mechanisms such as cgroups, ASA_X is able to improve the cluster utilization without hurting jobs performance. Although applicable across a wide range of HPC environments, such scheduling features are particularly suitable for rackscale systems like Numascale that offers close to infinite vertical scaling of applications. In such systems, scheduling mechanisms like ASA_X are key as the default Linux Completely Fair Scheduler (CFS) – beyond cache-locality – does not understand fine-grained job requirements, such as walltimes limits and data locality.

Given one cluster workload, the goals of ASA_X are higher cluster throughput and increased resource utilization. A constraint for these goals is the performance degradation caused by resource sharing and collocation, as measured by applications' completion time. In our experiments, the same queue workload is submitted to three resource capacity scenarios and cluster sizes that creates low, medium, and high demands. This captures the main characteristics that many HPC clusters face as the cluster size and workload directly affect how resources should be scheduled, in particular given the goal to maximize resource utilization without affecting application runtime. As shown in Table 2 (Response Time), and due to collocation and sharing of resources, ASA_X impacts all application's completion time. This is mitigated by decisions the algorithm takes at runtime, which uses the applications state to evaluate the impacts of each collocation. However, it must also be noted that no hard deadlines are set for jobs' wall-clock times. This allows comparison of ASA_X to worst case scenarios

and evaluation of the overhead caused by resource sharing due to collocation. Collocation has neglectable total runtime impacts (up to 5% in our experiments) and users are known to request more time than needed when submitting their jobs. This that some predictable noisy interference will happen at runtime, and ASA_X's goal is simply to model this expected behavior. By learning with mistakes, ASAx avoids collocating two applications that may negatively impact the scheduler's loss function as described by the decision trees, evaluated at each time that the scheduler has to make a collocation decision.

Finally, the proposed *happiness* metric (Eq. 1) can enable ASA_X to mitigate and control the possible performance degradation due to bad collocation decisions. Together with the decision trees, this characteristic is specially useful for queues that share nodes among two or more jobs, e.g. debugging queues used during scientific applications development stages. It is important to note though, that the happiness metric assumes all tasks in a single job are sufficiently homogeneous, i.e., need approximately the same amount of time to complete execution. This is however the common case in most stages of a scientific workflow, as in-stage heterogeneity leads to inefficient parallelization and poor performance. As such, monitoring $Hp(t)$ before and after the co-scheduling of a job can also be useful to understand if such a decision actually is likely to succeed. However, this is outside the scope of this paper but can be seen as an natural extension to our co-scheduling algorithm as such feature can enable ASA_X to foresee performance degradation in running applications. This can ultimately enable ASA_X to optimize its co-schedule actions already before collocation, which further protects performance of collocated jobs. Another possible extensions is to generalize the ASA_X allocation capacity to accommodate more than two jobs in the same node simultaneously as long as they all impose limited performance degradation on each other.

6 Related Work

The scale and processing capacities of modern multi-core and heterogeneous servers create new HPC resource management opportunities. With increasing concerns in datacenter density, idle capacity should be utilized as efficiently as possible [37]. However, the low effective utilization of computing resources in datacenters is explained through certain important requirements. For instance, to guarantee user Quality-of-Services (QoS), datacenter operators adopt a worst-case policy in resource allocations [41], while in HPC, users expect stable performance guarantees. To reach QoS guarantees, users often overestimate resource requests and total execution times (wall-clock), further degrading datacenter throughput. Runtime Service Level Objectives (SLOs) is achieved through more efficient job scheduling, but this is rarely deployed in modern HPC infrastructures due to fear of performance degradation when sharing resources. Most of HPC batch systems use resource reservation with backfilling strategies, and rely on users to provide application resource and walltime requirements [18,36]. This model is based on the assumption that jobs fully utilize the allocated capac-

ity constantly throughout their lifespan. This is needed for some purposes such as cache optimizations, debugging, and is thus important during application development and testing. However, more often than not, jobs utilize less than this upper capacity [39], which is rarely the case, specially at early stages of development. Some proposals [2, 10] aim to extend this traditional model, with impacts need to be studied in more depth. Public cloud datacenters, on the other hand, offer alternatives for running HPC workloads, such as Kubernetes [6], and Mesos [16]. Their resource management policies are centered around low latency scheduling, offering fairness and resource negotiation [15]. However, these systems lack main capabilities such as resource reservation, gang scheduling, and also assume that applications fully utilize allocated resources [11, 39].

Tackling the utilization problem from a different angle, stochastic schedulers have been proposed as solutions to overcome user resource overestimates [14]. ASA_X can be used as an extension to this class of schedulers, because it offers a new scheduling abstraction through its experts, which can model stochastic applications as well. Similarly to ASA_X, Deep RL schedulers have been proposed [10, 21, 24, 42], though they focus either on dynamic applications, or on rigid-jobs. As previously discussed, an increasingly diverse set of applications are flooding new HPC infrastructure realizations. [29] uses Bayesian optimization to capture performance models between low latency and batch jobs. It differs from our experts approach in that ASA_X uses decision trees to assess such relationship, and [29] targets only two types of applications. Machine Learning (ML) models have been proposed to take advantage of the large datasets already available in current datacenters [7]. For instance, [28] repeatedly runs micro-benchmarks in order to build a predictive model for the relationship between an application and resource contention. In contrast to that type of ML, RL algorithms such as the one used in ASA_X and in [42] do not require historical training data to optimize scheduling. As mentioned in previous sections, ASA_X is a stateful extension of [35], where the main difference is that ASA_X can incorporate previous decisions in a RL approach. In [38], the authors combine offline job classification with online RL to improve collocations. This approach can accelerate convergence, although it might have complex consequences when new unknown jobs do not fit into the initial classification. Similarly to ASA_X, the Cognitive ScHeduler (CuSH) [10] decouples job selection from the policy selection problem and also handles heterogeneity. Finally, as an example of scheduling policy optimization, [42] selects among several policies to adapt the scheduling decisions. This approach is similar to our concept of a forest of decision trees (experts), although their work do not consider co-scheduling to target queue waiting time minimization combined with improved cluster utilization.

7 Conclusion

Since mainframes, batch scheduling has been an area of research, and even more since time-sharing schedulers were first proposed. However, today's HPC schedulers face a diverse set of workloads and highly dynamic requirements, such

as low latency and streaming workflows generated by data-intensive applications. These workflows are characterized by supporting many different features, such as system faults, approximate output, and resource adaptability. However, current HPC jobs do not fully utilize the capacity provided by these high-end infrastructures, impacting datacenter operational costs, besides hurting user experience due to long waiting times. To mitigate these, in this paper we propose a HPC co-scheduler (ASA_X) that uses a novel and convergence proven reinforcement learning algorithm. By analytically describing a cluster's policy through efficient decision trees, our co-scheduler is able to optimize job collocation by profiling applications and understanding how much of ongoing allocations can be safely reallocated to other jobs. Through real cluster experiments, we show that ASA_X is able to improve cluster CPU utilization by as much as 51% while also reducing response time and queue waiting times, hence improving overall datacenter throughput, with little impacts on job runtime (up to 10%, but 5% slower on average). Together with the architecture, our algorithm forms the basis for efficient resource sharing and an application-aware co-scheduler for improved HPC scheduling with minimal performance degradation.

A Convergence of $\mathbf{ASA_X}$

In here we explain how we can statistically bound the accumulated loss and create a strategy to take the best available actions in a given environment. For doing so, we define the *excess risk*, which estimates how "risky" taking an action can be. It is defined here as

$$E_t = \sum_{s=1}^{t}\sum_{j=1}^{n_s}\left(\sum_{i=1}^{n}\alpha_{s-1,i}\mathbf{p}_i(\mathbf{x}_{sj})\ell\left(a_{sj}\right)\right) - \min_{*}\sum_{s=1}^{t}\sum_{j=1}^{n_s}\mathbf{p}_*(\mathbf{x}_{sj})\ell\left(a_{sj}\right), \tag{4}$$

where $\mathbf{x}_{sj}$ denotes the decision tree states of the j-th case in the s-th round, and where a_{sj} is the action taken at this case.

Theorem 1 shows that the excess risk for Algorithm 1 is bound as follows.

Theorem 1 *Let $\{\gamma_t > 0\}_t$ be a non-increasing sequence. The excess risk E_t after t rounds is then bound by*

$$E_t \leq \gamma_t^{-1}\left(\ln n + \frac{1}{2}\sum_{s=1}^{t}\gamma_s^2\right). \tag{5}$$

Proof. Let a_{tj} denote the action taken in round t at the jth case, and let n_t denote the number of cases in round t. Similarly, let $\mathbf{x}_{tj}$ denote the state for this case, and let $\mathbf{p}_i(\mathbf{x}_{tj})$ denote the distribution over the a actions as proposed by

the ith expert. Let $\ell_{tj}(a)$ denote the (not necessarily observed) loss of action a as achieved on the tjth case.

Define the variable Z_t as

$$Z_t = \sum_{i=1}^{n} \exp\left(- \sum_{s=1}^{t} \gamma_s \sum_{j=1}^{n_s} \mathbf{p}_i(\mathbf{x}_{sj}) \ell_{sj}\left(a_{sj}\right) \right). \tag{6}$$

Then

$$\sum_{s=1}^{t} \ln \frac{Z_s}{Z_{s-1}} = \ln Z_t - \ln Z_0. \tag{7}$$

Moreover

$$\begin{aligned} \ln Z_t = \ln \sum_{i=1}^{n} \exp\left(- \sum_{s=1}^{t} \gamma_s \sum_{j=1}^{n_s} \mathbf{p}_i(\mathbf{x}_{sj}) \ell_{sj}\left(a_{sj}\right) \right) \geq \\ - \sum_{s=1}^{t} \gamma_s \sum_{j=1}^{n_s} \mathbf{p}_*(\mathbf{x}_{sj}) \ell_{sj}\left(a_{sj}\right), \end{aligned} \tag{8}$$

for any expert $* \in \{1, \ldots, n\}$. Conversely, we have

$$\begin{aligned} \ln \frac{Z_t}{Z_{t-1}} &= \ln \frac{\sum_{i=1}^{n} \exp\left(- \sum_{s=1}^{t} \gamma_s \sum_{j=1}^{n_s} \mathbf{p}_i(\mathbf{x}_{sj}) \ell_{sj}\left(a_{sj}\right) \right)}{\sum_{i=1}^{n} \exp\left(- \sum_{s=1}^{t-1} \gamma_s \sum_{j=1}^{n_s} \mathbf{p}_i(\mathbf{x}_{sj}) \ell_{sj}\left(a_{sj}\right) \right)} \\ &= \ln \sum_{i=1}^{n} \alpha_{t-1,i} \exp\left(-\gamma_t \sum_{j=1}^{n_t} \mathbf{p}_i(\mathbf{x}_{tj}) \ell_{tj}\left(a_{tj}\right) \right). \end{aligned} \tag{9}$$

Application of Hoeffding's lemma gives then

$$\ln \frac{Z_s}{Z_{s-1}} \leq -\gamma_s \sum_{j=1}^{n_s} \left(\sum_{i=1}^{n} \alpha_{s-1,i} \mathbf{p}_i(\mathbf{x}_{sj}) \ell_{sj}\left(a_{sj}\right) \right) + \frac{4\gamma_s^2}{8}, \tag{10}$$

using the construction that $\max_i \sum_{j=1}^{n_t} \mathbf{p}_i(\mathbf{x}_{sj}) \ell_{sj}\left(a_{sj}\right) \leq 1$. Reshuffling terms gives then

$$\begin{aligned} \sum_{s=1}^{t} \gamma_s \sum_{j=1}^{n_s} \left(\sum_{i=1}^{n} \alpha_{s-1,i} \mathbf{p}_i(\mathbf{x}_{sj}) \ell_{sj}\left(a_{sj}\right) \right) - \\ \sum_{s=1}^{t} \gamma_s \sum_{j=1}^{n_s} \mathbf{p}_*(\mathbf{x}_{sj}) \ell_{sj}\left(a_{sj}\right) \leq \ln n + \frac{1}{2} \sum_{s=1}^{t} \gamma_s^2. \end{aligned} \tag{11}$$

Application of Abel's second inequality gives the result.

□

References

1. ARCHER User Survey (2019), https://www.archer.ac.uk/about-archer/reports/annual/2019/ARCHER_UserSurvey2019_Report.pdf
2. Ahn, D.H., Garlick, J., Grondona, M., Lipari, D., Springmeyer, B., Schulz, M.: Flux: a next-generation resource management framework for large hpc centers. In: 43rd International Conference on Parallel Processing Workshops. IEEE (2014)
3. Altschul, S.F., Madden, T.L., Schäffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J.: Gapped blast and psi-blast: a new generation of protein database search programs. Nucleic acids research (1997)
4. Ambati, P., Bashir, N., Irwin, D., Shenoy, P.: Waiting game: optimally provisioning fixed resources for cloud-enabled schedulers. In: International Conference for High Performance Computing, Networking, Storage and Analysis (2020)
5. Berriman, G., Good, J., Laity, A., Bergou, A., Jacob, J., Katz, D., Deelman, E., Kesselman, C., Singh, G., Su, M.H., et al.: Montage: A grid enabled image mosaic service for the national virtual observatory. In: Astronomical Data Analysis Software and Systems (ADASS) XIII (2004)
6. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Queue (2016)
7. Carastan-Santos, D., De Camargo, R.Y.: Obtaining dynamic scheduling policies with simulation and machine learning. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2017)
8. Castain, R.H., Hursey, J., Bouteiller, A., Solt, D.: Pmix: process management for exascale environments. Parallel Computing (2018)
9. Cirne, W., Berman, F.: A comprehensive model of the supercomputer workload. In: Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No. 01EX538). pp. 140–148. IEEE (2001)
10. Domeniconi, G., Lee, E.K., Venkataswamy, V., Dola, S.: Cush: Cognitive scheduler for heterogeneous high performance computing system. In: DRL4KDD 19: Workshop on Deep Reinforcement Learning for Knowledge Discover (2019)
11. Feitelson, D.G.: Packing schemes for gang scheduling. In: Workshop on Job Scheduling Strategies for Parallel Processing. Springer (1996)
12. Feitelson, D.G., Rudolph, L.: Toward convergence in job schedulers for parallel supercomputers. In: Workshop on Job Scheduling Strategies for Parallel Processing. Springer (1996)
13. Feitelson, D.G., Tsafrir, D., Krakov, D.: Experience with using the parallel workloads archive. Journal of Parallel and Distributed Computing (2014)
14. Gainaru, A., Aupy, G.P., Sun, H., Raghavan, P.: Speculative scheduling for stochastic hpc applications. In: Proceedings of the 48th International Conference on Parallel Processing. ICPP 2019 (2019)
15. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: Fair allocation of multiple resource types. USENIX Symposium on Networked Systems Design and Implementation (2011)
16. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R.H., Shenker, S., Stoica, I.: Mesos: A platform for fine-grained resource sharing in the data center. (2011)
17. Janus, P., Rzadca, K.: Slo-aware colocation of data center tasks based on instantaneous processor requirements. arXiv preprint arXiv:1709.01384 (2017)
18. Jette, M.A., Yoo, A.B., Grondona, M.: Slurm: Simple linux utility for resource management. In: In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP). Springer-Verlag (2003)

19. Kolter, J.Z., Johnson, M.J.: Redd: A public data set for energy disaggregation research. In: Workshop on Data Mining Applications in Sustainability (SIGKDD), San Diego, CA (2011)
20. Lakew, E.B., Klein, C., Hernandez-Rodriguez, F., Elmroth, E.: Performance-based service differentiation in clouds. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE (2015)
21. Li, Y., Sun, D., Lee, B.C.: Dynamic colocation policies with reinforcement learning. ACM Transactions on Architecture and Code Optimization (TACO) (2020)
22. Li, Y., Tang, X., Cai, W.: On dynamic bin packing for resource allocation in the cloud. In: Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures. ACM (2014)
23. Lifka, D.: The anl/ibm sp scheduling system. In: Job Scheduling Strategies for Parallel Processing. IEEE (1995)
24. Mao, H., Schwarzkopf, M., Venkatakrishnan, S.B., Meng, Z., Alizadeh, M.: Learning scheduling algorithms for data processing clusters. In: ACM Special Interest Group on Data Communication (2019)
25. Mell, P., Grance, T., et al.: The nist definition of cloud computing (2011)
26. Menage, P.B.: Adding generic process containers to the linux kernel. In: Proceedings of the Linux Symposium. Citeseer (2007)
27. Monahan, G.E.: State of the art - a survey of partially observable markov decision processes: theory, models, and algorithms. Management science (1982)
28. Moradi, H., Wang, W., Fernandez, A., Zhu, D.: upredict: A user-level profiler-based predictive framework in multi-tenant clouds. In: 2020 IEEE International Conference on Cloud Engineering (IC2E). IEEE (2020)
29. Patel, T., Tiwari, D.: Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In: 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE (2020)
30. Reiss, C., Tumanov, A., Ganger, G.R., Katz, R.H., Kozuch, M.A.: Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: Proceedings of the Third ACM Symposium on Cloud Computing (2012)
31. Reuther, A., Byun, C., Arcand, W., Bestor, D., Bergeron, B., Hubbell, M., Jones, M., Michaleas, P., Prout, A., Rosa, A., et al.: Scalable system scheduling for hpc and big data. Journal of Parallel and Distributed Computing (2018)
32. Rocchetti, N., Da Silva, M., Nesmachnow, S., Tchernykh, A.: Penalty scheduling policy applying user estimates and aging for supercomputing centers. In: Latin American High Performance Computing Conference. pp. 49–60. Springer (2016)
33. Rodrigo Álvarez, G.P., Östberg, P.O., Elmroth, E., Ramakrishnan, L.: A2l2: An application aware flexible hpc scheduling model for low-latency allocation. In: Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing. ACM (2015)
34. Rustad, E.: Numascale: Numaconnect (2013), `https://www.numascale.com/index.php/numascale-whitepapers/`
35. Souza, A., Pelckmans, K., Ghoshal, D., Ramakrishnan, L., Tordsson, J.: Asa – the adaptive scheduling architecture. In: The 29th International Symposium on High-Performance Parallel and Distributed Computing. ACM (2020)
36. Staples, G.: Torque resource manager. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. SC'06, ACM (2006)
37. Stevens, R., Taylor, V., Nichols, J., Maccabe, A.B., Yelick, K., Brown, D.: Ai for science. Tech. rep., Argonne National Lab.(ANL), Argonne, IL (United States) (2020)

38. Thamsen, L., Verbitskiy, I., Nedelkoski, S., Tran, V.T., Meyer, V., Xavier, M.G., Kao, O., De Rose, C.A.: Hugo: a cluster scheduler that efficiently learns to select complementary data-parallel jobs. In: European Conference on Parallel Processing. Springer (2019)
39. Tirmazi, M., Barker, A., Deng, N., Haque, M.E., Gene Qin, Z., Hand, S., Harchol-Balter, M., Wilkes, J.: Borg: the next generation. SIGOPS European Conference on Computer Systems (EuroSys'20) (2020)
40. Uchroński, M., Bożejko, W., Krajewski, Z., Tykierko, M., Wodecki, M.: User estimates inaccuracy study in hpc scheduler. In: International Conference on Dependability and Complex Systems. pp. 504–514. Springer (2018)
41. Yang, H., Breslow, A., Mars, J., Tang, L.: Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In: ACM SIGARCH Computer Architecture News. ACM (2013)
42. Zhang, D., Dai, D., He, Y., Bao, F.S.: Rlscheduler: Learn to schedule hpc batch jobs using deep reinforcement learning. arXiv preprint arXiv:1910.08925 (2019)