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,tordsson}@cs.umu.se

[2]Department of Information Technology
Uppsala University, Sweden
kristiaan.pelckmans@it.uu.se

Abstract: High Performance Computing (HPC) datacenters process thousands of diverse applications, supporting many scientific and business endeavours. Although users understand fundamental resource job requirements such as amounts of CPUs and memory, internal infrastructural utilization data and system dynamics are often visible only to cluster operators. In addition to that, due to increased complexity, heuristically tweaking a batch system is even today a very challenging task. When combined with applications profiling, infrastructure data enables improvements to job scheduling, and also better support for Quality-of-Service (QoS) metrics such as queue waiting times and total execution times. Targeting improvements in utilization and throughput, this paper evaluates and proposes a novel reinforcement learning co-scheduling algorithm that combines capacity utilization with application performance profiling. We first profile a running application by assessing its resource utilization and progress by means of a forest of decision trees, enabling our algorithm to understand the application's resource capacity usage. We then use this information to estimate how much capacity from the current allocation can be used for co-scheduling additional applications. Our algorithm learns from incorrect actions and evaluates when co-scheduling decisions results in QoS degradation, such as application slowness. Our co-scheduling architecture uses handful metrics to help minimizing performance degradation, enabling improvements on utilization of up to 25% even when the cluster is experiencing high demands, with 10% average queue makespan reductions when experiencing low loads.

Key words: Datacenters, scheduling, high performance computing, reinforcement learning

1 Introduction

Today's dynamic application characteristics demand processing power and capabilities not well supported by large High Performance Computing (HPC) infrastructures. Some of these demands are barely satisfied by HPC, with long queue waiting times reported, and societal concerns on energy efficiency [**stevens2020ai**]. Similarly to rigid jobs, scientific workflows are characterized by handling large amounts of data, highly dynamic and adaptable to resource changes, system faults, and also allow approximate solutions to their problems [Pra+15; Reu+18]. However, the common HPC policy is space sharing, which by itself does not suffice for today's demands, mostly because they will soon outpace datacenter supply budgets, specially for upcoming exascale applications in the pipeline [**stevens2020ai**]. These illustrates the need for new developments in the intelligence and heuristics coded in HPC resource management. As such, new adaptive solutions that integrate learning strategies into applications' workflow are needed for accommodating extreme workload demands expected in the future.

On another spectrum, cloud computing datacenters favors response time over throughput, deeply contrasting with HPC, which uses a queued batch system environment and prioritizes throughput or total computation accomplished over-time, ignoring latency entirely. On the one hand, cloud resource managers such as Borg [Bur+16] and Mesos [Hin+11] assume adaptive workloads with changes in resource usage overtime, and where managed workflows can scale up, down, or be migrated dynamically as needed. Their main target is to reduce fragmentation and enable low latency task scheduling (e.g. first-come-first-served), while improving datacenter capacity utilization. On the other hand, HPC resource managers such as Slurm [JYG] assume rigid workloads with constant resource demands, limited running times and number of resources throughout workflows lifespan. Such workloads must acquire resources before jobs can start execution, and main resource manager targets are predictable scheduling for parallel allocations (e.g. gang scheduling), at the cost of potential higher fragmentation and scalability issues.

However, the ever increasing scale, size, and processing capacities of modern multi-core and heterogeneous servers composing most of upcoming pre-exascale systems open new opportunities for resource management. With increase concerns in datacenter density, idle capacity should be utilized as much as possible, although low effective utilization of compute resources is a major drawback of many modern datacenters [Yan+13]. In order to guarantee Quality-of-Services (QoS) and other requirements on the Service Level Objectives (SLOs) offered users, datacenter operators implement a worst-case view in resource allocation [Yan+13], specially aggravated in HPC operations. HPC users expect certain capacity guarantees while developing and testing scientific workflows, though total execution times commonly are overestimated, hurting datacenter efficiency. Guaranteeing runtime SLOs can be achieved through more efficient job allocation.

In this paper, we propose an algorithm with strong theoretical guarantees for co-scheduling batch jobs and resource sharing. Considering the problem of a HPC datacenter where jobs needs to be scheduled in order to optimise utilisation, we develop a

Reinforcement Learning (RL) algorithm to model a co-scheduling policy, sufficiently assertive to minimize idle capacity processing, while observing jobs' constraints such as total runtimes. The potential benefit of more assertive schemes is huge as current practice often leaves computational units in this setting idling for about 40% [Tir+20; Rei+12]. Our work maps this problem to a stateful reinforcement-learning problem resulting in a novel co-scheduling architecture, while matching the theoretical optimal guarantees with a practical algorithm that improves cluster utilization by up to 25%, with 10% reductions in queue makespan.

The rest of this paper is organized as follows. Section 2 describes the overall HPC and RL contexts, and illustrates some challenges. Section 3 describes our proposed architecture, whereas its evaluation follows in Section 4, with discussions in Section 5. Related work comes in Section 6, and conclusions and future work in Section 7.

2 Background and Challenges

Cloud datacenters are designed to respond in real time to users scattered around the world and using cell phones, tablets, or PCs. Low latency response is a much different environment than batch processing. HPC environments are managed by centralized batch systems [Reu+18; JYG], and commonly require users to describe jobs by the total run time and amount of resources such as number of CPUs, memory, and accelerators such as GPUs [FTK14]. As depicted in Figure 1, there is a resource reservation model where jobs arrive and wait to be scheduled in a queue until enough resources to match jobs' needs are available for use. The emphasis is on keeping CPU cores allocated, and on total throughput, with little attention to how long any particular job takes or how long one specific job may be delayed while other jobs finish first. In such environments, it can be useful to buffer up work and then schedule many jobs together at once, such as done in gang scheduling.

However, this model assumes the full capacity of allocated compute resources are used, which is rarely the case [Tir+20], specially at early stages of development. A way to maximize capacity is through consolidation, where jobs requiring complementary resources share the same (or parts of) physical servers. In HPC, consolidation happens mostly at the shared file systems and sometimes also at the network level. Resource managers such as Slurm allow nodes to be shared among multiple jobs, but do not dynamically adjust the preemption time slices. Consolidation is usually disabled due to the Service Level Objectives (SLOs) and user expectations on exclusive access to all allocated resources. Another level of complexity when scheduling regards heterogeneous requests, where multiple types of resources are allocated to a job, e.g. CPUs + GPUs. Although allocating heterogeneous resources is common in HPC, in here we focus on homogeneous allocations.

2.1 Resource Management with Reinforcement Learning

Reinforcement learning is defined as a problem with states, actions, and rewards, with state transitions that are affected by the current measured state, chosen actions, and

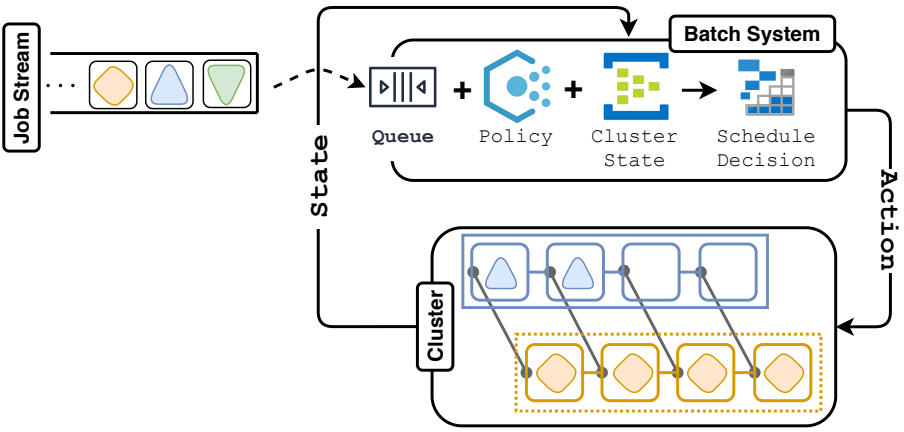


Figure 1: A normal architecture for a batch system such as Slurm. Green job (upside-down triangle) waits for resources even though some of them are not being utilized.

environment's rewards. These are embedded in RL's definition, which is formulated as a Markov Decision Process [Mon82]. Generally there are no stateless variants, but some related stateless problems such as bandit optimization. In resource management terms, we basically have a set of resources onto which jobs needs to be assigned according to an object function. This function can be any anything that can be optimized, such as an strategy to minimize the expected queue waiting time for a given job geometry (i.e. number of resources and total runtime). The scheduler's (learner's) task is to model the behavior of each available resource and/or application by interacting (exploring) with a system and observing its reactions. This interaction could mean, for instance, to collocating two jobs in the same server. After execution ends, the scheduler observes how long it takes to both of them to finish execution, and then calculates a *reward* (or alternatively, *loss*) to that interaction (action). The reward function describes how the agent (i.e. scheduler) "ought" to behave, with a "normative" evaluation of what an agent has to accomplish. There are no strong restrictions, but if the reward function describes well an observed behavior, the agent learns more optimally. This process can go for n jobs in a pool of resources being managed by a specific queue with a set of constraints, such as deadlines. Once enough learning has been collected, the learner starts exploiting the system in order to make better choices (future interactions), such as better co-placements to achieve higher rewards (or lower losses). In RL, the problem of balancing accuracy with discovery is known as the exploration-exploitation trade off [Thr92].

Finally, there are usually two scenarios: (i) Stochastic, and (ii) Adversarial. An instance of a stochastic setting (focus of this paper) happens when the scheduler places two jobs in a same server for a couple of times in the same configuration for all repetitions. There is a high probability that observations will be similar. This does not

happen in an adversarial setting, where in each repetition is different, as jobs would (hypothetically) compete for resources by learning the behavior of other jobs in order to adapt against it. The task of the scheduler then is to learn this malicious adversarial behavior to minimize losses to both jobs simultaneously.

2.2 Challenges

Default HPC resource managers such as Slurm do not profile jobs, nor take job-specifics into its scheduling decisions. Multiple extensions have been proposed. Some suggest the use of more flexible policies [Zha+19], commonly motivated by future exascale applications expected to be highly dynamic [Rod+17], where systems need to handle the amount of orchestration needed due to thousands of processes being spawned at runtime [Cas+18]. Jobs come with several constraints as they are tightly coupled in nature, requiring periodical message passing, synchronization barriers, and checkpointing for fault-tolerance [Gai+19]. These characteristics and observations such as idleness in the utilization of resources create a margin to improve resource efficiency by learning algorithms, which can allocate spare resources – ineffectively used otherwise – to other applications with similar dynamic characteristics, such as scientific workflows.

Different extensions to the main scheduling scheme are of direct relevance in practical scenarios. Server *capacity* (a resource) can be viewed as single dimensional, but in practice it is usually defined as a multivariate function depending on CPU, memory, network, I/O (Input/Output) utilization. All of these have capacity fluctuations over time, as they may similarly be shared among other datacenter users. As such, a practical and natural extension to this problem is to generalize its formulation to multi-dimensional cases since one-dimensional packing algorithms can be extended to multiple dimensions by vector packing methods [LTC14]. Additionally, placing one or more jobs together affect their performance as a whole [JR17]. That means that the actual utilization of capacity of a job depends on the collocated jobs. For example, cache misses (and the consequent I/O operations) might depend on all active jobs.

2.3 The Adaptive Scheduling Architecture

Before digging into our co-scheduler architecture, it is useful to explain its predecessor algorithm, ASA: The Adaptive Scheduling Algorithm [Sou+20]. ASA is a scheduling strategy to reduce perceived waiting times as well as to optimize workflows resource usage. In addition to that, ASA is an architecture that can be used to allocate and control resources to workflows, vertically and horizontally adjusting it at runtime. It uses Mesos [Hin+11] to encapsulate stage tasks into cgroup [Men07] containers, therefore enabling a fine-grain control of assigned resources through the operating system (OS). Within ASA, there is a unified view layer that bridges the management of the resources made available in the cluster, or through low level resource manager layers such as Slurm. Essentially, the application only sees a global pool of resources, which can be used freely according to application needs. ASA handles scheduling,

fault tolerance, resource isolation and control (among collocated tasks), elasticity, and other user defined policies.

Moreover, as an algorithm, ASA models the likelihood of a set of actions, each representing possible environmental outcomes. Assume there are m potentially good actions $\{a\}$, as for example, m different time estimations for possible waiting time in a Slurm priority queue[1], say $m = 4$. ASA tries then to learn which of the actions in an arbitrary four dimensional vector – e.g. $(1s, 10s, 100s, 1000s)$ – works best as a waiting time estimate for a given job geometry request (i.e. user request for total number of cores and estimated runtime for its job allocation). Rather than focusing on one specific time such as averaging each observed queue waiting time, ASA distributes the alternatives following a probability vector. When a particular alternative action works well, ASA makes this probability p_t to tend to such alternative in the vector. Generally, ASA aims to achieve an acceptable mixture on accuracy and exploration, because it needs to detect changes in the queue workload and embed them in its estimations.

ASA is a good choice when considering a setting where a sequence of jobs is submitted in a way to simultaneously maximize overall throughput and minimize overall resource usage [Sou+20]. These jobs are represented as stages of a scientific workflow, where a stage (job) can only start execution once an earlier (associated) stage is finished. Specifically, resources for upcoming stages are allocated so that a previous required stage finishes execution as close as possible to the time resources for an upcoming stage get allocated.

3 A Co-Scheduler Algorithm and Architecture

In this section we introduce a new HPC co-scheduler, and since it has an important extension in comparison to ASA, in here we are calling it ASA$_X$. ASA$_X$'s main difference to ASA concerns its inability to handle states, a crucial aspect needed to enable the use of dynamic scheduling strategies and an intelligent exploitation of the space between user requirements and the compute power available in clusters. For example, a tradeoff can be allowing two or more jobs to simultaneously run in a server (i.e. time-sharing) and then evaluating this in relation to running the jobs in isolation (i.e. space-sharing). These tradeoffs have to be monitored and controlled in a way that mitigate the impacts on current cluster policy. In here, the policy we compare with are time-sharing, and space-sharing, where resources are not time-shared between two jobs. As mentioned in the previous section, space-sharing guarantees predictable performance levels. Hence, a scheduler which learns over time has to minimize the accumulated losses incurred from the use of alternative policies. Main ASA$_X$ end goals are higher throughput and utilization, with performance degradation controls, in here measured by time to complete and job slowdowns.

[1] A priority queue differs from a classical *queue* in that jobs are primarily served based on priority, and secondarily on order of arrival. In Slurm this concept is termed as *partition*.

3.1 Algorithm

In here, we extend ASA to situations where each case has an associated state $\mathbf{x} \in \mathbb{R}^d$. That is, at any time t a scheduling decision has to be made, one action a (out of m) is taken with the system in state $\mathbf{x}$. Action a taken while in state $\mathbf{x}$ incurs a loss $\ell(a, \mathbf{x})$. The overall idea is represented in Algorithm 1, where we have a double loop: the inner loop (starting in line 3) iterates over job cases until the collected loss exceeds a threshold. The outer loop (starting in line 1) initiates and updates the parameters for each scheduling decision that can be made, as for instance from the job stream. Now, instead of having the vector $\mathbf{p} \in \mathbb{R}^m$ as in ASA, let $\mathbf{p} : \mathbb{R}^d \to \mathbb{R}^m$ be a function of states $\mathbf{x} \in \mathbb{R}^d$, and learning then means approximating this mathematical functional relationship. The central quantity is however the *risk*, defined here as a vector in $\mathbb{R}^n$ where

$$\mathbf{r}_i = \ell(a)\mathbf{p}_i(a) \tag{1}$$

We maintain at each scheduling iteration t a vector $\mathbf{r}_t$ of the total risk accumulated in that round. At each time t that a scheduling decision has to be taken, we let $\mathbf{r}_{t,i}$ denote the accumulated risk of the ith expert. We then describe a strategy that performs almost as well as

$$\min_{*} \sum_{s=1}^{t} \sum_{j=1}^{n_s} \mathbf{p}_*(\mathbf{x}_{sj})\ell(a_{sj}) \,. \tag{2}$$

Note the strategy minimising $\mathbf{r}_t$ corresponds to the maximum likelihood estimate, meaning we want to follow the expert that assigns the most probability to an action with the lowest loss. However, there may be cases where a scheduling decision goes wrong, therefore we set a threshold $\mathbf{r}_t$ to bound the accumulated risk for all bad scheduling decisions. Finally, we prove that the *excess risk* E_t after t scheduling decisions is bound, meaning the algorithm converges to an optimal point in a finite time after few iterations (See Appendix).

3.2 Experts

There can be an arbitrary number of experts, and each can be described by anything which can be easily evaluated (computed), from functions to decision trees (forming a forest). However, the combination of all experts needs to approximate the current state of the environment 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 from the allocation is used, or parallel, where all cores are used), time `interval` since the job started execution (e.g. 25%, 50%, 75%). In addition to these metrics, we also define `Hp(t)`, a `happiness` metric at time t for a given job as

$$Hp(t) = \frac{|t_{Deadline} - t| * \#RemainingTasks}{\#Tasks/s}. \tag{3}$$

Devised from a similar concept found in [Lak+15], the happiness metric relates the remaining time to job completion (also called deadline) with its remaining amount of

Algorithm 1 ASA_{X}

Require: Initialise $\alpha_{0i} = \frac{1}{n}$ for all $i = 1, \ldots, n$.

1: **for** $t = 1, 2, \ldots$ **do**
2: Initialise the total risk $\mathbf{r}_{ti} = 0$ for all $i = 1, \ldots, n$ (all experts).
3: **while** $\max_i \mathbf{r}_{t,i} \leq 1.0$ **do**
4: Measure the state $\mathbf{x}$ for this case.
5: Compute all n experts $\mathbf{p}_i(\mathbf{x})$, each one providing a vector in $\mathbb{R}^m$.
6: Form the mixture $\mathbf{p} = \sum_{i=1}^{n} \alpha_{t-1,i} \mathbf{p}_i(\mathbf{x}) \in \mathbb{R}^m$.
7: Choose action a according to vector $\mathbf{p}$.
8: Score its loss $\ell(a)$ for the case.
9: Update as $\mathbf{r}_{ti} \leftarrow \mathbf{r}_{ti} + \mathbf{p}_i(a)\ell(a)$ for all $i = 1, \ldots, n$.
10: **end while**
11: Update as

$$\alpha_{t,i} = \frac{1}{N_t} \alpha_{t-1,i} \exp\left(-\gamma_t \mathbf{r}_{ti}\right)$$

for all $i = 1, \ldots, n$. Here N_t is a normalising factor so that $\sum_{i=1}^{n} \alpha_{t,i} = 1$.
12: **end for**

work to complete execution and throughput. It enables a way to assess the resource capacity sensitivity for a running application. As defined in Equation 3, the happiness metric is useful in particular when the exploration process is happening in Algorithm 1. Given the remaining time ($t_{Deadline}$) and performance ($\#Tasks/s$, or throughput), if `Hp(t)` is near, or greater than 1.0, then it can be inferred that the job is likely to complete execution successfully; else (if `Hp(t)` < 1.0), the job is likely to fail.

Finally, it is reasonable to let states be represented by decision trees (DT) for each of the above metrics (`CPU%`, `Mem%`, `type`, and `interval`), all combined with the `happiness` metric. This is represented and structured in Figure 2. For instance, a DT can evaluate if `CPU%` is high (e.g. > 0.75), and then checking if `Hp(t)` is near 1.0, in which case one could expect high performance degradation due to job collocation. Conversely in the same DT, when `CPU%` is low, and `Hp(t)` is greater than 1.1, then the performance degradation is likely to be small. Rather than describing these relations explicitly, we work with a mixture of different decision trees (line 5 in Algorithm 1). Hence, $\mathbf{p}_{i,j}(\mathbf{x})$ represents how likely an action a_j should be taken given the evaluated state x_t, according to the i-th expert in DTi. For instance, if there are four actions representing the performance degradation likelihood in `Hp(t)` (i.e. $\mathbb{P}(Hp(t) \leq 1.0)$), the `DT1` may evaluate how a state x_t is mapped into a distribution of performance degradation $\mathbf{p}_{i,j}(\mathbf{x})$ for the situation `CPU% > 0.75`, and `Hp(t)` $\cong$ `1.0`, which could return $\mathbf{p}_1(\mathbf{x}) = (0.6, 0.3, 0.05, 0.05)$, meaning action a_0 is likely the decision to be taken. Another DT2 could evaluate something else related to memory, or to how a given special resource behaves, etc, returning a different $\mathbf{p}_{i,j}(\mathbf{x})$ and impacting the final $\mathbf{p}$ accordingly (line 6 in Algorithm 1).

The loss can then be taken proportionally to the actual runtime a workflow takes to complete execution. For example, if a user requests a walltime of 100s, and the

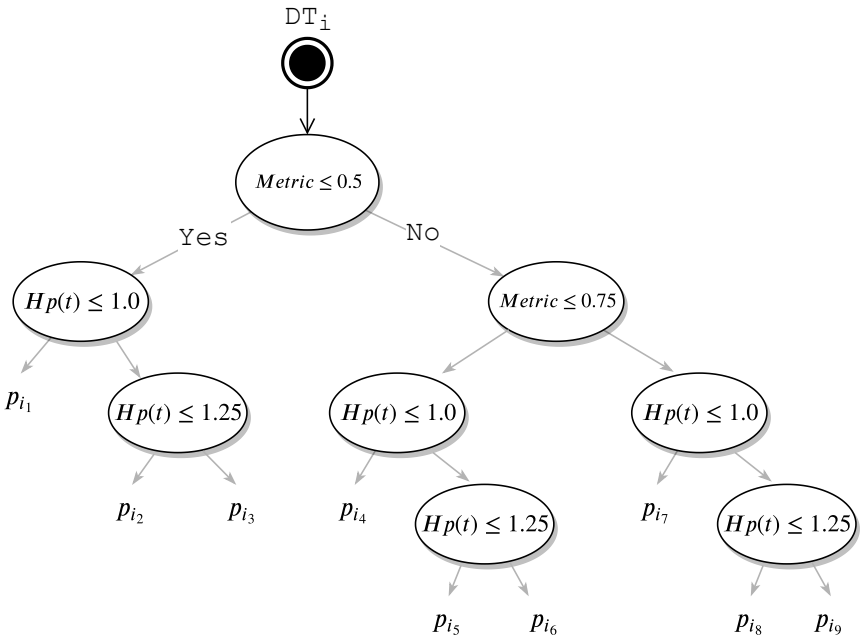


Figure 2: Decision tree (DT) structure used to evaluate a state expert. An action strategy is devised by combining four different distributions p_i, one among nine ($p_{i_1}, ..., p_{i_9}$) for each DT 'Metric': `CPU%`, `Mem%`, `type`, and `interval`.

workflow finishes in 140*s*, the loss would be proportional to 140-100s. On the other hand, if finishes in 70s, then the loss is proportional to 100-70s. In here actions $(a_1, a_2, \ldots, a_m)$ now correspond to a discretisation of resource allocation rather than to waiting times as in ASA, and they can be in terms of CPU% allocations, in 10% intervals, i.e. $0, 10, \ldots, 90\%$. This means fractions of an ongoing allocation are re-allocated to other jobs, and ASA_X then learns the distribution $\mathbf{p}$ over those m actions a through co-scheduling decisions and their collected rewards.

3.3 $\text{ASA}_\mathbf{X}$: A co-Scheduler Architecture

By leveraging on experts information (see previous subsection), where actions and the risk history are logged (see line 9 in Algorithm 1), we can combine such capabilities in a per-job basis to develop an architecture for a co-scheduler aimed at improving datacenter utilization and throughput. This is represented in Figure 3, contrasting with the architecture presented in Figure 1 mainly by how the cluster policy is assessed and by how past actions are recorded to improve future ones. Note the cluster 'Policy' is now analytically described by means of *experts*, evaluated through decision trees as explained in the previous subsection. Another difference regards the addition of a

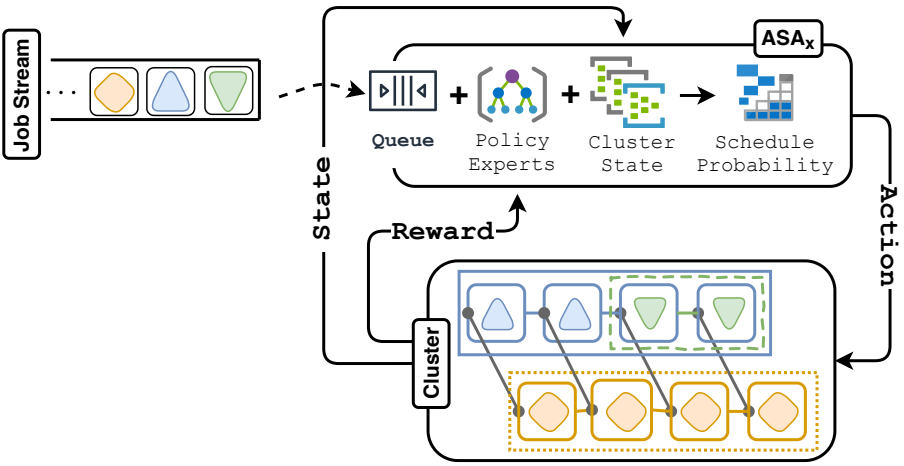


Figure 3: ASA$_X$ Architecture, where rewards are collected after scheduling actions enabling a fine grain control of the cluster Policy, described analytically by the experts. Note now the green job (upside down triangle) runs.

feedback loop, where rewards are registered for each scheduling action, therefore enabling a job-aware scheduler. This feedback-loop mechanism tracks mistakes, hence it also minimizes bad scheduling decisions overtime on a per-job basis. ASA$_X$ thus finds out which expert is the best one to maximize the reward for a given scheduling action.

In HPC, resource reservation is a basic need for rigid jobs which can only start execution once their resource needs are fulfilled (see Section 2), or *gang scheduled* [Fei96]. Although Mesos supports resource negotiation, it does not support gang scheduling, nor reservation based scheduling[2]. Therefore, ASA$_X$ also works as a framework to negotiate resource offers coming from Mesos in a way that satisfies jobs needs. ASA$_X$ basically withholds Mesos offers until enough resources fulfill job requirements, which is a version of the *First fit* capacity based algorithm [Fei96]. Although many different aspects also influence parallel job scheduling in general, such as data movement and locality, the present work does not (explicitly) tackle these. Our focus is on improving cluster throughput and utilization, which have direct impacts on reducing queue waiting times.

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

[2]Not to confuse with Mesos role-based reservation, which exclusively allocates resources to some jobs

setting of Slurm, and a static setting of ASA.

4.1 Metrics

The total runtime is measured by summing up the execution time of each stage in a workflow job. Another important metric is the response time, which is defined as the time elapsed between the submission (time when the job is ready to execute) to the time when the job finishes its execution. A related metric is the waiting time, which is the time a job waits in the queue before starting execution. Other metrics relates to CPU and memory utilization as measured by the Linux kernel, and not by scheduler allocation. These two last metrics are widely used as indicators for describing the general behavior of how resources are actually utilized by applications. They also indicate and help understanding if a workload mostly uses CPU or memory, and enable finer grained schedulers such as ASA_X to model applications more precisely.

4.2 Computing System

The experimental evaluation runs over a NumaScale system [Rus13] comprising 6 nodes with two 24-cores AMD Opteron Processors 6380, with 185 GB memory each. The NumaConnect forms a single system with a total of 288 cores and 1.11TB of memory equipped with an on-chip, distributed switch fabric 2D Torus network, supporting sub microsecond latency between nodes. The NumaScale storage uses a XFS file system, providing 512GB of storage. The system runs a CentOS7 (Kernel 4.18), with Slurm 18.08 with its default backfill scheduling plugin loaded.

4.3 Applications

Four different scientific workflows were selected for comparing ASA_X to default Slurm and static ASA scheduling strategies, as explained earlier in this section: (i) Montage, (ii) BLAST, (iii) Statistics, and (iv) Synthetic.

Montage [Ber+04] is a I/O intensive application that constructs the mosaic of a sky survey. The workflow has nine stages, grouped into two parallel (first two, and fifth) and two sequential (third and fourth, and last three) stages. All runs of Montage construct an image survey from the `2mass` Atlas images.

BLAST [Alt+97] 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 and 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 [KJ11]. The statistics workflow is composed mainly of a two sequential and two parallel

stages, intertwined, spending most processing time exchanging messages among parallel tasks.

Synthetic is a two stage workflow composed of a basic sequential and a parallel stages. Synthetic is simultaneously data and compute intensive workload. The data intensive part consists of filling up the memory with over one billion floating points, prior to a compute intensive part which calculates the values of these floating points sum and multiplication. The first and second stages differs only on how they use resources, sequentially and in parallel, respectively.

4.4 Workloads

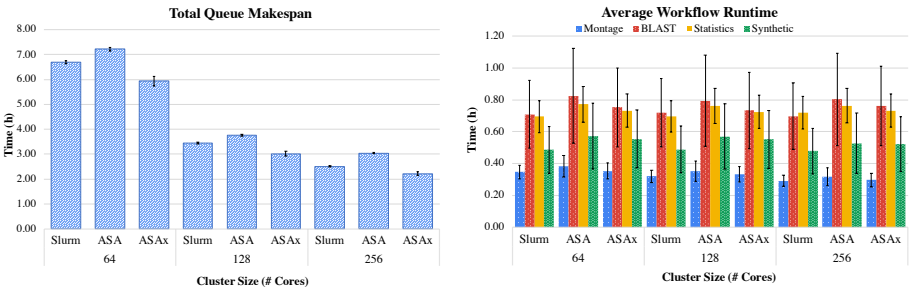
In order to demonstrate and compare ASA$_X$'s co-scheduling and adaptability features, we run it against a static configuration of ASA and a default space-sharing Slurm configuration and backfilling enabled. We run the same set of workflows three times in three different cluster sizes: 64, 128, and 256 cores. The workflows have different job geometry scaling requests ranging from 8 cores to up to 64 cores, consistently to the minimum cluster size (i.e. 64 cores), totalling 512 cores and 45 job submissions for each cluster size. The workload is statically set in the three cluster sizes to demonstrate how the different scheduling strategies work when faced with high (8x), medium (4x), and low (2x) cluster loads respectively. When comparing ASA static and ASA$_X$ against Slurm, neither of them can access more (and same) cores than the scaling factor set at the start of experiments. Besides that, Slurm statically allocates resources for the full job duration, regardless if some stages in a workflow require less resources to be processed. Moreover, when scheduling jobs with ASA it is allowed a maximum of two jobs to share a server, and the time-sharing CPU capacity is set at 50% for each application. Likewise, the same happens when scheduling through ASA$_X$, though the time-sharing CPU capacity is dynamically changed once rewards are collected. Due to increased analysis complexity, an allocation offered to a co-schedule job cannot be created with resources from multiple ongoing allocations, for both ASA and ASA$_X$. Finally, as mentioned in the previous section, the loss function in place to optimize co-scheduling decisions is calculated proportionally to the user requested walltime and to the actual workflow runtime.

4.5 Results

Figures 4(a) and 4(b), and Table 1 and 2 summarize all experimental evaluation.

4.5.1 Makespan and Runtimes

Figures 4(a) and 4(b) show respectively queue workload makespans and workflow runtimes, both in hours. It is possible to see that the total makespan reduces as the cluster size is increased. As expected, Slurm experiments are very contained (small standard deviations) because it uses the space-sharing policy and isolates jobs with exclusive resources throughout their lifespan. ASA, on the other hand, takes longer to



(a) Total workload queue makespan per strategy

(b) Average total runtime per strategy

Figure 4: Slurm, ASA, and ASA$_X$ Results - Total (a) Queue makespan and (b) runtime for different cluster sizes (64, 128, and 256 cores) and scheduling strategies (Slurm, ASA, and ASA$_X$).

complete the workload, because it does not do anything to understand resource usage by applications. As ASA basically takes deterministic decisions about scheduling decisions (i.e. allocate half capacity to each collocated application), its results are contained as well. ASA$_X$ however, reduces the overall workload makespan by up to 12% (64 cores) because it learns overtime which jobs fit well together. Due to the same reason, its scheduling decisions go wrong in the beginning, explaining the higher standard deviations shown in Figure 4(a).

In conjunction with Table 1, Figure 4(b) shows the 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). BLAST and Synthetic are two very scalable, CPU intensive workloads, which do not depend on I/O and network as Montage and Statistics do. ASA scheduling strategy has the highest standard deviations overall because its workload experiences more performance degradation when compared to both Slurm and ASA$_X$, and this is due to its static capacity allocation.

Table 1 also shows the normalized results for each strategy, besides resource consumption (CPU and Memory Utilization (%)). It can be seen that ASA$_X$ is near Slurm in average total runtimes, with overall 10% increase, and as low as 3% for Montage. A notable achievement for both ASA and ASA$_X$ relates to CPU utilization. Because on both strategies only a specific fraction of resources are actually allocated, the utilization ratio increases considerably. For instance, when a job requests 1 CPU, ASA allocates 50% of one CPU to the job. The other 50% is allocated to another job. For non CPU intensive workloads (Montage and Statistics), this strategy results in higher utilization ratios because such workloads fare less than the upper bound capacity, which is notable when comparing to Slurm submissions. For CPU intensive workloads such as BLAST, this strategy hurts performance, slowing down the workflow total runtime execution, and this can be seen in ASA results. However, as ASA$_X$ learns overtime that co-scheduling jobs with both BLAST and Synthetic results in low

Table 1: Workflows summary for Slurm, ASA, and ASA$_X$ in three different cluster sizes. CPU Util. and Mem Util. represent resource utilization (%) proportionally to total allocated capacity. Normalized averages are shown below results for each workflow. Acronyms: WF (Workflow), Stats (Statistics), and Synth. (Synthetic).

		Slurm			**ASA**			**ASA$_X$**		
WF	**Size**	**Runtime (h)**	**CPU Util. (%)**	**Mem Util. (%)**	**Runtime (h)**	**CPU Util. (%)**	**Mem Util. (%)**	**Runtime (h)**	**CPU Util. (%)**	**Mem Util. (%)**
Montage	64	0.35 ±4%	45 ±5	10 ±5	0.38 ±7%	94 ±5	10 ±5	0.35 ±5	95 ±4	10 ±4
	128	0.32 ±4%	33 ±5	7 ±3	0.35 ±6%	73 ±5	7 ±3	0.33 ±5	75 ±5	7 ±3
	256	0.29 ±4%	21 ±5	5 ±3	0.32 ±6%	55 ±5	5 ±3	0.30 ±4%	51 ±5	5 ±3
Normalized Average		-	-	-	*+10%*	*+230%*	*0%*	*+3%*	*+227%*	*0%*
BLAST	64	0.71 ±21%	96 ±4	44 ±5	0.82 +30%	99 ±1	44 ±5	0.75 ±25%	99 ±1	44 ±5
	128	0.72 ±22%	95 ±3	45 ±5	0.79 ±29%	99 ±1	44 ±5	0.73 ±24%	99 ±1	44 ±5
	256	0.70 ±21%	91 ±3	42 ±6	0.80 ±29%	99 ±1	43 ±6	0.76 ±25%	99 ±1	43 ±6
Normalized Average		-	-	-	*+19%*	*+5%*	*0%*	*+9%*	*+5%*	*0%*
Stats	64	0.69 ±10%	26 ±2	5 ±1	0.77 ±30%	51 ±3	6 ±1	0.73 ±11%	61 ±4	5 ±1
	128	0.70 ±10%	16 ±2	1 ±1	0.76 ±11%	36 ±6	2 ±1	0.72 ±11%	43 ±7	1 ±1
	256	0.72 ±10%	10 ±2	1 ±1	0.76 ±11%	26 ±2	1 ±1	0.73 ±11%	86 ±6	1 ±1
Normalized Average		-	-	-	*+10%*	*+36%*	*0%*	*+5%*	*50%*	*+1%*
Synth.	64	0.49 ±15%	99 ±1	95 ±3	0.57 ±21%	99 ±1	95 ±1	0.56 ±18%	99 ±1	95 ±1
	128	0.50 ±15%	99 ±1	93 ±4	0.57 ±21%	99 ±1	93 ±3	0.55 ±18%	99 ±1	93 ±4
	256	0.48 ±14%	99 ±1	92 ±2	0.53 ±19%	99 ±1	91 ±2	0.52 ±17%	99 ±1	93 ±1
Normalized Average		-	-	-	*+18%*	*0%*	*0%*	*+13%*	*0%*	*+1%*

rewards, its scheduling decisions become more biased towards allocations running the Montage and Statistics workflows. It is indeed notable the increase in CPU utilization for the Statistics workflow, as it is a non CPU intensive workload.

A final note should be made about the memory utilization. Since we are running these experiments in a NumaScale system, the whole memory (1.1 TB) is available to the jobs. The only workload capable of utilizing most of the memory available in the system is the Synthetic workflow. This property specifically made ASA$_X$ avoid co-placing jobs with the Synthetic job, as memory is one of the key metrics set in our decision tree expert evaluations (see previous section).

4.5.2 Aggregated Queue and Cluster Metrics

Although total runtime results are important from a user point of view, other aggregated cluster metrics such as queue waiting time are also important. This is summarized in Table 2, which shows average waiting times (h), cluster CPU utilization (%), and response times (h). The key point in Table 2 related to queue waiting time, which is reduced by as much as 50% in both ASA and ASA$_X$. Different from ASA, ASA$_X$

Table 2: Slurm, ASA, 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%
ASA	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%

has a reduced response time, showing it takes better co-scheduling decisions than Slurm and ASA simultaneously. Additionally, in order to improve job response times, ASA$_X$ reduces cluster (CPU) utilization, and also improves queue waiting times considerably, even when the cluster load is low (2x). On the other hand, Slurm decreases utilization as the load decreases, and is neither able to improve response time.

5 Discussion

The evaluation presents how ASA$_X$ combines application profiling with an assertive learning algorithm in order to simultaneously improve resource usage and cluster throughput, with direct impacts on the workload makespan. By combining an intuitive yet powerful abstraction through its decision tree experts, together with an agnostic (happiness) metric describing application performance, ASA$_X$ works very efficiently to co-schedule jobs while improving cluster throughput. The overall performance degradation experienced by ASA$_X$ (10% average runtime slowdown) is negligible when compared to its benefits, specially when it is often reported that users overestimate walltime requests. When compared to default Slurm, the common system used in HPC, ASA$_X$ achieves reduced average job response times of up to 10%, which can enable some users to achieve faster time to results in their projects. Similar results apply even when the cluster is faced with low loads, where the co-schedule actions save cluster resources that can be used to serve other incoming jobs.

It is important to note that current HPC scheduling strategies, specially the ones using 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 in users' workflow and job performance predictability. However, such strategies do not take advantage of current dynamic workflows, which are highly

adaptable to changes in resources, faults, and even on input accuracy. Nor it takes advantage of many Linux capabilities providing userspace and fine-grained control of process scheduling [Men07]. Even rigid and classical jobs may suffer from traditional space-share policies, because the reservation based model common in HPC does not take into consideration how resource capacity is consumed at runtime. This is due to the way the community evolved overtime and its relation with the CPU scarcity and the performance and consistency of applications. By precisely taking advantage of such workflow properties, and by using well established schedule controllers such as cgroups, ASA_X is able to improve the cluster utilization without hurting jobs performance. This can be a useful feature in NumaScale clusters, where vertical "true" scaling is the target, enabling applications to scale up without modifications. In these scenarios though, resource management is key, because the default Linux Completely Fair Scheduler (CFS) does not understand fine-grained job requirements, such as walltimes and data locality.

Finally, the proposed *happiness* metric (Eq. 3) can enable ASA_X to use it at runtime to mitigate and control the possible performance degradation due to bad placement decisions. It is important to note though, that the happiness metric assumes all tasks in the job are sufficiently homogeneous. This means that each task takes approximately the same amount of time to complete execution, which is the common case in most stages of a scientific workflow. As such, monitoring $Hp(t)$ before and after the co-placement of a job can also be useful to understand if such a decision actually will succeed. However, this is outside the scope of this paper and is planned as an natural extension to our co-scheduling algorithm as such feature can enable it to foresee the performance degradation on running applications. This can ultimately enable ASA_X to optimize its co-schedule actions already before collocation, respecting jobs even more efficiently. Another related path towards extensions is to generalize the ASA_X allocation capacity to accommodate more than two jobs simultaneously as long as they degrade each one's performance minimally.

6 Related Work

Batch schedulers such as Slurm [JYG] and Torque [Sta06] are the main resource managers in HPC datacenters. Most of them use resource reservation with backfilling strategies, and rely on users to provide application resource and walltime needs. Additionally, the granularity of resource allocation to jobs is set at the level of a full compute server. This entails that jobs fully utilize the allocated capacity constantly throughout their lifespan. However, more often than not, jobs utilize less than this upper capacity [Tir+20]. Some proposals have been vowing to extend this traditional model, with impacts to be studied [Ahn+14] in more depth. Public cloud datacenters, on the other hand, offer alternatives for running HPC workloads, such as Kubernetes [Bur+16], and Mesos [Hin+11]. Their internal policy assume low latency scheduling, offering fairness and resource negotiation. However, these systems lag on capabilities such as resource reservation, and still assume applications fully utilize allocated resources [Tir+20].

On another spectrum, stochastic schedulers have been proposed as solutions to overcome the highly overestimated user provided resource needs [Gai+19]. ASA_X can be used as an extension to such schedulers, because it offers a new scheduling abstraction through its experts, which can model stochastic applications well. Similarly to ASA_X, Deep RL schedulers have been proposed [Zha+19; LSL20; Mao+19], though they focus either on dynamic applications, or on rigid-jobs. As noted, a mix of a diverse set of applications are flooding new HPC infrastructure realizations. Some Machine Learning (ML) models have been proposed [CD17] that take advantage of the large dataset already available in current datacenters. Differently from ML, RL methods such as the one used in ASA_X and in [Zha+19] do not need any previous data to be fed into its logic to optimize scheduling. As mentioned in the previous sections, ASA_X is a stateful extension of authors' previous work [Sou+20], where the main difference is that ASA_X now supports past actions to be taken into consideration by taking a full scope of a RL approach. Finally, in regards to policy optimization, [Zha+19] selects among several of them to adapt the scheduling decisions. It is a similar approach to ours, although it does not consider co-scheduling for achieving queue waiting time minimization together 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, HPC systems today face a very diverse set of workloads with very dynamic requirements, such as low latency and streaming workflows coming from AI applications. These workflows are characterized by supporting many different features, such as system faults, approximate output, and resource adaptability. Additionally, current HPC workloads do not fully utilize the capacity provided by these high end infrastructures, impacting datacenter operational costs, besides hurting user experience due do long waiting times. In this paper, we proposed a HPC co-scheduler by using a novel, convergence proven, reinforcement learning algorithm. By analytically describing a co-scheduler policy through decision trees, ASA_X is able to optimize job collocation by profiling applications so to understand how much of an ongoing allocation can be safely reallocated to other jobs. By conducting real cluster experiments, we show that ASA_X is able to improve cluster utilization by as much as 30%, while also reducing queue waiting times, and hence improving overall datacenter throughput. Together with the architecture, our algorithm forms the base of an application-aware co-scheduler for improved datacenter utilization with minimal performance degradation.

Appendix A Convergence of $\text{ASA}_{\mathbf{X}}$

The excess risk is defined here as

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

where $\mathbf{x}_{sj}$ denotes the states of the jth case in the sth round, and where a_{sj} is the action taken at this case.

The excess risk of this algorithm is then 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

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

Then

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

Moreover

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

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

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

Application of Hoeffding's lemma gives then

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

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

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

Application of Abel's second inequality gives the result.

□

References

[Ahn+14] Dong H Ahn, Jim Garlick, Mark Grondona, Don Lipari, Becky Springmeyer, and Martin Schulz. "Flux: a next-generation resource management framework for large HPC centers". In: *2014 43rd International Conference on Parallel Processing Workshops*. IEEE. 2014, pp. 9–17.

[Alt+97] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs". In: *Nucleic acids research* (1997).

[Ber+04] GB Berriman, JC Good, AC Laity, A Bergou, J Jacob, DS Katz, E Deelman, C Kesselman, G Singh, M-H Su, et al. "Montage: A grid enabled image mosaic service for the national virtual observatory". In: *Astronomical Data Analysis Software and Systems (ADASS) XIII*. Vol. 314. 2004, p. 593. URL: `http://montage.ipac.caltech.edu/`.

[Bur+16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, omega, and kubernetes". In: *Queue* 14.1 (2016), pp. 70–93.

[Cas+18] Ralph H Castain, Joshua Hursey, Aurelien Bouteiller, and David Solt. "Pmix: process management for exascale environments". In: *Parallel Computing* 79 (2018), pp. 9–29.

[CD17] Danilo Carastan-Santos and Raphael Y De Camargo. "Obtaining dynamic scheduling policies with simulation and machine learning". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2017, pp. 1–13.

[Fei96] Dror G Feitelson. “Packing schemes for gang scheduling”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1996, pp. 89–110.

[FTK14] Dror G Feitelson, Dan Tsafrir, and David Krakov. “Experience with using the parallel workloads archive”. In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2967–2982.

[Gai+19] Ana Gainaru, Guillaume Pallez Aupy, Hongyang Sun, and Padma Raghavan. “Speculative Scheduling for Stochastic HPC Applications”. In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: ACM, 2019, 32:1–32:10. ISBN: 978-1-4503-6295-5. DOI: 10.1145/3337821.3337890. URL: http://doi.acm.org/10.1145/3337821.3337890.

[Hin+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.” In: 11.2011 (2011), pp. 295–308.

[JR17] Pawel Janus and Krzysztof Rzadca. “SLO-aware Colocation of Data Center Tasks Based on Instantaneous Processor Requirements”. In: *arXiv preprint arXiv:1709.01384* (2017).

[JYG] Morris A. Jette, Andy B. Yoo, and Mark Grondona. “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.

[KJ11] J Zico Kolter and Matthew J Johnson. “REDD: A public data set for energy disaggregation research”. In: *Workshop on Data Mining Applications in Sustainability (SIGKDD), San Diego, CA*. Citeseer. 2011.

[Lak+15] Ewnetu Bayuh Lakew, Cristian Klein, Francisco Hernandez-Rodriguez, and Erik Elmroth. “Performance-based service differentiation in clouds”. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2015, pp. 505–514.

[LSL20] Yuhao Li, Dan Sun, and Benjamin C Lee. “Dynamic colocation policies with reinforcement learning”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 17.1 (2020), pp. 1–25.

[LTC14] Yusen Li, Xueyan Tang, and Wentong Cai. “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, pp. 2–11.

[Mao+19] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. “Learning scheduling algorithms for data processing clusters”. In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 270–288.

[Men07] Paul B Menage. “Adding generic process containers to the linux kernel”. In: *Proceedings of the Linux Symposium*. Vol. 2. Citeseer. 2007, pp. 45–57.

[Mon82] George E Monahan. “State of the artâa survey of partially observable Markov decision processes: theory, models, and algorithms”. In: *Management science* 28.1 (1982), pp. 1–16.

[Pra+15] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V Kale. “A batch system with efficient adaptive scheduling for malleable and evolving applications”. In: *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE. 2015, pp. 429–438.

[Rei+12] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. “Heterogeneity and dynamicity of clouds at scale: Google trace analysis”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. 2012, pp. 1–13.

[Reu+18] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. “Scalable system scheduling for HPC and big data”. In: *Journal of Parallel and Distributed Computing* (2018).

[Rod+17] Gonzalo P Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. “Enabling workflow-aware scheduling on hpc systems”. In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 2017, pp. 3–14.

[Rus13] Einar Rustad. *Numascale: NumaConnect*. 2013.

[Sou+20] Abel Souza, Kristiaan Pelckmans, Devarshi Ghoshal, Lavanya Ramakrishnan, and Johan Tordsson. *ASA – The Adaptive Scheduling Architecture*. Research Report. Umeå University, 2020. URL: http://umu.diva-portal.org/smash/record.jsf?pid=diva2%5C%3A1423086.

[Sta06] Garrick Staples. “TORQUE Resource Manager”. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC ’06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0. DOI: 10.1145/1188455.1188464. URL: http://doi.acm.org/10.1145/1188455.1188464.

[Thr92] Sebastian B. Thrun. *Efficient Exploration In Reinforcement Learning*. Tech. rep. 1992.

[Tir+20] Muhammad Tirmazi, Adan Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. “Borg: the Next Generation”. In: *SIGOPS European Conference on Computer Systems (EuroSys’20)* (2020).

[Yan+13] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. “Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers”. In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 3. ACM. 2013, pp. 607–618.

[Zha+19] Di Zhang, Dong Dai, Youbiao He, and Forrest Sheng Bao. “RLScheduler: Learn to Schedule HPC Batch Jobs Using Deep Reinforcement Learning”. In: *arXiv preprint arXiv:1910.08925* (2019).