ASA - The Adaptive Scheduling Architecture

Abel Souza[1], Kristiaan Pelckmans[2], Devarshi Ghoshal[3], Lavanya Ramakrishnan[3], and Johan Tordsson[1]

[1]Department of Computing Science
Umeå University, Sweden
{abel,tordsson}@cs.umu.se

[2]Uppsala University, Sweden
kristiaan.pelckmans@it.uu.se

[3]Lawrence Berkeley National Lab., Berkeley, California
{dghoshal,lramakrishnan}@lbl.gov

Abstract: In High Performance Computing (HPC) infrastructures, resources are controlled by batch systems and may not be readily available, which can negatively impact applications with deadlines and long queue waiting times. In particular, this is noticeable for data intensive and low latency workflows where resource planning and timely allocation are key characteristics for efficient processing. On the one hand, allocating the maximum capacity expected for a scientific workflow guarantees the fastest possible execution time, at the cost of spare and idle infrastructural resources, as well as extended queue waiting times and costly resource usage. On the other hand, dynamically allocating resources according to specific workflow stage requirements optimizes resource usage, although it may also negatively impact the total workflow makespan. With the aim of enabling new scheduling strategies and features for scientific workflows, we propose ASA: the Adaptive Scheduling Architecture, a novel and convergence proven scheduling method to reduce perceived queue waiting times as well as to optimize resource usage and planning in scientific workflows. The algorithm uses reinforcement learning to estimate queue waiting times, and based on these estimates pro-actively submits resource change requests, with the goal of minimizing total workflow inter-stage waiting times, idle resources, and makespan. The algorithm takes into consideration both learning (the waiting times), and acts on what is learnt so far, and thus handles the exploration-exploitation trade-off. Experiments with real scientific workflows in two real supercomputers show that ASA combines the best of the two aforementioned approaches for resource allocation, with average workflows' queue waiting time and makespan reductions of up to 10% and 2% respectively, with up to 100% prediction accuracy, while obtaining near optimal resource utilization.

Key words: Scheduling, Workflows, HPC, Reinforcement Learning

1 INTRODUCTION

Large scale experiments model different aspects of nature such as weather forecasting, drug discovery, fluid dynamics, and many other scientific endeavours. Higher resolution sensors have been generating an ever larger amount of data, usually processed over large and complex computing infrastructures such as High Performance (HPC) and cloud computing datacenters. Due to its complexity with modeling and handling great amounts of data, such time consuming scientific campaigns are organized in independent data pipelines, known as *scientific workflows* [Tay+07]. Figure 1 exemplifies parts of the Montage Workflow, an image mosaic engine [18]. A scientific workflow is composed of sequentially interconnected stages (the different colors in Figure 1), where each stage is responsible for a specific set of tasks inside the overall application data flow. Moreover, scientific workflows are not only common in HPC centres, but also virtually in every sector of industry and academia, where they are used for analyzing and correlating data for predictions and decision support.

Intrinsically, a stage in a workflow structure describes its scalability and the amount of resources required to perform all of its tasks. To ensure acceptable task performance during workflow execution is the responsibility of the developers and the workflow management systems (WMS). When time to scale the developed workflow comes, users make use of HPC infrastructures. However, HPC platforms are primarily designed to support monolithic applications and provide a static allocation scheduling model i.e., the resource allocation is fixed throughout the entire job lifespan [Jha+14; Reu+18; Sch+13; Com+16]. This methodology guarantees good performance, however it results in fragmentation and lower datacenter efficiency due to underutilization. It also hinders the development of newer scheduling strategies needed in dynamic computational models, like data intensive and streaming workflows, increasingly used for conducting online and in-situ experiments [Dee+18]. These problems are likely to exacerbate with highly dynamic workflows in the next-generation exascale systems [Ber+08], expected to have applications issuing and orchestrating thousands of simultaneous processes [Cas+18]. With increasing use of workflows to process big amounts of data, a closer integration between the WMS and the datacenter resource manager (RM) is of vital importance for meeting scientific application constraints, like placement, resource isolation and control, turnaround times, and overall datacenter efficiency [Dee+18; Com+16; Asc+18].

In this paper we propose ASA: the Adaptive Scheduling Architecture for Scientific Workflows. Leveraging conceptual ideas from distributed operating systems [Hin+11], ASA decouples application development and scheduling planning from resource management. Packaged as a library, ASA presents applications with resources from multiple job allocations as one global pool of resources. This allows workflow management systems to be fault-tolerant, elastic, besides enabling the use of new scheduling strategies. ASA pro-actively estimates the waiting time for coming stages in a workflow during the currently executing stage to improve workflow turnaround times. In this way, ASA not only optimizes total resource usage, but it also reduces the total workflow makespan. For estimating how an user waits in the queue, we designed a

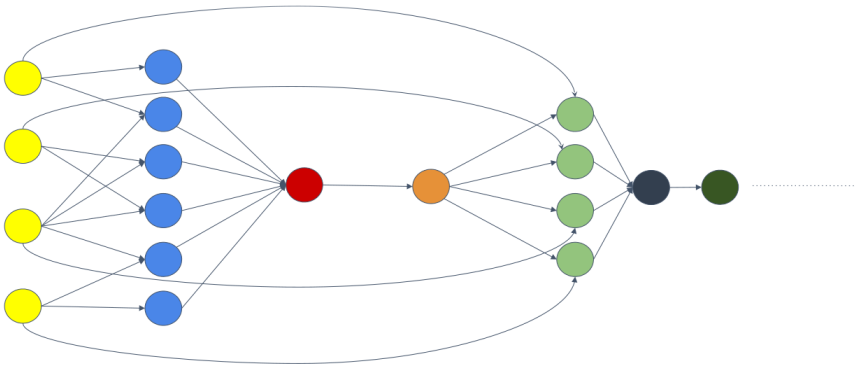


Figure 1: Montage scientific workflow pipeline structure (snippet), an image mosaic software used at NASA [Ber+04]. Each color in the graph describes a set of specific tasks within a stage. Each stage produces outputs used as inputs at subsequent stages that produce the final result at the end.

simple Reinforcement Learning (RL) algorithm, which can adapt to the current state of a queue. This amounts to both learning (the waiting times), and acting on what is learnt thus far, and amounts hence to a realization of the exploration-exploitation trade-off. Experiments with real workflows in real supercomputers show that ASA achieves a middle ground between the two aforementioned ways for resource allocation: lower total turnaround times, with near optimal resource utilization.

The rest of this paper is organized as follows. In Section 2, we describe in more details the characteristics of scientific workflows, their scheduling trade-offs, past work, and challenges. In Section 3, we present ASA, an architecture and algorithm for resource orchestration, management, and planning. Experiments, evaluations, various analyses, and discussion follow in Sections 4 and 5. Finally, we present our conclusions in Section 6.

2 Background and Related Work

The nature of scientific models is very complex and thus projects are often organized in distributed collaborations. Rather than developing large monolithic applications, scientists use *scientific workflows*: runtime systems for describing and executing applications as pipelined distributed components. For example, data modeling, staging, handling, processing, and pre- and post-processing are concrete tasks that may occur before, within, or after these pipeline stages. Figure 1 shows the Montage workflow, an image mosaic application used at NASA [18] with seven sequential stages, each colored differently. Edges describe sequential data dependencies, where data outputs

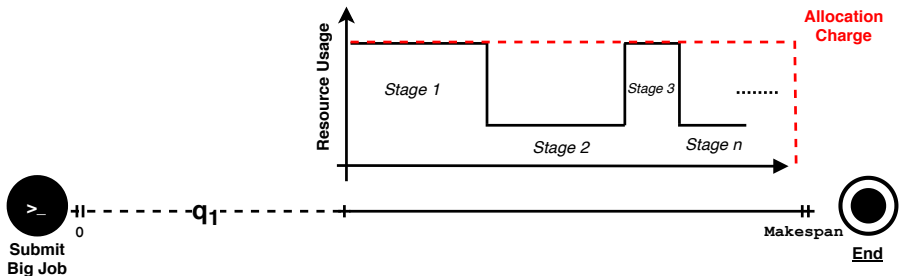
of a previous stage are sent to following stages produced at the end of such (previous) stages. The number of nodes (shown as color circles in Figure 1) in a stage describes its scalability: one node means the stage is inherently sequential, using only one available resource (e.g. CPU/core, GPU, etc.), whereas two or more nodes mean parallel stages that may use more than one resource. Streaming workflows are used in in-situ and online experiments, where all stages run concurrently and data (known as *tuples*) are continuously streamed over the workflow pipeline and processed by each stage as they arrive from predecessor stages [TBR11].

2.1 Related Work

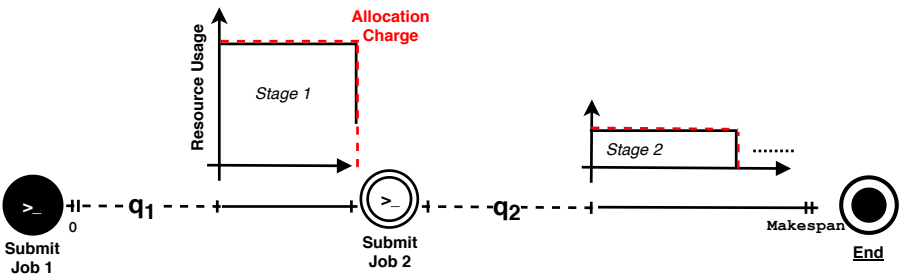
Scientific workflows are orchestrated, scheduled, and managed by a WMS, realized by programming language extensions through Application Program Interfaces (API), or by new dialects of common programming languages [Ams+16]. WMS's are used as execution engines for helping users and developers to run, scale, and integrate the distributed components of a workflow. Some WMS examples are Apache Taverna [Oin+04], Kepler [Alt+04], Pegasus [Dee+04], and Tigres [Hen+16]. Because these systems do not assume specific runtime behaviors like resources' performance variability, most of them do not support Quality-of-Services (QoS) application requirements. To overcome such limitations, VGrADS [Ram+09] combines resources from different providers into a single virtualized abstraction layer to enable applications with smarter scheduling and fault-tolerant strategies. Additionally, new tools enable WMS's with stage elasticity, achieving optimal resource expenditures, though with larger makespans [Fox+17].

In environments like HPC centers, jobs have to wait in (priority) queues for resources before starting execution [GC07]. Thus, a natural way to improve total workflow makespan and to enable deadline planning features, is to estimate the queue waiting times. For this, three main approaches have been used: (i) simulating scheduling according to the job queue (at certain point in time), (ii) statistical modeling, and (iii) a mix of these two [SY10]. Queue simulation (i) is a way to predict waiting time, but can be challenging if estimations needed at runtime do not take into account future (non-deterministic) job submissions from other users, which may degrade predictions. Although (i) can be used as a baseline for comparisons with more elaborate methods, and though a normal user may have access to the queue statistics for doing so, static methods can be seen as non adaptive if they do not adjust to such queue workload changes.

Traditional Machine Learning models (ii) tend to overfit the wait time because the dataset used for training can rapidly change. Its application without understanding the system and its workload does not work well as boundaries and medians tend to produce great over-estimations on the waiting time. QBets [NBW07] used to be a reference system, but it is not in production anymore due to today's workload high dynamicity with sudden changes, well captured with time-series analysis. QBest predictions were not bounded, and did not take into account variables that affect the job wait time. QBets' solution ('quantile prediction') is quite different from ASA (see



(a) Big Job Strategy



(b) Per-stage Strategy

Figure 2: (a) Big Job vs (b) Per-Stage managed resource allocation strategies in HPC. Fig. 2(a): an unique allocation for the entire workflow duration, with single queue waiting time. Fig. 2(b): per-stage allocations with only as many resources as required by a particular stage, with extra inter-stage queue waiting times. Note the differences in makespan and resources charging in each case (summation of area(s) under the dashed red lines).

Section 3), as QBet is based on traditional learning, not online learning. A recent solution is implemented by Karnak [09]. In this work, a large number of variables are used to model the wait time, including seasonal patterns, current system load, queue composition, job geometry, particular user, particular queue, particular group, etc. A decision tree is used to classify jobs according to different criteria, and then the resulting bag of jobs are modelled individually, giving better precision. In a second version, wait times are improved with scheduling simulations. Although not perfect, this model was shown to achieve much better results than QBets. Suggestions on the future work of Karnak [WSN16] point towards the application of neural networks.

2.2 Scheduling Tradeoffs for Scientific Workflows

In this subsection, we formulate scheduling tradeoffs for different strategies when submitting workflow jobs to HPC environments. We analyze the differences in total resource expenditure and workflow turnaround time.

Commonly, users submit scientific workflow jobs to HPC clusters using two different strategies, as shown in Figure 2. In Figure 2a, a workflow job is submitted as a big allocation (Big Job Strategy in the figure). Mathematically, the core-usage (C) is defined as $C = n * t$, where n is the number of cores assigned during job execution, and t is the allocated time (often measured in hours, and specified as *core-hours*). Hence, for a workflow with s stages and each needing time t_i to execute, the total core-hours usage is calculated as the sum of core-hours used by each stage as

$$C_{BigJob} = n * \sum_{i=1}^{s} t_i. \tag{1}$$

Because stages with different resource requirements are not taken into consideration, the maximum amount of resources n is allocated for the entire duration of the workflow lifespan. This wastes resources (white areas above the resource usage black, and under the dashed red lines in Figure 2a), but guarantees lower total workflows' execution times. Alternatively, users can manually manage the different stages in the workflow by submitting them as multiple sub-jobs. E-HPC is a library that does exactly this, providing elasticity for workflows running over HPC resources [Fox+17]. Figure 2b shows this per-stage resource assignment. The change in the amount of resources occurs at the end of each stage, where a coming stage is assigned with the exact number of cores required for its execution. Thus $C_{Per-Stage}$, the core-hours usage in a per-stage managed workflow with s stages, is calculated as

$$C_{Per-Stage} = \sum_{i=1}^{s} (t_i * n_i), \tag{2}$$

where t_i and n_i respectively represent the time and the number of resources needed to execute the i-th stage. Comparing definitions (1) and (2), per-stage management results in lower total core-hour usage iff the accumulated sum of cores needed at each stage$_i$ is lower than n, or: $\sum_{i=1}^{s} n_i < n$. It follows that any workflow with one or more sequential stages and at least one parallel stage can have optimal core-hour usage if per-stage management is used [Fox+17].

Although per-stage management provides lower resource usage, it may negatively affects the total turnaround time (also known as the *makespan*). Workflow turnaround time (T) can be defined as $T = t + q$, where t is the workflow execution time, and q is the queue waiting time. Because resource allocation is performed for each stage in per-stage management, the makespan can be estimated as $T_{Per-Stage} = \sum_{i=1}^{s} (t_i + q'_i)$, where t_i and q'_i are respectively the execution times and queue waiting times of the i-th stage. With a *BigJob* scheduling strategy, and assuming $t = \sum_{i=1}^{s} t_i$, the workflows' makespan is estimated similarly to T above. Thus, for a per-state management to have lower makespans, the accumulated sum of its waiting times q'_i has to be lower than the single waiting time q_1 in the *BigJob* strategy, i.e., $\sum_{i=1}^{s} q'_i < q_1$. One strategy to achieve this is to heuristically pack multiple stages within medium-sized job submissions [ZKC09], though it may not achieve optimal resource usage. Finally, as the queue waiting time is a system parameter controlled by the resource manager, another natural strategy for the users is to observe its behaviour and estimate it.

2.3 Challenge: Waiting Time Estimation

The clear tradeoffs analyzed in the previous subsection show that, in one hand, submitting a large job for execution may have a long single waiting time, with the potential side effect of idle resources during sequential stages. At the expense of inefficient resource usage, these two characteristics achieve the minimum application runtime possible. On the other hand, submitting many pilot jobs separately (composing each stage) has the advantage of efficient resource usage as it uses per-staged allocations (as done in [Fox+17]). However, the extra inter-stage waiting times increase the workflow makespan, specially if it interweaves many stages with different resource requirements. A way to mitigate this would be to estimate the queue waiting time, with a proactive submission strategy that uses such estimations for coming stages, requesting needed resources during the execution of ongoing stages, thus resulting in a minimization of the accumulated inter-stage waiting times [ZKC09]. However, depending on the estimation accuracy, three outcomes are possible: (i) perfect estimation, (ii) overestimation, and (iii) under-estimation. In (i), resource usage and workflow makespan would be optimal. In (ii), resource usage would be optimal, but a probable increase in workflow makespan would be seen (though less than achieved by per-stage allocation). In (iii), resources would be ready for use before they are actually needed, and depending on the policy used to mitigate the extra costs regarding this, both the resource usage and workflow makespan would increase. Some resource managers (such as Slurm [JYG]) allow job dependency constraints to be specified, including when a job may start execution in case previous jobs have not finished execution.

3 ASA: the Adaptive Scheduling Architecture

In this section we describe the proposed architecture offering a global and unified view of resources to the application, and the proposed algorithm being used to estimate user's queue waiting time for upcoming workflow stages.

3.1 Architecture

Figure 3 illustrates the unified view presented to applications. Within ASA, the *Unified View* layer bridges the management of the physical resources made available through a low level resource manager like Slurm. Essentially, the application only sees a global pool of resources, where each one can be used freely according to the application's needs. By extending upon Mesos [Hin+11], a distributed resource manager, ASA handles scheduling, fault tolerance, resource isolation and control (among collocated tasks), elasticity, and other user defined policies. Mesos was chosen due to its simplicity and non-intrusiveness at managing resources, allowing users to pack it as a library which can be dynamically loaded. Moreover, similar resource managers require administrative capabilities to perform similar features, diminishing their portability and usefulness in restricted environments such as HPC clusters.

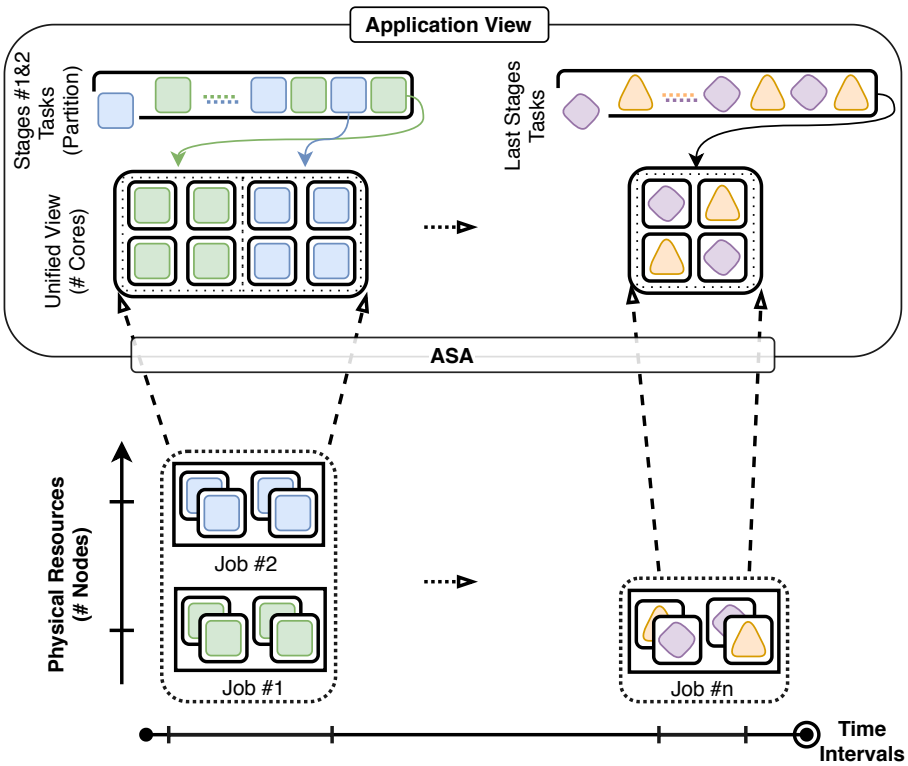


Figure 3: ASA - Architecture managing the physical resources. Tasks (the different shapes in the partitions) from different jobs can access resources from multiple jobs. The unified view layer enables users to apply different scheduling strategies, such as pro-active job submissions.

Current workflows can be easily managed by, and submitted to Mesos. This can either be achieved through directly submitting the application through a default Mesos executor, or by extending the WMS's internal APIs, bridging them with Mesos and enabling it to manage all workflow's tasks. Specific scheduling and placement policies can be realized through a Mesos Framework, which is the implementation of a scheduler tailored specifically to an application (e.g. MPI, Spark, etc). Mesos then monitors task states (e.g. `RUN`, `COMPLETE`, `FAIL`, etc.) to handle problems such as task crashes, misbehaviours, and unresponsiveness. In each case, frameworks can trigger specific actions, e.g., asking for extra resources, or migrating a failed task to another resource. This model and its associated runtime system enable applications with enhancements such as fault-tolerance, resource isolation, performance control, and the development of novel scheduling algorithms. For example, one particular feature of this model could be for users who belong to a same project to dynamically share resources with one another and save on total resource consumption for the project. This

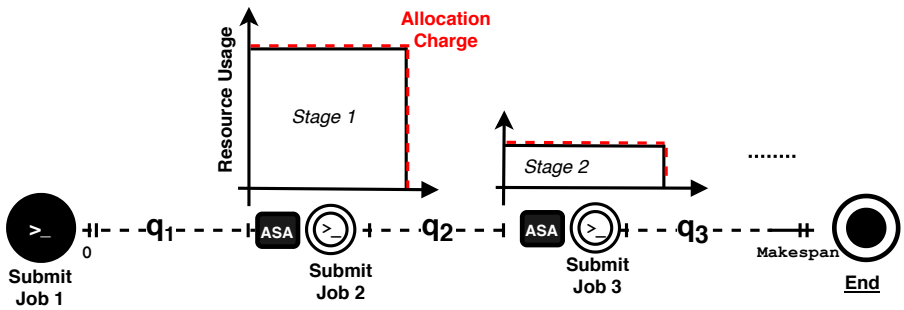


Figure 4: ASA - Algorithm workflow illustrating two concurrent pro-active submissions (2 and 3) within ongoing stages. Note the per-staged charging and lower workflow makespan.

can be extra useful at developing phases of a project, where trials and errors are the norm, and performed for testing and prototyping applications.

3.2 Algorithm

Figure 4 illustrates the algorithm's overall approach. The algorithm works by maintaining a distribution over a number m of fixed queue waiting times. For example, for $m = 4$, ASA tries to learn which of the four alternatives (indexes) in the vector $(1s, 10s, 100s, 1000s)$ works best as queue waiting time estimation for a given resource allocation request. Rather than focusing on one specific time such as averaging each perceived queue waiting time, ASA distributes the alternatives following a probability vector $p_t \in [0,1]^m$ with $\sum_{t=1}^{m} p_t = 1$. That is, in case one particular alternative works well, for instance index '1s' (or m_1), one wants probability p_t to tend to e_1 ($p_t \to e_1 = (1,0,0,0)$; the first unit vector in $R^{m=4}$) when the number of trials t goes to infinity ($t \to \infty$). Generally we aim to achieve a good mixture on accuracy and exploration, as a good algorithm needs to be able to detect changes in the queue workload and embed such behaviour in its predictions. In Reinforcement Learning, the problem of balancing accuracy with discovery is also known as the exploration-exploitation trade off [Thr92].

This methodology is applied in the following way. For each stage at iteration t, a waiting time a is estimated for a workflow stage$_y$ and used to submit job^y - the request for change of resources - at time $t_{y-1} - a$, where t_{y-1} is the expected end-date/deadline of an ongoing workflow stage$_{y-1}$ (Figure 4). If all goes as planned, it is expected that this pro-active job submission strategy minimizes the perceived waiting times between all workflow stages (see Section 2). However, if a workflow stage ends later, or an allocation gets assigned earlier than expected, resources may idle for some time before they can be effectively utilized. Conversely, if a workflow stage ends sooner, or resources become available later than expected, the total workflow process may

Algorithm 1 ASA - Adaptive Scheduling Algorithm

Require: Initialise $\mathbf{p}_0$ as $\mathbf{p}_{0a} \leftarrow \frac{1}{m}$ for all m actions a

1: **for** $t \leftarrow 1, 2, \ldots$ **do**
2: Initialise $\ell_{ta} \leftarrow 0$ for all a
3: **while** $\max_a \ell_{ta} \leq 1$ **do**
4: Sample action a according to vector $\mathbf{p}_t$
5: $\ell_{ta} \leftarrow \ell_{ta} + \ell(a)$ for action a
6: **end while**
7: Update

$$\mathbf{p}_{t+1,a} \leftarrow e^{-\gamma_t \ell_{ta}} \frac{1}{N_t} \mathbf{p}_{t,a}$$

for all actions a. N_t is a normalising factor so that $\sum_a \mathbf{p}_a = 1$
8: **end for**

take longer to complete. Both can be expressed in terms of a loss function $\ell_y(a)$, associated to the workflow stage$_y$ based on the waiting time a. This process is detailed in Algorithm 1 as follows.

Assume there are m potentially good actions $\{a\}$, as for example, m different time estimations for the queue waiting time. Assume also that after each application of an action a to a given case, one can score its loss $\ell(a)$. The ASA (Adaptive Scheduling Algorithm) consists basically of a double loop. The outer loop (line 1) iterates over mini batches of cases, referred to as *rounds* and collects as many cases in that round so that the total accumulated loss is bounded. The inner loop (line 3) iterates over such rounds and ensures that the vector $\ell_t \in \mathbb{R}^m$ collecting loss of the various actions $\{a\}$ is initialised properly before starting a new round, and that the vector $\mathbf{p}$ is updated properly after each round (line 7). e^{γ_t} is used as a non-increasing sequence, and guarantees the proven convergence of ASA (see Appendix A for more details and the mathematical proof). $\mathbf{p}$ is a distribution over all possible actions a that can be taken: after a while (when it has learned well), it peaks on the best action a_*, while in the beginning it is spread evenly over all actions a. In other words, in the beginning the algorithm *explores* options, like trying out random queue waiting times, or using the resource manager estimate features, while an *exploitative* stance is taken when enough evidence is collected, and more accurate estimations can be done. By following upon these principles, the coming section evaluates how all these concepts link together.

4 EVALUATION

In this section, we evaluate ASA's strategy with respect to workflows' total runtime, resource usage, queue waiting times, and makespan, all defined in the following subsections. We additionally evaluate Algorithm 1 convergence over time for three different estimation policies for a simulated scenario where the queue waiting time being experienced by the user changes at 5 different points in time (see Subsection 4.4). At

the end of this section, we evaluate ASA estimation accuracy and how it influences the perceived waiting time experienced by applications at runtime.

4.1 Metrics

The total runtime is measured by summing up the execution time of each workflow stage. The summation of each workflow's stage runtime multiplied by the amount of resources used in such stages, measures the total resource usage, or core-hour (measured in hours). We compare ASA with two different scheduling strategies: (i) traditional, Big Job allocation strategy; and (ii) dynamic, Per-stage job allocation. As explained in Section 2, the Big Job strategy (i) allocates the maximum capacity needed for the entire duration of the workflow, regardless of its stages' needs. The second strategy (ii), though, allocates resources to workflows in a per-stage manner, for the exact duration of each stage. Our proposed strategy ASA (iii) pro-actively submits resource changes for a coming stage during the execution of an ongoing stage (see Section 3). The total queue waiting times is thus calculated slightly different in each strategy: in (i), there will be only one queue waiting time (the first one), whereas strategy (ii) has one or more queue waiting times (one additional wait for each workflow stage). In ASA (iii), the waiting times are measured by the perceived queue waiting times (PWT), i.e., the time interval a coming workflow stage actually waited for resources after a previous stage finished (see Figure 4). As this waiting time overlaps with a previous stage execution, the perceived queue waiting time is potentially reduced and can be observed through the makespan metric. On the other hand, if the perceived queue waiting time is lower than expected, an extra corehour overhead (OH) loss might be incurred. Then, for each strategy, the total queue waiting time is calculated as the summation of all queue waiting times . Finally, the total makespan is calculated by subtracting the time the workflow is submitted for execution from the time the workflow successfully finishes execution. The total makespan takes into consideration all the inter-stage waiting times in each strategy. We evaluate these metrics for each strategy (Big Job allocation, Per-stage allocations, and ASA) by submitting three different scientific workflows, each with different resource usage and requirement profiles (see Subsection 4.3).

4.2 Computing Systems

In order to demonstrate and compare ASA's feasibility, adaptability, and generality features, we run a set of workflows in two different supercomputer centers, with different resource scaling factors: at HPC2N and UPPMAX. UPPMAX comprises 486 nodes with two 10-cores Intel Xeon E5 CPU (v4), with 128 GB memory each. UPPMAX's storage uses the Lustre file system and provides 6.6 PB of storage. The interconnect is Infiniband FDR, supporting a theoretical bandwidth of 56 Gb/s and a latency of 0.7 *ms*. All UPPMAX's nodes run CentOS 7, with Slurm 19.05 with its default fair-share scheduling policy.

HPC2N compromises 602 nodes with two 14-cores Intel Xeon E5 CPU (v4), with 128 GB of memory each, and similar Infiniband interconnection as UPPMAX's. HPC2N's storage also uses the Lustre file system, providing 2 PB of storage. All HPC2N's nodes run Ubuntu Xenial (16.04 LTS), with Slurm 18.08 with its default fair-share scheduling policy.

4.3 Applications

Three different, real scientific workflows were selected for comparing ASA to Big Job and Per-Stage scheduling strategies, as explained earlier in this section: Montage, BLAST, and Statistics.

Montage [18] is a data intensive application that constructs the mosaic of a sky survey. The workflow has nine ordered stages, grouped into two parallel (first two, and fifth) and two sequential (third and fourth, and last three) stages (Figure 1). All runs of Montage construct an image for survey `M17` on `band j`, `degree 8.0` from the `2mass` Atlas images.

BLAST [Alt+97] is a compute intensive applications that matches DNA sequences against a large ($>$ 6 GB) sequence database. The workflow splits an input file (of few KBs) into several smaller files and then uses parallel tasks to compare the input against the large sequence database. BLAST is composed of two main stages: one parallel and one sequential. The database is loaded in-memory on all compute nodes during the parallel stage. Finally, all the outputs from the parallel stage are merged into a single file (sequential stage).

Statistics [KJ11] is an I/O and network intensive application that calculates various statistical metrics (mean, median, average, standard deviation, variance, etc.) from a large dataset with measurements of electric power consumption in a household with an one-minute sampling rate over a period of almost 4 years. Different electrical quantities and some sub-metering values are available in a public dataset. The statistics workflow is composed mainly of a two sequential and two parallel stages, intertwined, consuming most of the processing due to communication among the parallel tasks.

Workflow configuration. For each one of the three strategies (Big Job, Per-stage, and ASA), these three workflows are submitted sequentially to the queue, concurrently one after the other. This was done using six different scaling factors: In HPC2N, workflows use 28, 56, and 112 cores, respectively; whereas in UPPMAX, workflows use 160, 320, and 640 cores, respectively. This combination creates a total of 54 different runs. For ASA's strategy, Algorithm 1's state is kept across different runs, meaning all of its variables are shared among the different workflow submissions. This allows the algorithm to converge and adapt itself more quickly to the current queue state, minimizing errors. Finally, in this evaluation the loss function $\ell_y(a)$ for a given job

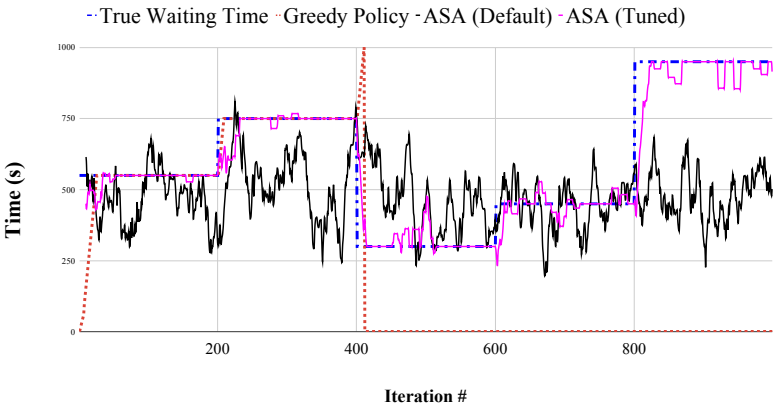


Figure 5: ASA's estimation convergence over time regarding queue waiting time (dark dashed blue line) with three different sampling policies: Greedy (red dotted line), ASA's default (black line), and ASA tuned (light pink line).

geometry y is defined as

$$\ell_y(a) := \begin{cases} 0, & optimal; \\ 1, & otherwise \end{cases} \tag{3}$$

where *optimal* means the algorithm sampled the best possible (closest to the true queue waiting time) action a among the m alternatives available, and thus it returns a loss of 0, and 1 otherwise. Although more complex functions could be used, choosing a simple loss function allows ASA's behaviour to be understood more easily. Moreover, queue waiting times can be very large at some supercomputer centers, as it depends on resource availability and on many job constraints. As mentioned in Algorithm 1, lengthier m's should theoretically return more accurate estimations. However, for the purposes of this evaluation and due to practical runtime reasons, m is empirically set to represent a maximum queue waiting time of $\sim$28 hours (100k seconds), since this was the maximum queue waiting time reported in Systems 1 and 2. Thus, the value of $m = 53$ is used in Algorithm 1 to split the possible range estimators in 53 time intervals representing possible queue waiting time alternatives. The alternatives cover multiples of 10's, 100's, 1k's, 10k's, and 100k time intervals (in seconds), with higher number of alternatives assigned to values 10's and 100's due to the higher queue waiting times variability usually faced by smaller jobs (with up to 112 assigned cores) usually falling down in these ranges. Finally, all three workflows use the Tigres WMS[1] for runtime execution, and the Per-Stage submissions use Tigres' E-HPC feature.

[1] http://tigres.lbl.gov

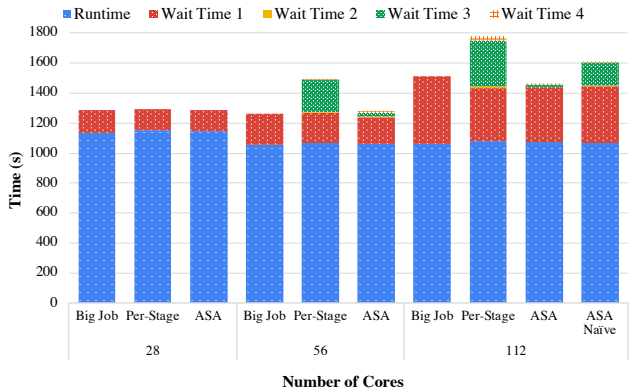
4.4 Convergence Results

Figure 5 shows a 1000 iterations simulation demonstrating how ASA (Algorithm 1) waiting time estimations converges in a hypothetical scenario where the true waiting time changes over time. To test ASA's adaptability capabilities, the true waiting time (blue stepped line) is randomly varied at five different occasions: at iterations numbers 0, 200, 400, 600, and 800. The default ASA policy (black thick line) takes rather too many iterations to converge to the true waiting time, which suddenly changes and worsens its convergence trend. It does so because it keeps exploring the interval space in order to validate its knowledge. However, with a tuned policy (pink thin line), where the perceived queue waiting times are used to randomly and repeatedly adjust the probability distribution p (used in Algorithm 1) with the calculated losses, the convergence velocity changes drastically. As it can be seen, at every true waiting time variation the tuned policy strategy enables ASA to converge to the true waiting time more rapidly. Even though, it still allows ASA to keep exploring the interval space, though it makes fewer miss predictions than the default sampling policy. A greedy approach is also shown (dashed red line), where the minimum perceived loss is always used for making estimations for the waiting times. Because the simple loss function $\ell(i)$ (see definition (3) in Section 3) is used, when the true waiting time suddenly drops, the greedy policy reaches a local minimum, and does not behave correctly afterwards, defaulting to a very conservative loss estimator (i.e., every proactive submission happens at the end of a stage, similarly to the Per-Stage's strategy) and thus behaving as if the algorithm was not used at all.

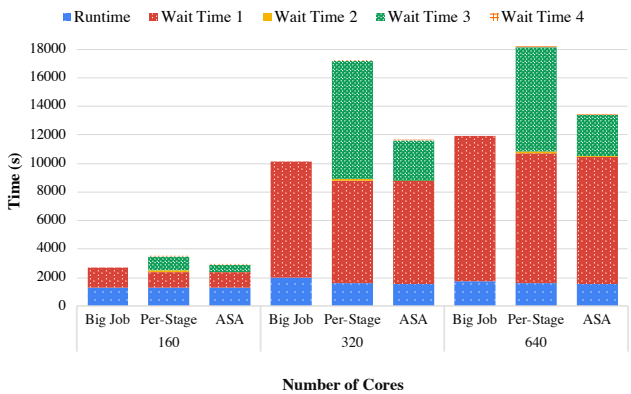
4.5 Sensitivity Analysis

By default, ASA uses the resource manager's helpers to set job dependencies. This allows ASA to specify different dependencies between sorted workflow stages with the advantage of optimizing resource usage, as a job cannot have its resources allocated (and thus charged) until all dependent jobs completed execution. However, this can also cause ASA to deviate towards non-optimal estimations, and furthermore randomly defer the start of jobs. Thus, to illustrate how ASA behaves in environments managed by resource managers with no support of job dependency helpers, and to calculate such impacts on the total workflow resource usage, an experiment without this setting is evaluated in HPC2N for the Montage workflow with 112 cores. This strategy is henceforce denoted *ASA Naïve*.

A large repetition number within the ASA tuned sampling policy has the effect of influencing (or biasing) ASA to follow the last observed waiting time, and thus this feature should be used with caution to not make ASA simply follow (or exploit) its first queue waiting time observations, devoting it of learning new outcomes and changes in the queue workload. In the following sub-sections, Algorithm 1 is tuned with a repetition parameter of 50, same value used for the previous simulation shown in Figure 5.



(a) HPC2N



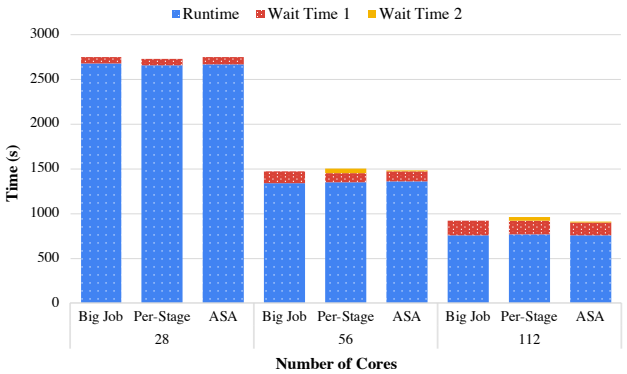
(b) UPPMAX

Figure 6: Montage Workflow - Makespan results for (a) HPC2N and (b) UPPMAX for different scaling factors (28, 56, 112, 160, 320, and 640 cores), and scheduling strategies (Big Job, Per-Stage, and ASA). Number of cores indicate peak allocations for a given strategy. ASA Naive strategy means no resource manager dependency setting is used.

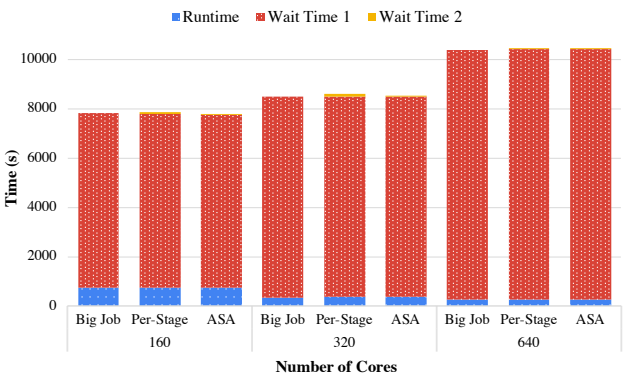
As a complement to these empirical results, we proof the theoretical convergence of the ASA algorithm in Appendix A.

4.6 Makespan Results

Figures 6, 7, and 8 show all three workflows' makespan breakdowns, showing the different inter-stage queue waiting times for each scheduling strategy (Big Job, Per-Stage, and ASA allocations), and for the six different number of cores (scaling, representing the peak allocations for each strategy). The figures are split in two columns,



(a) HPC2N

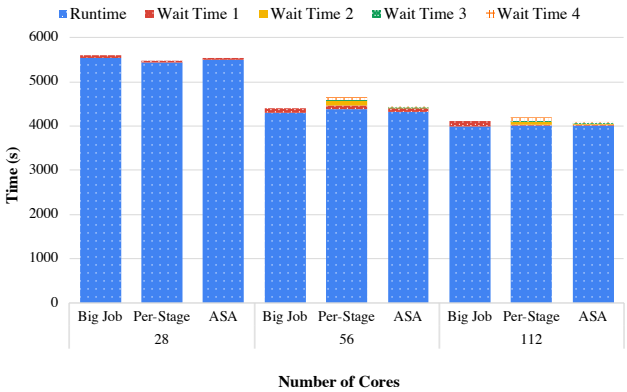


(b) UPPMAX

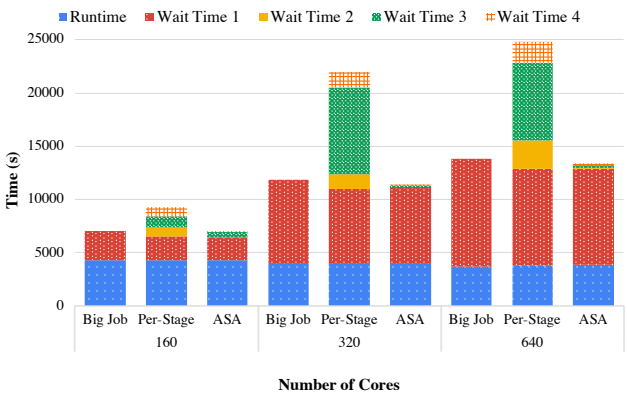
Figure 7: BLAST Workflow - Makespan results for (a) HPC2N and (b) UPPMAX for different scaling factors (28, 56, 112, 160, 320, and 640 cores), and scheduling strategies (Big Job, Per-Stage, and ASA). Number of cores indicate peak allocations for a given strategy.

(a) left and (b) right, which show results for HPC2N and UPPMAX, respectively. The difference in the queue waiting times between the two supercomputer centers are substantial, with higher waiting times in UPPMAX.

Figures 6(a) and (b) show the makespan for Montage. It is noticeable that early stages suffer from higher queue waiting times due to Montage beginning execution with a parallel stage. The total waiting time gets worse as the peak core allocation (Number of cores) scales, negatively impacting the Per-Stage strategy in every scenario with more than 28 cores. Due to multiple interactions of concurrently running workflows, dynamically updating how the queue is behaving (p distribution in Algorithm 1) are required. ASA's strategy leverages this information to submit resource changes earlier,



(a) HPC2N



(b) UPPMAX

Figure 8: Statistics Workflow - Makespan results for (a) HPC2N and (b) UPPMAX for different scaling factors (28, 56, 112, 160, 320, and 640 cores), and scheduling strategies (Big Job, Per-Stage, and ASA). Number of cores indicate peak allocations for a given strategy.

lowering the total waiting times. This is noticeable already at scaling to 56 and 112 cores, where the total makespan is lower than for the Big Job allocation. For Montage Naive, with 112 cores at HPC2N, there were additional delays as resources for the third stage got ready for use before the second stage had completed. In this case, ASA canceled the submission and re-submitted the job for the third stage, which incurred in an additional (perceived) queue waiting time as it can be noticed in the larger *Wait Time 3* in Figure 6(a).

However, the same effect does not happen to BLAST, as can be seen in figures 7 (a) and (b). As BLAST is a two stages workflow, where the first parallel stage is considerable larger than the second sequential one, the effects of using different scheduling

strategies are neglectable. Besides that, BLAST is a very scalable application, essentially keeping the resource utilization high as the core scaling factor increases. Furthermore, due to the higher queue waiting times at UPPMAX, BLAST's total makespan gets severely impacted. Finally, more notable dynamics are perceived in the Statistics workflow (Figures 8 (a) and (b)). This is a four stage and network intensive workflow, with large execution times in each stage. At first, in HPC2N (Figure 8(a)), the queue waiting time has limited impact on the total makespan. However, at the busier center (UPPMAX, Figure 8(b)) with larger queue waiting times, the learnt information gathered by other concurrent workflows allows ASA's pro-active system to essentially submit future resource change jobs earlier than even upcoming stages. This has significant effect, specially at 320 and 640 cores, where the queue waiting times are severely impacted for the Per-Stage strategy, which sometimes had twice the makespan of Big Job allocations.

4.7 Resource Usage and ASA Performance

Table 1 summarizes all runs and the measured metrics explained in previous sections. Below each workflow, a normalized average of collected metrics shows workflow's results in overview. This average is related to the lowest metric for each resource scaling row. Besides that, percentages inside parentheses represent the extra times incurred when comparing specific metrics with the best metric for that resource scaling.

It is visible that Per-Stage and ASA strategies provide the best resource usage in most scenarios, as shown in the table as Core-hour Usage. This comes, however, at the cost of extra workflow's makespan. Also, due to some variations in how long the workflows take to run, the resource usage in each strategy can vary. As noted in the previous subsection, the total makespan can be severely impacted when the amount of time an application waits in the queue is larger than the application's total execution time. As Montage is a not a scalable application, its execution time across different scaling factors does not considerably decrease its total execution time, and requesting larger amounts of resources actually impacts the total makespan negatively. This is illustrated by the Montage and BLAST workflows (320 and 640 cores), where the Per-Stage allocation strategy had 82% and 13% increase in the makespan. Pro-active ASA submissions reduced these severe extra times by 72% and 9% respectively, reducing large queue waiting time impacts as the normalized average makespans show. This behavior is particularly more noticeable for runs using the Per-Stage strategy than it is for ASA, as it learns about the queue's current state by observing the impacts other concurrent workflow submissions had. This information allows ASA to act to mitigate such severe impacts in earlier stages.

4.8 Prediction accuracy

In order to validate and quantify ASA's decision accuracy that has impact on its performance as a scheduling algorithm, additional experiments were conducted. Here,

Table 1: Experimental results respectively for Montage, BLAST, and Statistic Workflows in the six different core scalings. Bold values show best normalized results with relation to the three different scheduling strategies (Big Job, Per-Stage, and ASA allocations). Normalized averages are shown below results for each workflow (the lower, the better), with extra percentages (in relation to the *best* result achieved for any strategy) included inside parentheses (values over 1%).
Acronyms: WF (Workflow), TWT (Total Waiting Time), and CH (Core-Hour).

		Big Job Allocation			**Per-Stage Allocation**			**ASA**		
WF	**Cores**	**TWT (s)**	**Makespan (s)**	**CH (h)**	**TWT (s)**	**Makespan (s)**	**CH (h)**	**TWT (s)**	**Makespan (s)**	**CH (h)**
Montage	28	150 (+13%)	1287	9 (+24%)	258 (+95%)	1408 (+10%)	7	**132**	**1277**	**7**
	56	**206**	**1261**	16 (+32%)	426 (+105%)	1496 (+19%)	12	219 (+6%)	1280 (+2%)	**12**
	112	452 (+15%)	1513 (+3%)	33 (+65%)	699 (+78%)	1779 (+22%)	20	**393**	**1464**	**20**
	160	**1415**	**2718**	58 (+47%)	2220 (57%)	3507 (+29%)	40	1652 (+17%)	2921 (+7%)	**39**
	320	**8135**	**10126**	177 (+68%)	15582 (+91%)	17170 (+69%)	106	10062 (+24%)	11637 (+12%)	**105**
	640	**10200**	**11940**	309 (+83%)	16600 (+63%)	18200 (+52%)	171	11851 (+16%)	13436 (+12%)	**169**
Normalized Average		***+5%***	***+1%***	*+53%*	*+82%*	*+34%*	*0%*	*+10%*	*+6%*	***0%***
BLAST	28	70 (+3%)	2750	20	**68**	2727	**20**	75	2749	20
	56	133 (+20%)	**1476**	20	153 (+2%)	1508	21	**111**	1477	**21**
	112	165 (+15%)	926 (+2%)	23	194 (+35%)	965 (+6%)	24	**144**	**907**	**23**
	160	7100	7846	33	7125	7880	33	**7041**	**7800**	**33**
	320	**8133**	**8494**	32	8240	8611	33	8194	8557	**32**
	640	**10133**	10394	**46**	10150	10429	47	10144	10419	48
Normalized Average		*+6%*	***0%***	*+1%*	*+13%*	*+2%*	***0%***	***+2%***	*0%*	*+1%*
Statistics	28	52 (+8%)	5593 (+2%)	43 (+2%)	**48**	**5487**	**42**	51 (+6%)	5549	43
	56	**96**	**4397**	66 (+99%)	263 (+174%)	4644 (+6%)	34	116 (+21%)	4444	**33**
	112	124 (+77%)	4110 (+1%)	124 (+99%)	191 (+173%)	4193 (3%)	62	**70**	**4085**	**62**
	160	2772 (+2%)	7095 (+2%)	192 (+102%)	4960 (+83%)	9241 (+32%)	95	**2712**	**6986**	**95**
	320	7935 (+6%)	11886 (3%)	351 (+99%)	18008 (+141%)	21993 (+92%)	177	**7471**	**11439**	**176**
	640	10122 (+7%)	13868 (+4%)	665 (+97%)	21014 (+121%)	24817 (+86%)	**338**	**9497**	**13369**	344
Normalized Average		*+17%*	*+2%*	*+83%*	*+115%*	*+36%*	***0%***	***+5%***	***0%***	*+1%*

each job geometry related to each workflow described in Section 4.3 is submitted to their respective system 60 times, with a one minute time interval between submissions. This is done to capture variations in each system's queue workload, which affects the experienced waiting times. As in the previous evaluations, HPC2N handles all job geometries submissions with 28, 56, and 112 cores, whereas UPPMAX handles job geometries with 160, 320, and 640 cores. For each submission, waiting times are compared to ASA predictions of waiting time. Table 2 summarizes averages results for each workflow job geometry. In this table, the real waiting time (WT) averages actual queue waiting times (in hours), ASA WT averages predicted waiting times (in hours), and Perceived WT averages workflows' actual waiting times (in hours) are given. The impact of the predictions are also assessed as follows: Hit (the higher, the better) and Miss (the lower, the better) ratios represent the fractions of ASA's accurate- and over-predictions. The latter is increased when jobs need to be re-submitted due to larger predictions than actual WTs, over all job submissions. Misses and over-predictions impact total resource usage (measured in core-hours) because job allocations get assigned earlier than the estimates, causing extra job submission overheads (OH) when compared to resource usage for the Per-Stage strategy.

As it can be seen, there are high variations in HPC2N (Cores 28-112), whereas there are no misses (incorrect predictions causing re-submissions) at all for UPPMAX due

Table 2: ASA - Average results (with standard deviations) summary for Montage, BLAST, and Statistic workflows in six job geometries. Cores 28, 56, and 112 are HPC2N's, whereas 160, 320, and 640 are UPPMAX's.
Acronyms: WT (Waiting Time), PWT (Perceived Waiting Time), and OH (Core-Hour overhead in hours).

	Cores	Real WT (h)	ASA WT (h)	ASA PWT (h)	Hit Ratio (%)	Miss Ratio (%)	OH Loss (h)
Montage	28	0.4±0.3	0.7±0.6	0.5±0.4	60	40	1.7±0.5
	56	1.1±0.8	1.2±0.9	0.4±0.4	68	32	3.0±0.8
	112	1.5±0.7	2.0±1.9	0.5±0.4	87	13	2.0±0.8
	160	11 ±1.6	3.9±4.6	0.7±0.3	100	0	0
	320	15 ±1.3	12±3.9	0.2±0.3	100	0	0
	640	17 ±0.6	12±3.3	0.3±0.2	100	0	0
BLAST	28	0.4±0.3	1.0±1.0	0.6±0.3	70	30	8±1.9
	56	1.1±0.8	1.3±1.2	0.7±0.5	71	29	11±2.7
	112	1.5±0.7	1.0±1.0	0.6±0.4	89	11	3±0.7
	160	11±1.6	4.5±5.0	0.7±0.4	100	0	0
	320	15±1.3	11±4.1	0.2±0.3	100	0	0
	640	16±0.6	11±3.8	0.3±0.2	100	0	0
Statistics	28	0.4±0.6	0.5±0.7	0.4±0.4	67	33	3±0.2
	56	1.1±0.8	1.2±0.9	0.4±0.4	69	31	6±2.0
	112	1.5±0.7	2.0±1.9	0.5±0.4	87	13	5±1.0
	160	11±1.7	5.2±5.8	0.6±0.4	100	0	0
	320	14±1.3	11±3.9	0.2±0.3	100	0	0
	640	16±0.6	12±3.3	0.3±0.2	100	0	0

to its stability. This can be explained due to the higher fragmentation caused by smaller jobs and allocations. Notable, although there is a fair amount of misses for smaller job geometries (up to 112 cores) and thus job re-submissions for HPC2N, ASA still controls considerably the core-hour overhead losses. We remark that ASA achieves very good overall results on HPC2N, as summarized in Table 1, where ASA has very good makespan and resource usage results compared to the alternative strategies (Big Job and Per-Stage).

5 DISCUSSION

The evaluation illustrates how ASA combines a pro-active submission scheduling with Per-stage's strategy to simultaneously minimize resource usage and waiting times. ASA can be specially useful when a workflow has multiple large consecutive stages, where the impacts of waiting in a queue can overtake the usefulness of non-monolithic applications and workflows, represented by the Big Job allocations.

To summarize, Table 1 demonstrates that ASA achieves makespans close to those of Big Job allocations, while using as little resources as Per-Stage allocation strategy. Big Job strategy results in shorter makespans, but always end up in larger resource usage (core-hours). E-HPC's Per-Stage [Fox+17] strategy results in best resource usage and worse makespans. ASA simultaneously tackles both, with close-to-optimal makespans when compared to Big Job's and specially to Per-Stage (Figures 6-8), with best core-hour usage (Table 1). In real systems, job submission planning is key, and as explained in Section 4.3. To avoid both bad estimations and violating workflow's ordering constraints, ASA uses Slurm dependency features to link the various stages. Thus ASA shows no losses, except in ASA Naive (Figure 5a) which does not use such features. Although the architecture supports collocation of different workflow tasks in other's workflow allocations and resources (Figure 3), we preferred not managing resource allocations among different workflow stages. For instance, the architecture allows task co-placement from different workflows to share a same resource like CPU. As mentioned, this is supported by Mesos in a fine-grained manner as Mesos supports resource capacity scheduling constraints to be specified, like for example CPU utilization: if one task uses only up to 10% of a CPU resource, Mesos can co-schedule additional tasks in the same CPU up to a global threshold is reached (e.g. 100%). Although task co-placement optimizes overall resource usage if done correctly, it may have direct impacts on the time limits set by users, besides workflow performance impacts. As additional actions would need to be studied to safely support such proposition, we decided to not do it in this paper.

Table 2 shows how HPC2N and UPPMAX affect ASA predictions and resource usage fares. Although smaller jobs experience shorter queue waiting times, they experience variations of up to almost 1 hour in HPC2N's queue workload, ASA controlled quite notably the core-hour overhead losses (OH). A high queue variation negatively impacts ASA, causing its predictions to also vary largely during experiments because ASA has to adjust its probability distribution modelling the queue. For smaller job geometries ($\leq$ 112 cores), ASA has to acquire knowledge from a large number m (see previous Section) of alternatives until it can build-up knowledge for making accurate estimates. However, as the system's load varies aggressively, ASA has to adapt to such variations to bound the overhead losses. ASA is still able to reduce the perceived waiting times seen by workflows most of the time. The high variation in a queue usually happens due to fragmentation caused in the system by smaller job geometries with varied similar, but not identical constraints, something that larger jobs ($\geq$ 160 cores) in UPPMAX do not experience, and explains the high ASA accuracy in such system.

As explained in Section 4, ASA can be tuned to follow closely the last observed queue waiting time, which would change the results seen in Table 2, though its effects should be extensively studied in more specific scenarios. Modern schedulers like Slurm allow dependencies to be set among different jobs, and such features would mitigate the core-hours overhead caused by over-estimations, as can be seen in Table 1. Although job-dependencies enable overhead control, it may affect perceived queue waiting times because schedulers postpone job submissions until their dependencies are

set. Our experimental results from Table 2 show that ASA can be specially useful for large job geometries (achieving 100% accuracy), which can enable the resource planning capability as a feature.

Generally, our paper focuses on the first of two ASA features: to the best of our knowledge, a new, convergence proven (see Appendix A) Reinforcement-Learning method for estimating queue waiting times (WT) exclusively from user's perspective; a library for finer-grain management and scheduling of workflow's tasks (Mesos). Rather than using traces and/or resource manager's queue waiting time estimates (which can speed-up ASA convergence), we opted for real experiments in production systems. In this way we can evaluate how ASA would work as a general scheduling algorithm, and not only as a neat library enabling a diverse set of scheduling strategies to modern HPC systems. Results summarize experiments using 1000s of core-hours across two production HPC systems with large differences in architecture, users, workload, etc (Table 2). Algorithm 1 is a very simple method which adapts its knowledge and estimations by adjusting mini-batches (or rounds), resetting them when bad estimates are detected so to bound its losses. By sharing this information in a per job-geometry basis across different experimental scales, improvements in both systems are reported (Figures 5-8).

6 CONCLUSIONS

Keeping the highest possible application performance for timely processing with no wastage of resources has always been a challenge, and will be even more relevant for upcoming Exascale systems designed for low latency and highly dynamic data intensive workflows. These newer constraints demand novel combined solutions to classical problems such as queue waiting time predictions with adaptive, elastic, and fault tolerant architectural features. To tackle these, and leveraging on user perceived system's performance, we propose ASA: the Adaptive Scheduling Architecture. ASA learns and estimates the queue waiting times by using a novel reinforcement learning algorithm, which combined with its resource manager layer provides applications with the ability to dynamically adjust job resource planning based on workflow stage requirements. These allocations are done proactively based on the waiting time predictions to ensure that resources for subsequent workflow stages are available upon completion of ongoing stages. The evaluation based on three real workloads running with different job sizes on two different HPC systems demonstrates that ASA achieves makespans close to those of traditional, large job allocations. ASA's makespan averages only 2% higher than large allocations across all three workflows. Combined with a lower resource usage of Per-stage allocation, ASA achieves a total core-hour usage within 0.2% compared to optimal, Per-stage allocations, which is 43% less than the large allocations across all evaluated scenarios. For large job geometry submissions with lower queue workload variability, ASA achieves 100% prediction accuracy, while simultaneously minimizing overall resource usage even when faced with high queue workload variability. Future work points to extending ASA with statefulness, allowing

ASA to support different metrics and/or heterogeneity, and enabling yet more complex pro-active scheduling techniques and the support to multi-constraint/dimensional scheduling.

Appendix A Convergence of ASA

In this appendix we mathematically prove ASA's convergence towards the true waiting time, as shown in Figure 5.

Theorem 1 *Let $\theta = (\theta_1, \ldots, \theta_m) \in \mathbb{R}^m$ be a fixed, given collection of waiting time alternatives amongst which to choose. Let the ASA algorithm run on a sequence of t processes, and let $\eta(t)$ denote the number of mini-batches created by the algorithm as of time t. Then for any $\delta > 0$ with probability exceeding $1-\delta$, one has that*

$$\sum_{s=1}^{t} \ell_s\left(\theta^{s-1}\right) - \sum_{s=1}^{t} \ell_s(\bar{\theta}) \leq 4\eta(t) + \ln(m) + \sqrt{2t \ln\left(\frac{m}{\delta}\right)}. \tag{4}$$

Proof: The key to this proof is to consider two different timescales: (1) runs from $1, \ldots, t$ in a linear fashion, and (2) runs over the same range in a different fashion as follows. Let $m_k \subset \{1, \ldots, t\}$ such that each $m_k = a, \ldots, b$ and $\cup_k m_k = \{1, \ldots, t\}$. We refer to m_k as a mini-batch, or *round*, of length $|m_k|$. Consider a sequence $\{a_i, \ldots, a_t\}$ for any t, then

$$\sum_{s=1}^{t} a_s = \sum_{k=1}^{\eta(t)} \sum_{j \in m_k} a_j, \tag{5}$$

with $\eta(t)$ the number of mini-batches $\{m_k\}$.

Let (s) point to the last *completed* mini-batch m_k before iteration s. Rather than fixing the length of the mini-batches, the algorithm itself constructs the minibatches according to how well the learned solution is working. This extra layer of adaptivity enables the non-stationary setting.

Let θ^{s-1} denote the estimated waiting time, randomly sampled according to $\mathbf{p}_{(s-1)}$, which is implemented for process y_s, with $s = 1, 2, 3, \ldots, t$. Define $Z_t > 0$ as

$$Z_t \quad = \quad \sum_{\theta_i} e^{-\sum_{s=1}^{t} I\left(\theta_i = \theta^{s-1}\right) \ell_s(\theta_i)} \quad = \quad \sum_{\theta_i} e^{-\sum_{k=1}^{n^k} \sum_{j \in m_k} I\left(\theta_i = \theta^{j-1}\right) \ell_j(\theta_i)}. \tag{6}$$

Then

$$\ln\frac{Z_t}{Z_0} = \ln(Z_t) - \ln(Z_0) = \ln\sum_{\theta_i} e^{-\sum_{s=1}^{t} I\left(\theta_i=\theta^{s-1}\right)\ell_s(\theta_i)} - \ln(m)$$

$$\geq -\sum_{s=1}^{t} I\left(\bar{\theta}=\theta^{s-1}\right)\ell_s(\bar{\theta}) - \ln(m). \quad (7)$$

Conversely,

$$\ln\frac{Z_{\bar{k}}}{Z_{\bar{k}-1}} \quad = \quad \ln\frac{\sum_{\theta_i} e^{-\sum_{\bar{k}=1}^{\bar{k}}\sum_{j\in m_k} I\left(\theta_i=\theta^{j-1}\right)\ell_j(\theta_i)}}{\sum_{\theta_i} e^{-\sum_{\bar{k}=1}^{\bar{k}-1}\sum_{j\in m_k} I\left(\theta_i=\theta^{j-1}\right)\ell_j(\theta_i)}} \quad = \quad \ln\sum_{\theta_i}\mathbf{p}_{\bar{k}-1,i}e^{\left(-\ell_{(\bar{k})}(\theta_i)\right)}, \quad (8)$$

where we use the definition

$$\ell_{(\bar{k})}(\theta_i) \triangleq \sum_{j\in m_k} I\left(\theta_i=\theta^{j-1}\right)\ell_j(\theta_i). \quad (9)$$

Then using the inequality property $1-x \leq e^{-x} \leq 1-x+x^2$ for all $x \geq -1$, gives

$$\ln\sum_{\theta_i}\mathbf{p}_{\bar{k}-1,i}e^{-\ell_{(\bar{k})}(\theta_i)} \leq \ln\left(1-\sum_{\theta_i}\mathbf{p}_{\bar{k}-1,i}\ell_{(\bar{k})}(\theta_i) + \left(\sum_{\theta_i}\mathbf{p}_{\bar{k}-1,i}\ell_{(\bar{k})}(\theta_i)\right)^2\right)$$

$$\leq \ln e^{-\sum_{\theta_i}\mathbf{p}_{\bar{k}-1,i}\ell_{(\bar{k})}(\theta_i)+1}$$

$$= -\sum_{\theta_i}\mathbf{p}_{\bar{k}-1,i}\ell_{(\bar{k})}(\theta_i)+1, \quad (10)$$

where by construction $\left(\sum_{\theta_i}\mathbf{p}_{\bar{k}-1,i}\ell_{(\bar{k})}(\theta_i)\right)^2 \leq 4$ for any $\bar{k}$.

In conclusion,

$$-\sum_{s=1}^{t} I\left(\theta_i=\theta^{s-1}\right)\ell_s(\bar{\theta}) - \ln(m)$$

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

$$\leq -\sum_{s=1}^{t}\sum_{\theta_i}\mathbf{p}_{(s),i} I\left(\theta_i=\theta^{s-1}\right)\ell_s(\theta_i) + 4\eta(t), \quad (11)$$

or

$$\sum_{s=1}^{t}\sum_{\theta_i}\mathbf{p}_{(s),i}I\left(\theta_i=\theta^{s-1}\right)\ell_s(\theta_i)-\sum_{s=1}^{t}\ell_s(\bar{\theta})\leq 4\eta(t)+\ln(m). \tag{12}$$

So by defining the expectation at iteration s as

$$\mathbb{E}_s[\cdot]=\sum_{\theta_i}\mathbf{p}_{(s),i}\ell_\cdot(\theta_i), \tag{13}$$

one gets

$$\sum_{s=1}^{t}\mathbb{E}_s[\ell_s(\theta^{s-1})]-\sum_{s=1}^{t}\ell_s(\bar{\theta})\leq 4\eta(t)+\ln(m). \tag{14}$$

Finally, invoking Azuma's inequality [CL06] gives that with probability exceeding $1-\delta<1$, one has

$$\sum_{s=1}^{t}\ell_s\left(\theta^{s-1}\right)-\sum_{s=1}^{t}\ell_s(\bar{\theta})\leq 4\eta(t)+\ln(m)+\sqrt{2t\ln\left(\frac{m}{\delta}\right)}, \tag{15}$$

as desired.

□

References

[09] *Karnak from TeraGrid Round Table discussion*. 2009.

[18] *Montage - An astronomical image mosaic engine*. 2018. URL: `http://montage.ipac.caltech.edu/`.

[Alt+04] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher and Steve Mock. "Kepler: an extensible system for design and execution of scientific workflows". In: *Proceedings. 16th International Conference on Scientific and Statistical Database Management*. 2004.

[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).

[Ams+16] Peter Amstutz, Michael R Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich et al. "Common workflow language, v1. 0". In: (2016).

[Asc+18] M Asch, T Moore, R Badia, M Beck, P Beckman, T Bidot, F Bodin, F Cappello, A Choudhary, B de Supinski et al. “Big data and extreme-scale computing: Pathways to Convergence-Toward a shaping strategy for a future software and data ecosystem for scientific inquiry”. In: *The International Journal of High Performance Computing Applications* (2018).

[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*. 2004.

[Ber+08] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller et al. “Exascale computing study: Technology challenges in achieving exascale systems”. In: *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep* (2008).

[Cas+18] Ralph H Castain, Joshua Hursey, Aurelien Bouteiller and David Solt. “Pmix: process management for exascale environments”. In: *Parallel Computing* (2018).

[CL06] Fan Chung and Linyuan Lu. “Concentration inequalities and martingale inequalities: a survey”. In: *Internet Mathematics* (2006).

[Com+16] Isaías Comprés, Ao Mo-Hellenbrand, Michael Gerndt and Hans-Joachim Bungartz. “Infrastructure and api extensions for elastic execution of mpi applications”. In: *Proceedings of the 23rd European MPI Users' Group Meeting*. ACM. 2016.

[Dee+04] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi and Miron Livny. “Pegasus: Mapping scientific workflows onto the grid”. In: *European Across Grids Conference*. Springer. 2004.

[Dee+18] Ewa Deelman, Tom Peterka, Ilkay Altintas, Christopher D Carothers, Kerstin Kleese van Dam, Kenneth Moreland, Manish Parashar, Lavanya Ramakrishnan, Michela Taufer and Jeffrey Vetter. “The future of scientific workflows”. In: *The International Journal of High Performance Computing Applications* (2018).

[Fox+17] William Fox, Devarshi Ghoshal, Abel Souza, Gonzalo P. Rodrigo and Lavanya Ramakrishnan. “E-HPC: A Library for Elastic Resource Management in HPC Environments”. In: *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science*. WORKS '17. 2017.

[GC07] Francesc Guim and Julita Corbalan. “A job self-scheduling policy for HPC infrastructures”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2007.

[Hen+16] Valerie Hendrix, James Fox, Devarshi Ghoshal and Lavanya Ramakrishnan. “Tigres workflow library: Supporting scientific pipelines on hpc systems”. In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE. 2016.

[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: (2011).

[Jha+14] Somesh Jha, Jian Qiu, Andre Luckow, Pradeep Mantha and Geoffrey C Fox. “A tale of two data-intensive paradigms: Applications, abstractions, and architectures”. In: *IEEE BigData Congress, 2014*. 2014.

[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.

[NBW07] Daniel Nurmi, John Brevik and Rich Wolski. “QBETS: queue bounds estimation from time series”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2007.

[Oin+04] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat et al. “Taverna: a tool for the composition and enactment of bioinformatics workflows”. In: *Bioinformatics* (2004).

[Ram+09] Lavanya Ramakrishnan, Charles Koelbel, Yang-Suk Kee, Rich Wolski, Daniel Nurmi, Dennis Gannon, Graziano Obertelli, Asim YarKhan, Anirban Mandal, T Mark Huang et al. “VGrADS: enabling e-Science workflows on grids and clouds with fault tolerance”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM. 2009.

[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).

[Sch+13] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek and John Wilkes. “Omega: flexible, scalable schedulers for large compute clusters”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 351–364.

[SY10] Subhashini Sivagnanam and Kenneth Yoshimoto. “TeraGrid Resource Selection Tools: A Road Test”. In: *Proceedings of the 2010 TeraGrid Conference*. TG ’10. 2010.

[Tay+07] Ian J Taylor, Ewa Deelman, Dennis B Gannon, Matthew Shields et al. *Workflows for e-Science: scientific workflows for grids*. Vol. 1. Springer, 2007.

[TBR11] Rafael Tolosana-Calasanz, José A Bañares and Omer F Rana. "Autonomic streaming pipeline for scientific workflows". In: *Concurrency and Computation: Practice and Experience* (2011).

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

[WSN16] Jungha Woo, Shava Smallen and J.P. Navarro. *Improving Karnak's Wait Time Predictions*. 2016. URL: `https://www.xsede.org/ecosystem/science-gateways/gateways-symposium`.

[ZKC09] Yang Zhang, Charles Koelbel and Keith Cooper. "Batch queue resource scheduling for workflow applications". In: *IEEE International Conference on Cluster Computing and Workshops*. IEEE. 2009.