Using Reinforcement Learning to Size Tasks for Scientific Workflows

Nicolas Zunker

n.zunker@campus.tu-berlin.de

November 30, 2021

BACHELOR'S THESIS

Distributed and Operating Systems Chair

Technische Universitaet Berlin

Examiner 1: Prof. Dr. Odej Kao

Examiner 2: Prof. Dr. Volker Markl

Advisor: Jonathan Bader

Declaration

I hereby declare that I have created the present work independently and by my own without illicit assistance and only utilizing the listed sources and tools.

Hiermit erklaare ich, dass ich die vorliegende Arbeit selbststaandig und eigenhaandig sowie ohne unerlaubte fremde Hilfe und ausschliesslich unter Verwendung der aufgefaehrten Quellen und Hilfsmittel angefertigt habe.

Die selbstaendige und eigenstaendige Anfertigung versichert an Eides statt:

Berlin, den 30. November 2021

Nicolas Zunker

Acknowledgment

I would like to thank:

Firstly, my advisor Jonathan for his helpful advice, constructive feedback and assistance.

Then Prof. Dr. Toussaint, Prof. Dr. habil. Kao and Prof. Dr. Zubow, whose courses I enjoyed immensely and from whom I learned a lot.

And finally my friends and family for their support.

NZ

Abstract

The growth of computational power and big data has led to a massive increase in demand for computing services by users who aim to process large datasets. Particularly in the natural sciences it has become common for scientists to break down their computing needs into a sequence of smaller tasks, a so called workflow. These workflows can then be run on a variety of different execution platforms, depending on the users needs. One of the most pertinent fields for this is Bioinformatics.

Since workflows are composed of segregated inter-dependent tasks which can run in their own containers, the individual tasks which make up a workflow can be assigned a fraction of the computational resources available to the entire execution platform and doing so intelligently could improve efficiency and performance.

This thesis aims to investigate the allocation of resources to individual tasks, and specifically how reinforcement learning can be applied to aid in choosing more efficien allocations. The implementation of a reinforcement learning solution will be integrated into source code of the popular scientific workflow management system nextflow and tested against common bioinformatic workflows. Two different reinforcement learning approaches (Gradient Bandits and Q-Learning) will be compared and their performance will be judged both against eachother and against the performance of the task's default resource configurations.

Should such an approach prove fruitful and provide an improvement in resource usage efficiency it would naturally indicate this is an area which should be explored further and that the use of these methods can improve the performance of scientific workflows. This would be helpful to both the scientists which use workflow managers as well as the ownders and administrators of the execution platforms on which they run.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

This chapter gives a general introduction to the content of and the motivation for this thesis. This is done by initially outlining the motivation and problem in section 1.1, defining the goal in section 1.2, and explaining the structure of the rest of thesis in section 1.3.

1.1 Motivation & Problem Description

The rising popularity of scientific workflows and the enormity of the aggregate resource requirements of their constituent tasks [14] presents an opportunity to try and fine tune the resources allocated to these tasks in order to achieve better performance. Since a scientific workflow is composed of many tasks which each have unique relationships between the resources they require, their input, and how it affects their performance, it becomes cumbersome or even futile to try and hand-pick the most efficient resource allocation for each individual task. Furthermore as the tasks themselves or their order in a workflow might change over time it is also difficult to adapt to this. In addition to all of this it also quite common that the users may not have the requisite knowledge to properly size the tasks themselves and tend to make poor estimations [21] of what a given task's resource

requirements are. Finally it must also be considered that for a given use case the exact topology of the deployment scenario (i.e. whether it is being run on an individual computer, in a grid or cluster, or on a cloud computing platform) and the hardware infrastructure being used will also influence the performance of the workflow and its constituent tasks. When one considers all of these factors it becomes clear that fine tuning the resources for each of the tasks within such workflows "by hand" is difficult and complex.

However it is also obvious that sizing tasks properly is of critical importance. Specifically for large scale, distributed computations the resources being requested are so large that even small improvements in efficiency represent a large amount of reclaimed resources which could be allocated to other users or could reduce costs (in a scenario where computational resources are being paid for by the workflow user).

Thus this presents a scenario in which although better task sizing is beneficial for everyone it is not a problem with a simple solution.

At this point it becomes quite reasonable to begin to consider whether machine learning could provide a solution. There are many different methods which could be tried out, however reinforcement learning presents two distinct advantages. Firstly it does not need to be trained with data from good allocations in the past because reinforcement learning agents 'learn on the go' by trying to discover the optimal policy for their goal through interaction with their environment. This is important because the problem of verifying a given resource allocation is optimal is exactly as hard as finding an optimal allocation, so gathering training data with examples of optimal allocations is ruled out. Secondly, reinforcement learning is adaptable. Because it only ever aims to learn an optimal policy though interactions, a reinforcement learning agent is constantly gathering feedback on its actions (even after it has reached a point where it could be considered 'optimal') and constantly trying to refine its approach to be as perfect as possible. Therefore if the environment or the problem changes and the old policy is no longer the best one, the agent will notice this and adjust its approach and learn a new policy to achieve its goal. Within the context of scientific workflows this is desirable

because the tasks and the profile of the input data may change (over time or at once) and render the previous allocations of resources for the given task completely wrong. Indeed the entire system to which the workflow is deployed could change, for example migrating to new infrastructure or a different cloud computing provider. In all of these scenarios the way that resources are allocated to tasks would probably need to be adjusted and reinforcement learning is an approach which would be able to handle these changes and adapt to them.

1.2 Goal of the Thesis

The goal of this thesis is to use reinforcement learning to improve the efficiency of scientific workflows by more accurately assigning resources to the individual tasks within the workflows. Specifically the CPU and memory usage will be examined using two different approaches: Gradient Bandits and Q-Learning, and compared to the performance when using the task's default configuration.

1.3 Structure of the Thesis

The remainder of the thesis is structured as follows: in chapter 2 an overview and an explanation of the ideas and technologies relevant to this paper is given, and then in the related work chapter (chapter 3) some interesting papers are discussed which addressed similar topics. After that the approach to the problem is described in the approach chapter (chapter 4) and finally in chapter 5 the results are presented and analysed. The thesis concludes in chapter (TODO reference conclusion) with a summary and brief look towards other potential interesting areas of research.

Chapter 2

Background

This chapter provides background information on and explanations of the functionality and purpose of some of the technologies and concepts related to this thesis. To begin, the history of distributed computing and containerization are discussed in sections 2.1 and 2.2 before moving on to a discussion of their current usage in scientific workflows in 2.3. After that the workflow manager used in this thesis, nextflow, is introduced in section **??**. Finally, in 2.5 reinforcement learning is touched upon and its functionality as well as the benefits it could bring to the problem at hand are briefly discussed.

2.1 Distributed Systems, Batch Processing and Cloud Computing

Within the context of this thesis and scientific workflows it makes sense to discuss some of the ways in which tasks or jobs that have long run times and require large amounts of resources can be carried out. In general for a given user who needs to execute a workflow there are really only four options namely 1) running it on their personal computer, which

quickly becomes infeasible for large and difficult tasks, 2) using a cloud computing service, 3) using hardware with significantly more resources or 4) using a distributed system such as a cluster or grid. Of these four options number 1 and 3 obviously need no special explanation but cloud computing and distributed systems deserve to be briefly defined.

While there is a lot of debate to be had about what precisely constitutes a distributed system the article "Programming Languages for Distributed Systems" by Bal, Steiner and Tanenbaum generally defines a distributed system as one in which "multiple autonomous processors do not share memory but cooperate by sending messages over a communications network" [3] . Building on this definition, for this thesis the term distributed computing is considered to be the performance of a single computational task across multiple distinct machines. Usually this task is quite large or difficult, or is an aggregate of various smaller tasks but the core idea remains the same. Within the term 'distributed computational system' many distinctions can be made regarding the exact architecture of the distributed system but the two which are specifically relevant to this thesis are only grids and the types of clusters typically used at universities. One of the possible uses for distributed systems is for parallel high-performance applications because of the vast resources available in the system.

TODO: batch processing?

Unlike the definition of what constitutes a distributed system, a cloud computing service is quite easy to define and can be considered any service which provides on-demand access to a flexible amount of computing resources. These services must be paid for of course and are heavily reliant on virtualisation. Cloud computing provides a configurable amount of computing resources so that users only pay for what they need. A given user or group of users no longer need their own computing resources but can instead purchase them from cloud computing providers which relieves them of the burden of maintaining and managing such systems as well.

For scientific workflows both distributed systems and cloud computing services represent

excellent deployment architectures for increased performance due to the increased resources available in such systems.

2.2 Containerization

Virtualization has existed for a long time however the overhead of running a virtual machine is often not worth the advantages it provides. Particularly for reproducible software development the most important qualities of the host machine may not even be the architecture but rather the interactions with the operating system and its filesystem. Software containers are a type of virtualisation which do not virtualise the architecture of the host machine but instead use namespaces and control groups to virtualise the operating system, network, filesystem and all of the other peripheral components a program interacts with. This enables containers to isolate processes from the host machine and also from each other. Specifically for the deployment of applications and software this provides a humongous advantage. With containers it is possible, for example, to run two different versions of the same software on the same machine or even to run two versions of the same software but with two different configurations in parallel to each other. In addition to this the software will not know anything of the other version of itself running in parallel. Most importantly, in both of these cases the behaviour of the programs in the containers should always be the same. This is because a software container provides a sanitized version of the host system where there is no danger of other users or processes interacting with the filesystem or using the devices made available by the operating system.

Beyond the benefits for co-locating services on the same machine without minimal danger of interference, containerization also simplifies the process of deploying software to a new machine. If a given machine supports the running of containers then all that is needed to deploy one's software on that machine is an image or container of the software. One of the most popular containerization softwares is Docker. Their philosophy is "build once, deploy

anywhere", and many cloud computing services only need to be provided an image of a container and they can instantly deploy that service. This simplifies software deployment for the software developers as well as the management of the machines on which they run. In addition to this it also becomes easy to scale applications up or down. Adding more resources to a container or starting a new machine running the same application becomes a trivial process. Lastly, for managers of cloud computing centers the additional layer of virtualization offered by containers enables them to move containers and applications between different machines.

This is particularly useful for scientific workflows because by using containers the tasks that make up a workflow can all be reliably reproduced on various different execution platforms.

2.3 Scientific Workflows

Over the last two decades computation has emerged to become an integral part of scientific research [8],[5] and with the increased use of simulations and digital sensors the importance of digital data continues to rise[19]. This has led to unprecedented computing power requirements in the sciences. Indeed much of the effort of scientists is now invested in the analysing and processing of the data rather than the gathering of the data, and software costs have come to dominate capital expenditures for many large-scale experiments [13]. Some of the scientific fields which have particularly large computational needs are Biology, Astronomy, and Seismology [19].

To handle so much data it is usually processed in a sequence of small individual steps or tasks, often using command-line tools [32], in a sophisticated structure of dependencies and pipelines which can run sequentially or concurrently based on the dependencies [5],[11]. Since these tasks may work on the same data or a version of that data which has been processed by another task, there are of course temporal and logical dependencies between the tasks.

Managing these complex interdependent structures can be quite difficult and has given rise to the concept of workflows, which are meant help model the constituent tasks and their dependencies.

Workflows can be modelled in many different ways, ranging from simple scripting languages to graphs and mathematical models [26]. At their core, workflows consist of four simple components: the inputs, the tasks, the dependencies between tasks, and the outputs. This can easily be modelled as a graph with vertices to represent tasks and edges to represent dependencies. An example of this can be seen in 2.1.

2.4 Scientific Workflow Management Systems

The emergence of scientific workflows as a method for representing and managing these complex computations has lead to the emergence of scientific workflow mangement systems (SWMS) (or scientific workflow managers) as a means to manage the execution of scientific workflows and their constituent tasks. Today there a litany of SWMS's available, such as Nextflow [9], Kepler [1], Pegasus [7] and others.

As the resource requirements of scientific workflows can be immense [14], workflow managers must be able to interface with powerful execution platforms, which usually use resource managers such as Kubernetes [4], YARN [30], or other resource managers. A workflow manager will schedule a workflow's tasks with the resource manager (this will include assigning the task a fixed amount of resources), handle failures as configured by the user, manage the communication between tasks, and if the workflow's execution succeeds it will collect and store the results.

At this point it is important to clarify the distinct responsibilities of a workflow management system, an execution platform and a scientific workflow, with regards to the tasks. The user or creator of the workflow decides which tasks need to be executed and when they need to

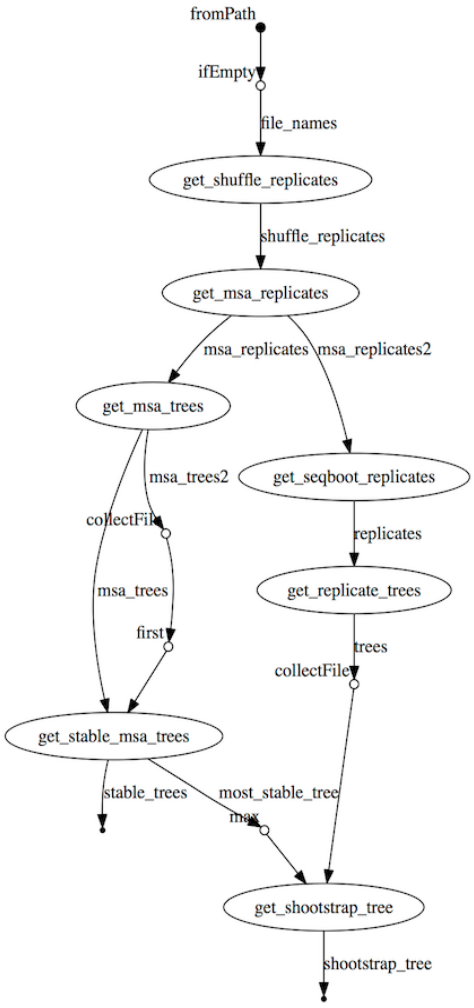


Figure 2.1: nf-core/shootstrap Workflow Represented as a Digraph [22]

be executed **relative to eachother** and the SWMS will schedule tasks with the execution platform according to this design. It is the execution platform which will ultimately decide when a task will **actually** be executed on physical hardware. In this transaction it is the SWMS which is acting as the middle man. Beyond which tasks to execute and in which order, the user will also decide what the inputs to a task should be and the workflow manager will ensure that before the task is executed those inputs are available and that they are passed to the task. Finally with regards to resources, the user does not need to specify what the resources assigned to a task should be, the user may do so if they like, however it is not a requirement. The assignment of resources to a task is a request by the workflow management system to the execution platform, and provided the platform can fulfil these requirements then the task should be executed in an environment where it has access to as many resources as it was assigned but not more. Tasks which exceed their resource requirements will often be aborted and this will be communicated to the resource manager which will then have to decided if it should try to run the task again, continue without the task, or abort the execution of the entire workflow- all depending on the configuration of the workflow.

2.5 Reinforcement Learning

Popularized in the seminal book by Sutton and Barto [27], reinforcement learning can broadly be said to present a framework for a learning system or "agent" to learn the optimal policy for achieving a given goal in an uncertain environment by interacting with the problem and the environment. Most importantly it is also able to adjust this policy "on the go", meaning it can both learn a new policy if the challenge or the environment changes and that it can be deployed immediately into any environment without any training and it will improve as it gains experience. In this context the agent's goal is always set by a reward function. Using reinforcement learning, the agent learns to maximize this function and thus, hopefully, achieve the desire of its designers. To put it simply, reinforcement learning is a

"computational approach to learning from interactions" [27]

Central to all reinforcement learning is the idea of exploration versus exploitation. Since the agent initially knows nothing about its environment it must attempt to learn through exploration. By trying different things and receiving different rewards the agent can construct a policy that always makes the optimal choice. But in order to know what a good or an optimal choice is the learning system must also occasionally make the wrong choice so that it can learn not to make it again. Trying different things is "exploration" and using the knowledge gained from this to make the right choice is"exploitation". An agent cannot simultaneously explore and exploit. This dichotomy is at the core of reinforcement learning. The agent must always make the choice between exploring more to potentially discover an even better policy and eventually yield even better rewards in the future, or using its current policy to increase its immediate rewards.

Beyond the dichotomy mentioned above, the other challenge presented to the designers of a reinforcement learning agent is the question of how to translate the overarching goals into a singular reward. A reinforcement learning agent run ad infinitum should converge to an optimal policy which maximises reward, and thus the onus is on the designer to pick a reward function which, when maximised, will help them to achieve their goals.

2.5.1 Actions, States, Policies and Rewards

For the purposes of this thesis only q-learning and gradient bandits will be discussed. In order to understand these approaches the concepts of states and actions must be introduced. An agent interacts with its environment by performing actions. These actions may have a tangible effect on the agent and the environment and the agent receives feedback about its actions. This feedback is given to the agent in the form of rewards. The agent also maintains an internal representation of itself (either independently or in relation to the environment), this representation is considered the agent's state. An agent may change states either as a

result of its own actions or as a result of changes in the external environment. A simple visual representation of these interactions can be seen in 2.2.

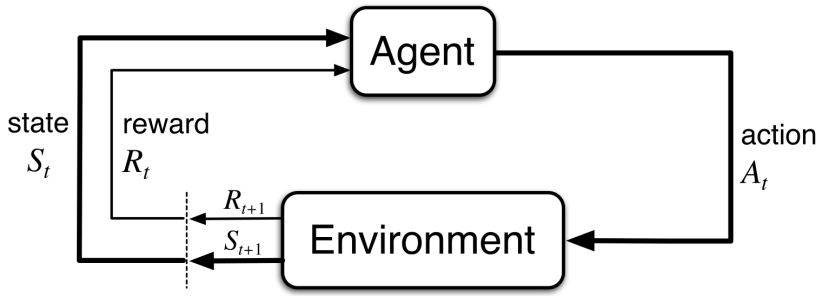


Figure 2.2: Agent Environment Interactions [27]

Policies and the question of how best to evaluate them are crucial to reinforcement learning. Within the context of states, actions and rewards, a policy is simply an instruction for which actions must be taken to maximise the cumulative reward over time. Which action to take may depend on the current state or the state which the action may cause the agent to transition to. In order to achieve its goal an agent needs to develop a policy that maximizes its reward, then as the agent encounters new situations it simply follows the policy it has learned. To evaluate a given policy it must be compared to the optimal policy. For any given problem and its reward function there exists at least one policy which maximizes the reward received and could be considered optimal. At its core, reinforcement learning aims to enable the agent to continuously refine its current policy so that it approaches the optimal policy.

2.5.2 Gradient Bandits

In this thesis two specific types of reinforcement learning are considered. First are gradient bandits. The "Bandit Problem" is a sequential learning problem, where each round the bandit has to decide which action to take by pulling one of N levers [10]. The analogy used by Sutton and Barto is of a room with several levers, and the question of which lever to pull (the pulling of a certain lever may also be called the action). From the bandit's perspective pulling any of the levers yields a certain reward and it is the bandit's task to

find a policy for pulling levers which yields the maximum reward. Approaches to solving the bandit problem present frameworks for solving problems in which an actor repeatedly returns to an unchanging situation where there are several choices, whose consequences are not known. Applying this to the case of scientific workflow managers and the sizing of tasks one can consider the levers, or the choice, as the resource configuration. The bandit is asked repeatedly to allocate a certain amount of resources for a task (equivalent to pulling one of the levers) and must find the best policy for maximising the reward (i.e. the best "lever" or resource allocation).

Gradient bandits solve the bandit problem by using the gradient of the reward function to learn a preference $H_t(a)$ for each of the available actions (or levers). This preference $H_t(a)$ is updated after each round t. Using gradient ascent the bandits take small steps in the direction of the ideal preference for each lever in order to maximize rewards. These steps are taken via stochastic gradient ascent, for which the formula is :

$$H_{t+1}(a) = H_t(a) + \alpha \frac{\delta E[R_t]}{\delta H_t(a)} \tag{2.1}$$

This formula aims to increment the preference for an action proportional to the increment's effect on performance. The preferences influence the bandit's probability of choosing a given action. Crucially, after choosing an action and receiving a reward, the preferences for all of the actions are updated to try and follow the gradient of the reward. In practice however the expected reward for each action is not known- if it were known the problem would be trivial and the agent could be configured to always pick the maximizing action. Instead the expected reward function and its gradient must be approximated over time. This leads to the formula for updating the preferences proposed by Sutton and Barto. For a preference $H_{t+1}(A)$ after taking action A at timepoint t (denoted by A_t) and receiving reward R_t , where $\hat{R}$ is the average of all the rewards so far and $\pi_t(a_t)$ is the probability of picking action a_t, its preferences are updated as follows:

$$H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \hat{R})(1 - \pi_t(A_t)) \qquad \text{and} \tag{2.2}$$

$$H_{t+1}(a) = H_t(a) - \alpha(R_t - \hat{R})\pi_t(a) \qquad \forall a \neq A_t \tag{2.3}$$

In this formula α is called the step size and it is a parameter which may be a constant or a function of the timepoint t but must be greater than 0. It has been proven in [27] that this formula eventually approximates the formula in 2.1 for gradient ascent.

After updating its preferences, the gradient bandit then uses these preferences to inform the probability of choosing the next action at time $t+1$. This can be done, for example, using a softmax distribution, and indeed in the rest of this thesis the softmax formula in 2.4 is what is used to update gradient bandits.

$$\pi_t(a) = \frac{e^{H_t(a)}}{\sum_{i=1}^{n} e^{H_t(i)}} \tag{2.4}$$

At the start (timepoint $t = 1$) the gradient bandit's preferences are all the same: $H_1(a_1) = 0, \forall a$. It then chooses actions based on the probabilities determined by the preferences for each action, and updates its preferences according to 2.2 based on the reward it receives, and uses 2.4 to update the probabilities, which in turn will determine the next action chosen. Through this method the gradient bandit will begin with a natural inclination to explore, and as it figures out which actions yield the greatest rewards, the probability of picking those actions will increase and the bandit will choose those actions more often, thus naturally striking a balance between exploitation and exploration.

It should be mentioned however that this balance is dependent on the step size α. For large values of α the bandit will spend less time exploring and more time exploiting. But despite this, if the nature of the rewards received for a given action begin to change, for example if an action goes from consistently receiving quite large rewards to consistently receiving poor

rewards, then the formula in 2.2 will naturally begin to decrease its preference for that action and the bandit will begin exploration again- regardless of step size. This is of course one of the strengths of reinforcement learning approaches, namely that they can adapt to changes in the environment or the reward function.

Finally, some terminology- when the preference for an action has become so large that the probability of picking this action approaches 1 the gradient bandit can be said to have "converged" to this action.

2.5.3 Q-learning

While the bandits in the previous section do not have a concept of state and must only learn about the actions available to them, stateful agents must maintain an internal state $s \in S$ and choose actions $a \in A$ which will yield new rewards and may cause them to transition to new states. The transition to a new state may also be independent of the action picked. The set of all possible states which the agent may conceivably find itself in and the set of available actions which the agent may potentially take are denoted as S and A respectively. Within this context of states and actions, a policy $\pi(s)$ must pick the agent's action a based on its current state s.

Q-learning was first proposed in [31] and it provides a simple way for an agent to act optimally in controlled Markovian domains [31]. An explanation of Markov domains or policy evaluation is beyond the scope of this thesis. Q-learning will therefore be explained within the context of the simplified agent-environment interface mentioned earlier (subsection 2.5.1 and figure 2.2).

Q-learning is an off-policy temporal difference control algorithm. An in depth explanation of temporal difference is also out of the scope of this thesis but a simplified explanation will come later in this section. Off-policy refers to the fact that q-learning is able to approximate the optimal action-value function independent of the policy being followed. The fundamental

formula underpinning q-learning called one step q-learning:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha \times (r + \gamma \times max_{\hat{a}}(Q_t(\hat{s},\hat{a})) - Q_t(s,a)) \tag{2.5}$$

In this formula the function $Q_t(s,a)$ is a state-action-value function. It reflects the value (which is based on both historic rewards up to timepoint t and the expected future rewards) of taking action a in state s. When an agent takes action a it will receive reward r and transition to state $\hat{s}$. After this the agent must update its Q_t function for s and a to Q_{t+1}. If the old value function for s and a was $Q_t(s,a)$ then the update to this function is the difference between the old valuation and the reward plus the discounted expected value of the new state. The expected value of the new state is $Q_t(\hat{s},\hat{a})$ where $\hat{a}$ is the action which maximises $Q_t(\hat{(s)},a)$ and $\gamma \in [0,1]$ is the discount.

$$(r + \gamma \times max_{\hat{a}}(Q_t(\hat{s},\hat{a})) - Q_t(s,a) \tag{2.6}$$

This is called the temporal difference. It reflects the difference between the current approximation of the value function ($Q_t(s,a)$) and the optimal value function ($r+\gamma \times Q_t(\hat{s},\hat{a})$). If the current approximation is too optimistic then 2.6 will be negative and on the other hand if Q_t is too pessimistic or too small then the temporal difference will be positive. Using the temporal difference, Q_t is then adjusted by the temporal difference multiplied with the step size α to yield Q_{t+1}. This is what equation 2.5 shows. All that is required for this iterative process to converge to the optimal value function is that all s,a pairs continue to be updated [27].

Using a value function built from equation 2.5 an agent can approximate an optimal policy by simply using a greedy approach and always picking the action a which maximises $Q(s,a)$, when it is in state s (and then updating its value function according to 2.5). By building on this the epsilon greedy q-learning algorithm can be constructed. The pseudocode for this algorithm can be seen below.

Initialise $Q(s, a) = 0$, $\forall(s, a)$
Initialise start state s
while *True* do
    if $random < \epsilon$ **then**
        choose action a randomly
    else
        choose action $a = max_a(Q(s, a))$
    end
    Take action a, receive reward r, transition to state $\hat{s}$
    Update Q: $Q_{new}(s, a) = Q(s, a) + \alpha \times (r + \gamma \times max_{\hat{a}}(Q(\hat{s}, \hat{a})) - Q(s, a))$
end

Algorithm 1: Epsilon Greedy Q-learning Pseudocode

This algorithm introduces random exploration with a probability ϵ and follows the q-learning algorithm the rest of the time. With sufficient exploration and an epsilon that eventually decreases to zero this algorithm should approximate an optimal value function and can then follow an optimal policy by greedily selecting the actions which maximise the Q function it learned. This is the algorithm which will be used by all q-learning agents implemented in this thesis. A final note- since the requirement for convergence to the optimal value function is that all pairs are visited, during early runs whenever the q-learning agent encounters a situation with an action it has never chosen before, it must choose this action. Only if all the available actions have been tried at least once can it use the greedy approach of selecting the maximising action.

Similar to subsection 2.5.2 the q-learning agent also has a step size parameter α and it also has to balance exploration against exploitation. However for the q-learning agent its exploration can be heavily influenced by the ϵ parameter whereas the gradient bandit's exploration is implicitly tied to its preferences.

Chapter 3

Related Work

This chapter discusses other works in related areas and considers the nuances of their approaches and how they differ from the one presented in this thesis. First there will be a discussion of select pieces of literature which use reinforcement learning approaches to address similar problems. Then two papers related specifically to resource management within the context of scientific workflows will be considered.

3.1 Using Reinforcement Learning to Schedule Tasks/Jobs in Clusters

In [28] the researchers use reinforcement learning to schedule data-parallel processing jobs in shared clusters. In particular they considered resource management systems such as YARN or Mesos. Their approach was to schedule the jobs based on co-location goodness, which is a metric for how well a job can share the resources of the machine on which it is scheduled with another job. This metric was based on CPU usage and disk and network Input/Output (I/O) since these two activities tend to be very compatible- while one job waits on the disk

or network another job can be free to use the CPU and vice versa. The paper considered three different scheduling approaches: first in first out (FIFO), a gradient bandit which starts new each time and a gradient bandit which is starts with the data from previous runs. In the end the best performing approach was the bandit which had the benefit of the data from previous runs. This was because that bandit skipped the exploration phase, during which bandits sacrifice performance for learning, by virtue of having the previous runs' data. This gradient bandit approach was then incorporated into three different schedulers called Mary, Hugo and Hugo*. Mary schedules based on co-location goodness, Hugo extends this by aggregating jobs into groups based on their similarities and scheduling the groups, and Hugo* does the same as Hugo but also considers how long a task has been waiting.

This paper does have a quite a few similarities with the approach used in this thesis. Notably they both use an approach based on Gradient Bandits and both use the data from previous runs. However the similarities end there, as the paper was concerned with scheduling within clusters based on co-location goodness whereas this thesis only looks at resource allocation for indin tasks are vidual tasks.

In the SCARL paper ([6]) researchers use reinforcement learning for scheduling jobs among machines with heterogeneous resources. Their approach (which is named SCARL) was based on combining attentive representation and the policy-gradient method. Attentive representation is a technique for focusing attention more quickly within a neural network. The model which they used to represent the problem used the allocation of jobs to machines as the state whilst the available actions at any given point were the scheduling of jobs. The reward function was based on a slowdown metric: $(elapsed\_time * penalty\_factor)/computation\_time$. Ultimately the researchers found that for high levels of heterogeneity SCARL outperformed the shortest job first (SJF) metric by 10% and for lower levels of heterogeneity it outperformed it by 4 %.

Once again the obvious difference between the paper and this thesis is that the paper concerns itself with scheduling as opposed to resource allocation, however there is also a large difference

in the methods used to approach the problem because of the use of a neural network, and the fact that SCARL attempts to consider the heterogeneity of the resources as well.

3.2 Using Reinforcement Learning to Allocate Resources

SmartYARN ([33]) applies a q-learning approach to balance resource usage optimization against application performance. This is one of the central considerations for any client of a cloud computing platform- the need to balance reducing costs by using less resources against the need to increase runtime by using more resources. In this paper the researchers used the performance of the application under a certain resource configuration as the state-space, while the action-space was made up of just 5 actions: increasing or decreasing one unit of CPU or memory or keeping the previous allocation. In the end the researchers found that the agent was able to achieve 98% of the optimal cost reduction and generally performed at the optimal level, finding the optimal allocation the vast majority of the time.

This paper served as the inspiration for the q-learning approach used in this paper and is also one of the few works discussed in this section which considers the central issue of the cost versus performance when assigning resources. One significant difference however is that SmartYARN only seeks to optimise individual jobs which do not depend on each other whereas for scientific workflows the jobs (or tasks) are part of a larger workflow and depend on each other and may need to wait for one another.

[25] tackled a similar problem- configuring the resource allocations of virtual machine's (VM's) using reinforcement learning. Their approach was a continuing discounted MDP with q-learning. The states were always a triple of CPU Time credits (used for scheduling cpu time), virtual cpu's (vCPU's), and memory. The agent's available actions were increasing, decreasing or leaving the allocations, with only one resource allowed to be changed per action. As a reward the ratio of the achieved throughput to a reference throughput was used. The key

trick used by the researchers was to use model based reinforcement learning. By modelling states the agent is able to simulate or predict the reward it can expect from previously unseen action-state pairs, whereas the classic q-learning approach (also used in this paper) requires the agent to experiment with each action-state pair. This means the model based agent is able to learn much faster and can enter its exploitation phase earlier than the static agent which must explore the action-state space for a long time, especially when there are large state spaces. Ultimately however the researchers found that the static agent performed quite well and achieved high levels of throughput in its own regard, although the model-based approach consistently outperformed the static one.

This paper's problem and its solution to it are again quite similar to those of this thesis but there is of course a significant difference since it schedules the resources of virtual machines as opposed to the resources of individual jobs. Another difference is of course the use of model based reinforcement learning to limit the time spent exploring the state action space for optimal rewards and spend more time exploiting this to achieve better results.

Finally there is the DeepRM [20] paper on resource management using deep reinforcement learning. Here the researchers used a policy gradient reinforcement learning algorithm combined with a neural network. The state-space consisted of the resources of the cluster and the resource requirements of arriving jobs, encoded as so called "images" which could be fed into the neural network. The actions available to the agent were simply to choose a job and allocate resources to it. In order to speed up the process the agent was given the option to schedule up to M jobs in a row, thus enabling it to leap forward by as many as M timesteps instead of always progressing by a single timestep. The reward given to the agent was the negative average slowdown. In the end the researchers' approach outperformed both the shortest job first metric and a very strong, heuristic based algorithm called Tetris [12]. A key reason that the agent was able to achieve this increased performance was that it learned a policy to maximize the number of small jobs it completed. While the approach taken in the paper was still better with large jobs it was significantly better with small jobs because

the network learned to always keep a small amount of resources available to quickly service any small jobs which might arrive, whereas the other approaches inevitably scheduled all the available resources so that small jobs also had to wait.

The similarities here are relatively few since this paper also had to concern itself with the decision of which job to allocate and when, whereas this thesis only considered how much of a given resource to allocate to a task. Beyond that there is also the fact that a neural network was employed and the decision to use the resources of the system to model the state of the agent. Though this is an idea which could be interesting as a topic of further research if one were to expand on this thesis. What is most interesting however is the policy that the neural network was able to learn. It was able to maximise throughput by always reserving a small amount of resources for small tasks which re-occur often. This kind of advanced strategy is something that could of course be built into an algorithm but the ability of a reinforcement learning agent to learn such a policy shows their potential for impressive results within the field of task scheduling.

3.3 Resource Allocations in Scientific Workflows

In [2] the researchers used cluster profiling to build node groups with similar performance profiles and map a task to a node group based on historical performance data. A 3 step system is used. In step 1 the cluster profiles are built based on microbenchmarks and hardware analysis tools. Then in step 2 the tasks are labelled in the SWMS based on their monitoring data. Finally, in the third step dynamic task-resource allocation is performed based on the information from the previous steps. Using this approach the authors were able to achieve a mean reduction of job runtimes of 19.8% compared to popular schedulers, and a reduction of 4.54% compared to the shortest-job-fastest-node (SJFN) heuristic.

This target of that paper is of course much more similar to that this thesis than any of the

previously mentioned papers however the obvious difference is that this thesis approaches task resource assignment with a reinforcement learning approach. One other similarity is that this paper used the exact same SWMS (nextflow) as this thesis. Beyond that it would be an interesting area of research to try and combine the topic of that paper with the topic of this thesis and try and apply reinforcement learning to task resource assignment amongst node groups with similar performance profiles.

Next, in [32] a feedback based resource allocation system for the scheduling of scientific workflows is designed. Their approach uses an online feedback-loop to improve resource usage predictions and thus more accurately allocate resources to the constituent tasks of a scientific workflow. Tasks within scientific workflows were assigned memory based on percentile or linear regression predictors. One such percentile predictor, called PC50, worked by using the median memory usage of a task as its prediction. This approach was significantly outperformed by the *LR mean* approach which was a linear regression predictor that predicted memory usage based on a linear regression that relates a task's input data to its memory usage. This prediction is then offset by the standard deviation between the predicted and the true peak memory usage.

This paper addresses a very similar problem to that of this thesis but attempts to use linear regression to predict memory usage whereas this paper is focused on using reinforcement learning. It also of course does not attempt to predict CPU usage or assigned CPUs to tasks. The PC50 approach and *LR mean* approaches ultimately served as inspiration for the naive approach to which the performance of the agents in this thesis are compared. The naive approach used here essentially takes the PC50 predictor and adds an offset based on standard deviation just like *LR mean*.

Finally there is [29] which presents a job sizing strategy for tasks in scientific workflows. The researchers used equations based on integrals over the resource usage to either minimise waste or maximise throughput. When gather training data they assigned tasks the maximum allowable value for all of their resources. This was repeated until the task's resource usage

statistics began to converge. Then they used their analytical solution to assign resources and if the task failed or was killed it was retried again with the maximum available resources. Ultimately an increase in throughput of between 10 and 400 was achieved.

This idea also served as inspiration for the naive approach used here, namely the idea to assign the maximum resources available and observe a task's performance is an interesting strategy for determining how to size a task. This of course assumes that a task's resource usage is independent of its available resources. It is also a notably different approach to exploring the available space of resource allocations from that of the reinforcement learning approaches used here. Lastly unlike in [32], [29] also looks at CPUs and how to assign this resource to tasks within scientific workflows.

3.4 Summary of Similarities and Points of Interest in these Works

All of the papers discussed in sections 3.1 and 3.2 managed to utilize reinforcement learning in an administrative fashion to improve the performance of jobs in their respective computing environments. A common theme among the papers concerned with resource utilization was to limit the action space to increasing, decreasing or keeping the current allocation or resources and storing the current allocation as the agent's state. The majority of the papers also utilized neural networks and it is also worth noting that most of them used runtime in their reward functions. A key difference between all of the above papers and this thesis is that these papers handled the direct allocation of resources among the machines in their environment to tasks or jobs, whereas in a scientific workflow management system the tasks are given a resource allocation by the SWMS and this is then passed on to a platform or cluster management software which handles the actual allocation of the job to a machine. Indeed it is possible for these two learning systems to interact with one another without knowing it.

In section 3.3 the topic of resource management in scientific workflows was looked at and the papers mentioned all tended to use more analytical approaches. Indeed there is relatively limited literature regarding combining machine learning and scientific workflow management. A consistent theme in the papers about sizing tasks in scientific workflows is the idea that a task's resource requirements are relatively static and can be learned over time using linear regression or profiling. One topic which is not really addressed in these papers is the relationship between resource assignment and performance, because assigning a task more resources (which may be less efficient) can still speed up performance. This is an important topic in this thesis and the papers [32] and [29] do not mention runtime, while [2] does not deal with task sizing. In this thesis the effect of task sizing on workflow performance is an delicate tradeoff which must be balanced correctly to both reduce resource wastage and maintain reasonable runtimes and improve (or at least not harm) workflow performance.

Chapter 4

Approach

This chapter covers how a solution was integrated into a scientific workflow manager, as well as the specific parameters and reward functions of the reinforcement learning agents. In section 4.1 the software architecture of the approach and its integration with the nextflow source code will be discussed, and then in 4.2 and 4.3 respectively the details of the gradient bandit and the q-learning agents will be considered.

4.1 Integrating a Solution into a Workflow Manager

Nextflow is a robust scientific workflow management system written primarily in groovy. It supports various execution platforms and has a large variety of tools to help users analyse the performance of their workflows [23]. Nextflow is free open source software and for this thesis a fork of the 20.12.0 version was used [1]. In order to integrate a reinforcement learning approach with the nextflow source code a simple structure was designed which externalised the storage of data from previous tasks and workflows so that when a task needed to be scheduled the reinforcement learning agent could retrieve historical data about the task from

[1] https://git.tu-berlin.de/zunkernicolas/nextflow-extension

a database to inform its decision. The structure can be seen in the following figure.

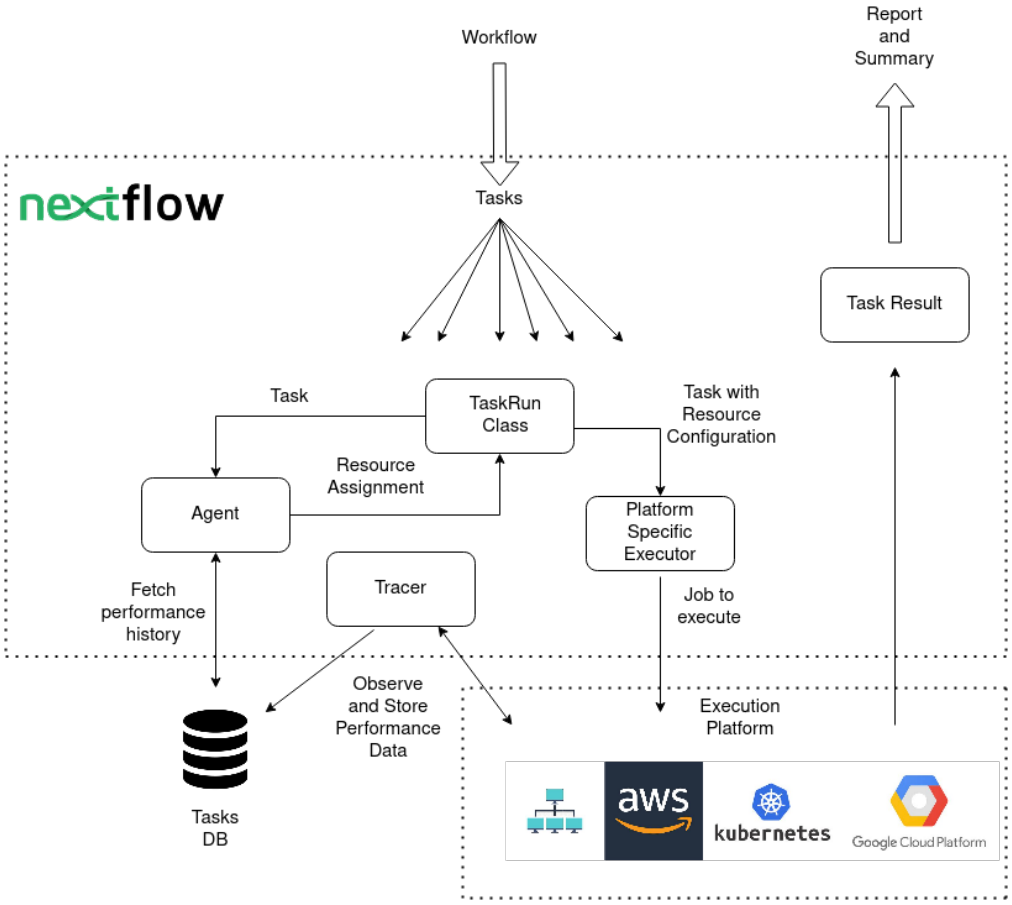


Figure 4.1: High Level Design: Integration of a Reinforcement-Learning Agent into Nextflow

Although the code for the reinforcement learning agents is logically a part of nextflow, when discussing the above design they will be referenced separately. The interaction between nextflow, the agents, and the external actors (the user, the database and the execution platform) is as follows. First, before a task is ready to be scheduled by it is first sent to a reinforcement-learning agent specific to that task, for the purposes of this thesis that was either a q-learning agent or a gradient bandit. The task's agent would then retrieve historical performance data and any other relevant data (i.e. what state the agent was last in) from the database. This database is external to nextflow and the creation and management of this database is not performed by nextflow or the agent- nextflow and the agent only use the database to save or read data between workflows. Using the data, the agent then selects a

new resource allocation for the task and overwrites the task's default configuration. After this, nextflow uses its custom executor for the given execution platform to prepare the task and pass it on to that platform. The task is then executed on the platform. It is important to note that the execution platform will also have its own system for managing, scheduling and executing jobs and processes. As the task is executed nextflow's tracer module will gather performance data, i.e. peak CPU usage, peak RSS and other such things [23]. Once the task is finished all of this data is stored in the database to be used by the agent the next time the task comes. It is important to note that there is one agent for each unique task. These agents are called whenever their task needs to be executed and they use the database to receive rewards for their previous actions.

Now that the structure of the agent's environment has been explained and the relationship between the collected data and assigning a task resources is clear, it is time to delve into the specific approaches tried.

4.2 Gradient Bandits

In the course of this thesis gradient bandit were used to allocate both CPUs and memory. These two approaches will be explained here.

4.2.1 CPU Bandit

Design

The first bandit or CPU Bandit, as it was nicknamed, had a very simple set of actions to choose from which were based on the number of CPUs available to the system. The bandit then learned how many CPUs to allocate to its task. Its reward was based on a very straightforward function:

$$reward = -t \times (1 + cpus - cpu\_usage/100) \tag{4.1}$$

In this equation *cpu_usage* is a value in percent of the number of single core CPUs used by the task. If a task were assigned 4 CPUs and only effectively used two of them then *cpu_usage* would be 200%. *cpus* is the number of CPUs assigned to the task and t is how long the task took to run. Put in other words this equation punishes the agent with a negative reward equal to the amount of time it ran plus the unused CPU time (time multiplied by the number of unused CPUs). The idea is to provide an incentive for the agent to try to minimise the amount of time where any of the allocated CPU's are not being used, and by punishing it with the amount of time that the task ran for the agent is also encouraged to try to reduce the amount of time taken for the task to complete. This also means that, should a task have 100% usage then its penalty is not 0 but is just the execution time. The reason for this is that if the agent has no concept of time it will always allocate the least amount of resources possible, since that is guaranteed to minimise the amount of resources wasted, and ultimately the tasks and their workflows would all be incredibly slow.

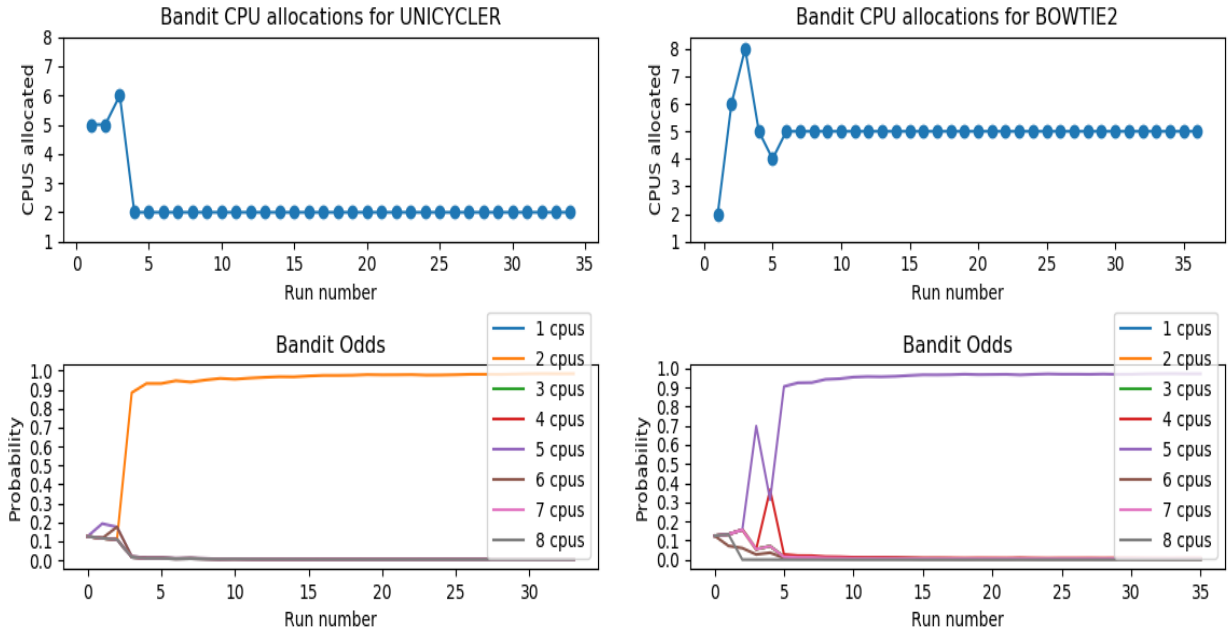
Picking the Right Step Size

The most pressing issue encountered with the CPU bandit approach was related to the step size. The initial step size α was 0.1, which would be an appropriate step size for a bounded reward function with a mean reward that is around 10 or is known to have a standard deviation less than that (these values are based on the example given in [27] where they use a step size of 0.1 for a reward function with a statistical mean of 4). However the reward function used in 4.1 has the bounds $[\ -time * (cpus + 1)\ ,\ -time\ ]$ and as such is unbounded, since *time* can be arbitrarily large, and in addition to this the standard deviation which was observed for the reward function was generally too large for this step size as well. Therefore whenever tasks had a runtime > 10 seconds, α was at risk of being too large.

The danger in using an inappropriate constant step size for a given reward function is that when the bandit receives a reward which is considerably larger than the previous average the "step" taken in the direction of that action will be too large and the bandits preferences will be updated such that the given action is always preferred and no other actions are tried any more. This causes the bandit to cease exploration too early. This effect was most commonly occurring for tasks which took longer than 20 seconds to complete and was especially pronounced for tasks with runtimes of 100 seconds and more. For these tasks the inherent volatility of the reward function (volatile relative to a step size of 0.1) meant that the bandits were converging very early and often had not explored the other actions at all. To provide a concrete example: if a bandit's third allocation of CPUs received a reward of -150 and the first two allocations had had an average reward of -200, then for a step size of 0.1 the preference for the new allocation would increase by approximately $50 * 0.1 = 5$. Now because of the exponential nature of the soft max distribution function used on the preferences, the probability of picking that action would increase by a large amount and then the bandit is in danger of converging to always pick that allocation without any more exploration. This is best demonstrated graphically, as can be seen in figures 4.2a and 4.2b.

These figures show two gradient bandits which were exhibiting the behaviour described above. Each figure is made up of 2 graphs which display the action chosen by the bandit in the top graph and the probabilities for all of the actions in the bottom graph. As can be seen in the two examples, after receiving a relatively large reward, the agent's probability of choosing a given action rises far too quickly and no other actions are attempted thereafter. This is occurring because the step taken by the bandit in the direction of a single action's preference is too large and the resulting probability of choosing that action grows so much no other actions are explored any more.

The solution to this problem is to adjust the step size so that it is not a constant value but is instead a function, which uses the average time that it usually takes the task to complete. This way tasks which have longer average runtimes are given smaller step sizes. The new



(a) Bandit 1

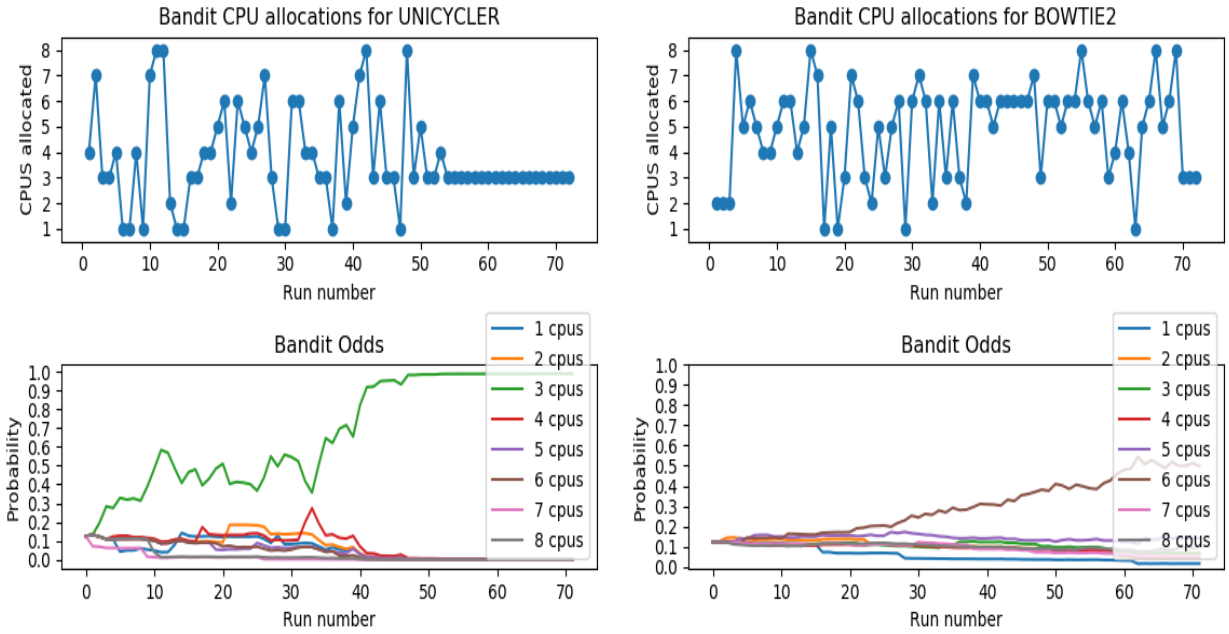
(b) Bandit 2

Figure 4.2: Example of Two Bandits Converging Too Fast

values for α is thus $1/avg(time)$.

The next two figures shows the same two tasks and their bandits but with the new step size. As can be seen the exploration phase is considerably longer and many different CPU allocations (actions) are tried before settling on one. Specifically in the case of the *UNICYCLER* task's bandit we can see that although allocating 3 CPUs seems to develop a strong preference early on (perhaps because it performed abnormally well once or perhaps because it genuinely is the best performing allocation) it does not become so preferred as to be dominant until much later and until that point the bandit continued to try other options.

It should be noted however that these changes mean that for optimal performance of the bandits there should already be some historical data about the task's and their runtimes available. It is not a necessity because over time the bandit can gather this data itself, but it is preferable.



(a) Bandit 1

(b) Bandit 2

Figure 4.3: Bandits with a Step Size Based on Their Historical Average Execution Time

4.2.2 Memory Bandit

The other gradient bandit approach used for assigning resources to tasks was called the memory bandit and its available actions were to memory assignments. Beyond the obvious, the memory bandit was different from the CPU in two respects. Firstly the amount of memory available is several order of magnitudes larger than the amount of cpus. Thus, in order to limit the number of available actions to a manageable amount, the original amount of memory assigned to the task m (in the default configuration) is split up into n chunks each of size $c = m/n$. The bandit is then presented a actions, each of which assigns $a * c$ amounts of memory to the task. a can be chosen as $1.5 * n$ to give a good balance between considering allocations that are larger or smaller than the default configuration, however since it is known in advance that user estimated are usually quite poor one could also choose $a < n$ to encourage memory efficiency. In addition to this by choosing larger or smaller n

values the size of the memory chunks explored by the bandit can be adjusted. Secondly, and this is the biggest difference between the two bandits, is that a task which is assigned too little memory and overuses it will be killed. This means that there must be a special penalty for such cases.

The reward function used for the memory bandit, after assigning $memory$ bytes of memory is as follows:

$$reward = \begin{cases} -2 * memory/c, & \text{if the task ran out of memory} \\ -1 * unused\_chunks & \text{otherwise} \end{cases}$$

Where $unused\_chunks = (memory - peak\_rss)/c$ and $peak\_rss$ is the peak resident set size (RSS), in bytes, of the task during its execution.

An obvious choice for this function may have been $reward = mem\_usage = peak\_rss/mem$ however during testing this was found to be too small for tasks which have a low memory footprint. For example a task which uses 1% of the assigned memory when assigned n chunks, and thus receives a reward of 1%, will only yield a reward of $1/n$% memory for the optimal allocation of just one memory chunks and for such small differences a gradient bandit will struggle to recognize the superior choice, even though it should be obvious. This is especially damaging to performance because these low usage tasks are precisely the ones where memory efficiency can be optimized the most and as such the reward function was changed to one which penalises the agent for the amount of unused memory. Of course to avoid the problem the CPU bandit had with its step size, the amount of unused memory in bytes could not be used and therefore the number of unused memory chunks was used instead. To return to the example of a task with 1% usage that task's rewards would now be $-n$ at worst and -1 at best, which makes it much easier for the bandit to determine good allocations. Finally in the case where a task has been killed the penalty is equal to double the number of memory chunks assigned because the agent must (at the very least) assign the same amount of memory again

or must assign even more memory.

The actual implementation of the memory bandit is slightly adjusted so that whenever a task fails with one memory allocation and is retried with a new one, the new allocation must be greater than the previous one. If the bandit does pick a smaller allocation then it is ignored and the new allocation is either doubled or the default configuration for that task is used (if double the new allocation is also less than that in the failed attempt). This prevents the workflow from being broken off if a given task fails too many times due to repeated poor allocations.

Finally, to return to a discussion of step size similar to that in 4.2.1, the memory bandit can be assigned a constant step size α equal to $1/n$. This is is because for n memory chunks, the bandits reward function is bounded by $[-n, 0]$.

4.3 Q-learning Agents

Similar to the approach with the gradient bandits, there was one q-learning agent per unique task. The q-learning agent's state was always the current allocation of cpu and memory for the given task, and as its set of actions it could choose between incrementing or decrementing the amount of cpus or memory, or it could do nothing. To limit the state space the maximum and minimum number of cpus and memory as well as the amount by which they were incremented or decremented was based on the default allocation given to the task by the developers of the workflow (provided they did not exceed the resources of the execution platform). When the task was scheduled by the agent for the first time it would start in the default state. Just like the memory bandit in section 4.2.2 the possible memory states of the reinforcement learning agent are also chunks.

For the q-learning agent a different reward function was used than for the CPU bandit, primarily because it also had to incorporate memory but also as part of an attempt to

experiment with different reward functions and approaches.

$$r = -max(0.1, unused\_cpus) \times (t/avg\_t) \times (mem \times (1 - max(0.75, mem\_use))) \quad (4.2)$$

Here the *mem* variable refers to the memory allocated to the process and *mem_use* is the value of of the peak RSS of the task divided by the memory assigned to it. *avg_t* is a constant value which is determined at runtime based on the historical average execution time for the task. Lastly, $unused\_cpus = cpus - cpu\_usage/100$ where *cpus* and *cpu_usage* are the same as in 4.1.

Since division by the task's average time is incorporated into the reward function it did not need to be incorporated into the step size as with the gradient bandit (see 4.2.1) and the issues associated with that were avoided. This function is effectively a product of the number of unused CPUs, the slow-down factor and the amount of unused memory. However there are some slight modifications. The *max* function is used to set an artificial floor for the penalty incurred by the unused CPUs and unused memory. Tasks which use more than 90% of the available CPUs are given the same reward as tasks which use exactly 90% in an attempt to prevent the agent from deciding to assign each task 1 CPU in order to minimise the number of unused CPUs. Additionally the floor for unused memory is capped at 25% in a similar manner. This is done to discourage the agent from allocating too little memory because tasks which use too much memory will of course be killed and have to start over, which has a hugely detrimental effect on performance. This reward function is of course negated in order to turn it into a penalty function so that the agent will seek to minimise its penalty by minimising the number of unused CPUs, the slowdown and the amount of unused memory.

The q-learning agent also has 3 other parameters- the step size α, epsilon ϵ and the discount γ. Step size was set to 0.1 and the discount was 1.0. Epsilon is adjusted over time- at first it is 0.5, to encourage exploration, then after 50 runs it is decreased to 0.25 and after 100

runs it is 0.1 to discourage exploration but leave room for the bandit to still occasionally try other actions and visit different states.

Exploring the State-Action Space

It should be noted here that unlike the gradient bandits from before, which had to decide from c or n number of actions for the CPU and memory bandits respectively, the q-learning agent must learn the value function for $(c-2)*((m-2)*a+2*(a-1))+2*((m-2)*(a-1)+2*(a-2))$ state-action combinations. Here c is the number of possible CPU states, m is the number of possible memory states and $a = 5$ is the number of possible actions. Now since the q-learning algorithm from 1 needs to try each action in every state combination at least once, and explores with probability ϵ then for arbitrary examples of $c = 4$ and $m = 3$ it will take, at the very least, 46 different runs with the same task to have tried every action in every state. This shows how much more complex the q-learning approach is than that with gradient bandits.

Q-learning agent for CPU assignments only

To have a q-learning approach which can be directly compared with the CPU bandit from section 4.2.1 a second q-learning agent is derived from the one in 4.3, which does not consider memory at all. For this agent its state is the current allocation of CPUs and its actions are a choice of either increasing,decreasing or leaving the current CPU allocation. This agent's reward function is effectively the same as in 4.2 but of course does not consider memory:

$$r = -max(0.1, unused\_cpus) \times (t/avg\_t) \tag{4.3}$$

The agent has a smaller state-action space to explore namely $(c-2)*a+(a-1)*2$ (or $c*a-2$) for $a = 3$. While this is more reasonable it is still close to three times as large as

the action space that the CPU bandit has to explore which is simply c. Ultimately however the theoretical benefit posed by q-learning is that when the agent learns, for example, not to increment the amount of CPUs when in state s (where s is the number of CPUs currently allocated) the agent is thus also learning to avoid all states/allocations $\{s_c \in S \mid s_c > s\}$ which allocate more CPUs than s. While this seems trivial, the gradient bandit is unable to see such connections and cannot tell that a task which has low CPU usage for s CPUs can be expected to have even lower usage for $s_c > s$ CPUs.

Chapter 5

Evaluation

This chapter looks at the results achieved and considers their origins as well as their implications going forwards. In section 5.2 the results are presented and analysed. Then section **??** considers what could be improved or done differently . Finally section 5.4 looks forwards and asks what implications this could have and what the next steps would be.

5.1 Testing the Agents

In order to evaluate the performance of these agents they were tested on a 32-core, 125GB RAM machine against 5 different workflows. Each workflow is run 10 times without any reinforcement learning agents active. This was done to have something to compare the results to later but also so that there was historical data about the average time each of the tasks take since the reinforcement learning agents need this information. After that the agents are tested. The gradient bandits are run 50 times for each workflow and the Q-learning agents 100 times.

The input data used for the workflows is based on custom combinations of different input files

provided by the workflow's maintainers- there are the full example datasets and the short example datasets. The combinations are designed to strike a balance between having large input data and realistic task configurations but also being short enough that it is feasible to run the workflows hundreds of times. For reference, the full datasets will often take 6 hours or more to run once whereas the short examples never take more than 5 minutes.

The nextflow source code inherently keeps track of the total number of resources available and the resources already assigned to tasks and it will not schedule more resources than the system has. The tasks are all run inside of docker containers with exactly number of CPUs and memory requested by the task.

For the final comparison in the evaluation chapter the 10 runs without the agents were compared to the last 10 runs with each of the agents.

5.2 Analysis of Results

5.2.1 Performance Comparison

Here the performance of the gradient bandit and the q-learning approach over their last 10 instances will be compared with the performance of 10 instances of the same sets of workflows when run with their default configuration.

The figures 5.1 and 5.2 show the performance of the different approaches and compare them to the default configuration. The left graph in the first figure looks at the CPU usage on a task by task basis whereas the left graph in the second figure is displays the total amount of CPU hours requested by by all of the tasks across all of the workflows and the amount of CPU hours that were actually used. Similarly, the right graph in figure 5.1 is the average time the tasks took whereas the right graph of figure 5.2 shows the total time it took for all of the workflows to complete. Finally in 5.3, the memory usage is shown. The unit measured

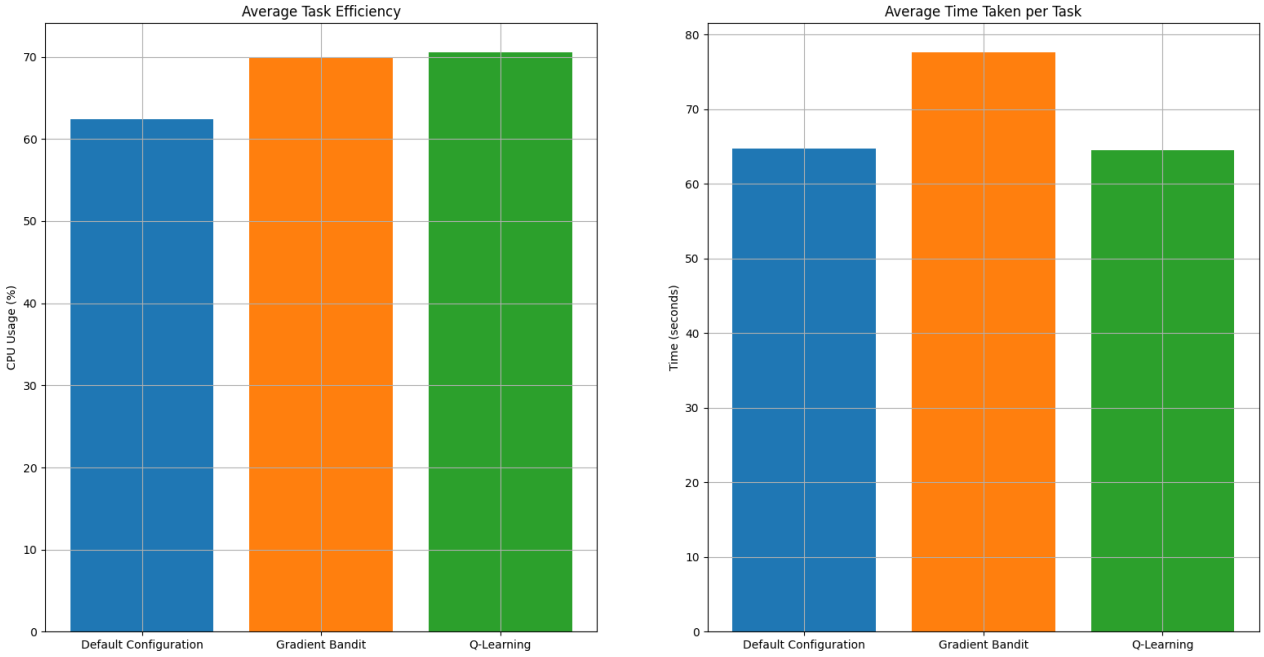


Figure 5.1: Comparison of Average Task Efficiencies and Runtimes

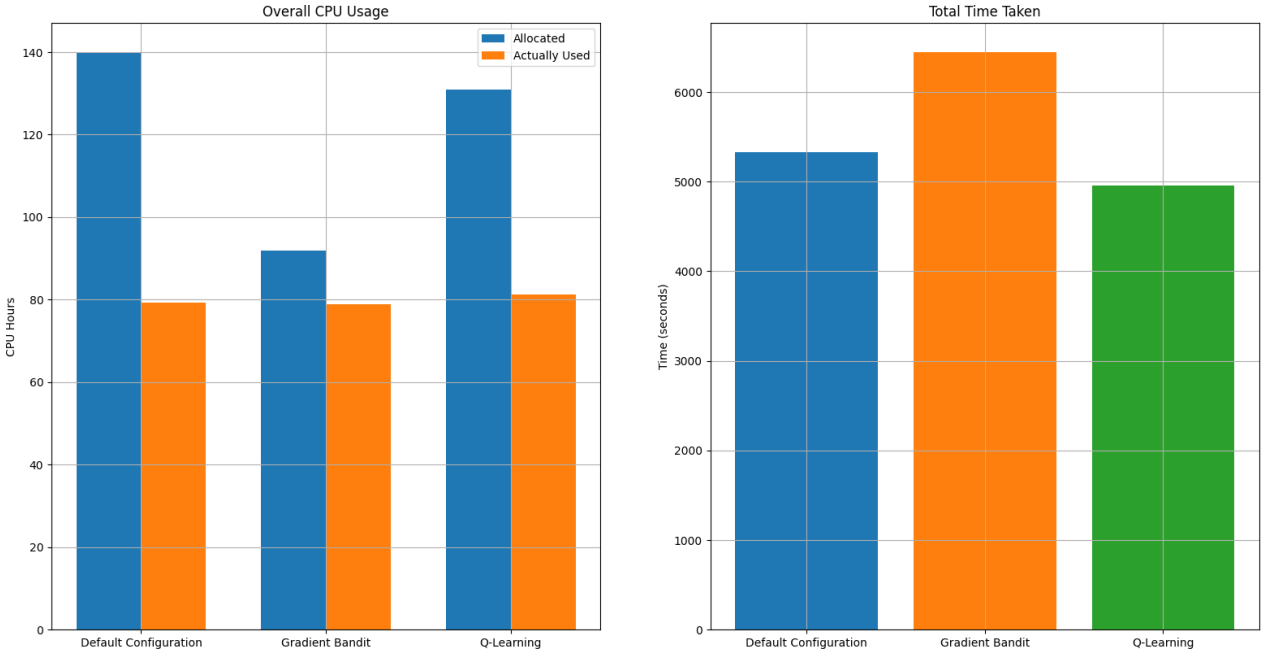


Figure 5.2: Average Performance of the Workflows

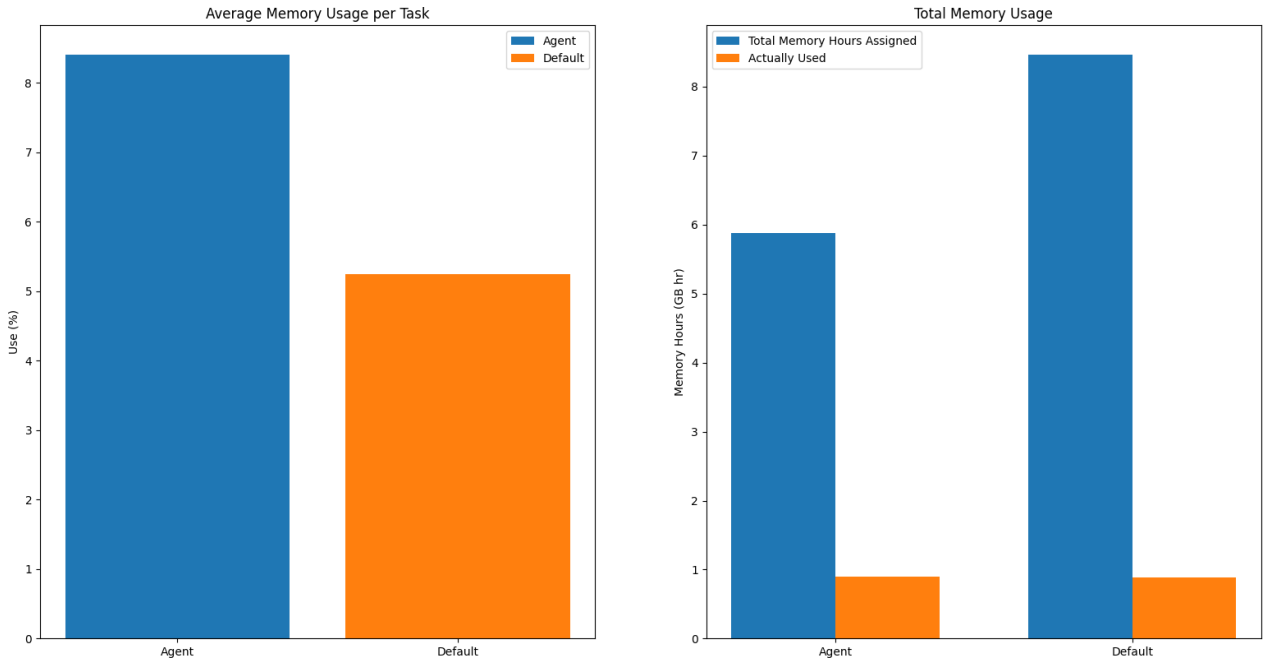


Figure 5.3: Memory Usage

is the gigabyte hour and the graphic displays the total amount of gigabyte hours allocated across all of the runs as well as the actual amount of gigabyte hours used (based on the task's peak RSS and execution time). The gradient bandit is not shown in this figure because it did not alter memory configurations.

From these graphics it is clear to see that both of the reinforcement learning approaches managed to improve the CPU efficiency of the individual tasks as well as allocating less CPU Hours across the entire workflow. In the case of the gradient bandit it was particularly effective at keeping the overall CPU hours assigned to an absolute minimum however the actual efficiency of this approach on a task by task level was not significantly better than the q-learning agent. However it seems that the gradient bandit's efficiency came at the cost of speed and it performed worse in this aspect. The q-learning approach was less efficient overall than the gradient bandit but it had the highest average efficiency per task and took the least time. Indeed the q-learning approach was both more efficient and faster than the default configurations and it used a great deal less memory. Memory is of the course last metric by

which the q-learning agent could be judged and in the end it proved to be significantly better with regards to this resource.

5.2.2 Explanation

One fact which may immediately jump out is that while the q-learning agent is slightly more efficient on a task by task basis than the gradient bandit, it is significantly less efficient from the perspective of CPU hours. This must mean that the gradient bandit performs better for longer tasks (and thus uses less CPU hours overall) but for most tasks the bandit and the agent behave similarly. This might also explain why the q-learning agent is faster- if it were choosing to assign more resources to the longest tasks it would be quicker but less efficient overall. Another possible explanation for the improved performance of the q-learning agent is that because it also reduces the overall memory usage it may be able to achieve a higher throughput. Recall that nextflow has a built in scheduler which never assigns more resources than the system has available. If there had been high memory tasks then these tasks may have been preventing other tasks from running and therefore by lowering the overall memory usage (and the overall number of CPU hours used) the q-learning agent may have increased the number of tasks running in parallel and thus sped up the workflow.

5.2.3 Summary of Results

In conclusion both of the reinforcement learning approaches were able to intelligently size the tasks which comprised the workflows and in the end the overall number of CPU hours required was less and indeed the percentage of those CPU hours which were wasted was reduced. In addition to this the q-learning agent was also able to significantly reduce the memory usage. The gradient bandit took slightly longer on average but was much more efficient for that whilst the q-learning agent struck a better balance between resource efficiency and increased performance and was ultimately able to outperform the default configurations with regards

to both efficiency and time.

While it could be argued that the q-learning agent is therefore the "better" option it is hard to say exactly what the right approach is because it also depends on the desires of the user. Many scientists may be prepared to accept slightly longer run times if the "niceness" of the workflows is increased and they are able to free up resources for others to use at the same time or reduce their own costs (if they are paying for CPU hours). Indeed from a financial perspective the gradient bandit wastes significantly less CPU hours than the extra time it takes to complete. However in a situation where the user is paying for the entire system it would make more sense from a financial perspective to use the q-learning agent approach since it reduces the amount of time for which the system needs to be up.

5.3 Potential Improvements

Immediately the most obvious area in which these two approaches could be improved upon would be to incorporate read and write calls into the reward function. When a thread reads or writes characters from or to a device it must perform a system call and wait for it to complete and during this time the thread cannot use the CPU. If a program has a high degree of parallel operations but also has to read and write lots of characters then it would probably have lower CPU usage (because of the many system calls) but simultaneously would benefit from being assigned more CPUs (because it is parallelizable). However because of the nature of the bandit and the agent's reward functions it is unlikely that this task would be assigned more resources simply because the number of read and write calls are not considered.

One aspect which weakened the gradient bandit is the disconnected nature of its model of the available actions. It cannot understand that actions which assign more resources are all quite similar. From the perspective of the gradient bandit all of its actions have the same probability of success and it can't tell when actions are very similar. As an example the

decisions to assign 5 CPUs or 6 CPUs to a task which has been exhibiting low CPU usage are considered two completely independent decisions even though they both fundamentally assign more resources and indeed if the assignment of 5 CPUs is not particularly successful then 6 CPUs can be expected to be even worse. The q-learning agent, on the other hand, while it cannot actually tell if certain actions are similar, can only increase or decrease the resources one step at a time and thus is unable to land in states of extreme excess or insufficiency. Returning to the previous example - if assigning 5 CPUs was unsuccessful then the agent is unlikely to return to that state and thus will not even be able to consider the question of whether or not to assign 6 CPUs.

A different issue which was relevant to both approaches was the length of time needed to train the agents. The gradient bandit required 40 runs and the q-learning agent took 90 runs before they were tested and for large scale workflows this is a considerable amount of time to spend. Of course the agents are able to learn on the go and the performance during the training time is not so much worse as to make the workflows unusable during that time, so it really depends on the user and how often they use a given workflow.

Finally by training the agents using the same inputs there is a danger of overfitting the agents' resource assignments. This may be less of a danger for the CPU assignments, because the character of a task's CPU usage is less dependent on the input and mores so on the task itself- but repeatedly using the same inputs for an agent which sizes tasks for memory does present a real danger of overfitting. This is because the memory footprint of a task is much more dependent on the input to the task and could indeed mean that the agents might learn a less performant memory allocation.

5.4 Implications and Further Areas of Research

The obvious implications of these findings are that much like some of the scheduling problems mentioned in chapter 2, this is also an area in which reinforcement learning has interesting potential. Additionally it has also gone to show that the resource configurations in scientific workflows have room for improvement and are another area which could be researched further. Nextflow and other scientific workflow managers represent a new class of actor in the interactions between user, application and execution platform. These workflow managers do no manage the execution platform and they do not create the workflows which are executed. They are a kind of middle man and thus present a unique opportunity. Normally the interactions in such an environment are between the user and the execution platform. The user has control over both the requests for resources and the tasks while the execution platform has control over the execution at a granular level. However in the context of a scientific workflow using a workflow manager, the manager sits in between the user and the execution platform and has no control over the tasks that must be executed nor over the execution itself (this is configured by the user and managed by the execution platform). It is only capable of scheduling and sizing the tasks. This position requires a slightly different approach to common problems such as scheduling and resource assignment but it also presents an interesting avenue for research and novel approaches to these familiar problems.

Following along this train of thought, an interesting area for further research would be incorporating a machine learning approach into the scientific workflows internal scheduler. Whilst the reinforcement learning agents used for each individual task had no knowledge of the other tasks and dependencies in the workflow a scheduler would be able to have the full picture of all workflows digraph and could perhaps improve performance to a greater degree.

The last area of research which could prove interesting would go in the opposite direction and would be to incorporate simpler, unintelligent approaches into the sizing of tasks. For example the memory assigned to a task could be picked based on the average of the peak

RSS of the task during its previous runs. Likewise, the number of CPUs assigned could be based on a linear regression which tests various different CPU assignments and then uses a line of best fit to model a function of CPUs and rewards and then attempts to pick the minimizing or maximizing parameter.

Lastly, improvement on the approach presented here using either a different reward function, a custom policy combined an on-policy reinforcement learning algorithm such as TD-learning, deep q-learning which could use a neural network to model the reward function and thus reduce exploration time, or an approach which uses fractional CPUs to containers (since containers can be assigned e.g. 1.5 CPUs) for fine-grained resource assignment, all could prove to be very interesting for further research.

Appendix A

Technologies and Concepts of a Larger Context

This appendix chapter contains definitions for technologies and concepts that are mentioned in the thesis, but are not of higher importance for it. Section A.1 introduces a consensus algorithm for distributed systems.

A.1 Finding Consensus in a Distributed System

The Paxos algorithm has first been described by Leslie Lamport in his work about the parliament on the island of Paxos [17]. The parliament's part-time legislators had been able to maintain consistent copies of their records by following the algorithm protocol. While the original work's main focus has not been the application in computer science, the author simplified his explanations later in [16] and described how Paxos can be used to find consensus in fault-tolerant distributed systems.

Even though the Paxos algorithm cannot mitigate Byzantine failures (see appendix C), it

can mitigate the effects of different processing speeds of participants and reordered, delayed, or lost messages. The only requirement is that at least half of the participants can somehow communicate.

In Paxos, four different roles exist. The **client** issues requests and waits for responses. Based on the client's request, multiple **proposer** propose a value. All these proposals are processed by the **acceptors** which will agree upon one at the end. Finally, there are the **learners** which guarantee persistence of agreed decisions and respond to client reads. The usual setup used in practice, in which every machine participating fulfills each of the roles besides the client role, is shown in figure A.1.

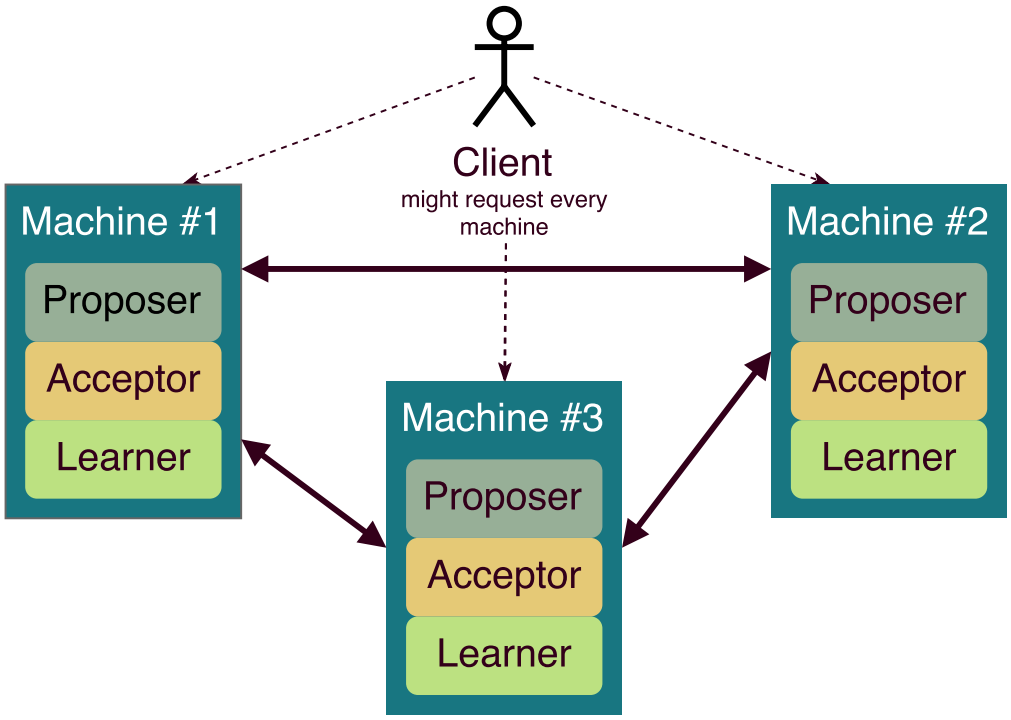


Figure A.1: Example Setup of Three Machines Agreeing on Values Based on the Paxos Algorithm

A.2 Detecting Mutual Inconsistency

Parker et al. claim that a system must ensure the mutual consistency of data copies by applying all changes made to one copy to all others correspondingly [24]. Each time two copies of the same original data item have a different set of modification applied to them, they become incompatible and this conflict must be detected. However, this is not trivial, because "network partitioning can completely destroy mutual consistency in the worst case". Nevertheless, Parker et al. state that the efficient detection of conflicts that lead to mutual inconsistency can be done by a concept they call *version vectors*. They define a version vector for an item f as "a sequence of n pairs, where n is the number of sites at which f is stored. The ith pair (S_i: v_i) gives the index of the latest version of f made at site S_i". An example vector for an item stored at the sites A, B and D is <A:2, B:4, D:3>, which translates in a file that has been modified twice on site A, four times on site B, and thrice on site D.

A set of vectors is "compatible when one vector is at least as large as any other vector in every site component for which they have entries". Otherwise the vectors conflict and are incompatible. E.g., the two vectors <A:2, B:4, D:3> and <A:4, B:5, D:3> are compatible because the second one dominates the other one. <A:3, B:4, D:3> and <A:2, B:5, D:3> conflict, because the first one indicates that the data item was modified one more time on node A, while the second one indicates one more modification occurred on node B. However, if we add a third vector <A:3, B:5, D:4>, no conflict exists anymore, because it dominates the two others. The consequences of an operation performed on a data item for a data item's vector is depicted in table A.1.

By using version vectors, one can detect version conflicts and initiate (automatic) reconciliation. However, Parker et al. warn that two identical updates made on separate partitions will result in a conflict, even though none is present. Thus, they recommend to additionally check two data items on differences before a conflict is raised in certain applications. Furthermore, the reconciliation is most times not trivial, that's why tools such as Cassandra delegate this task

Operation related to data item	Consequence for vector of data item
Update on site S_i	Increment v_i by one
Delete or rename on site S_i	Keep vector and increment v_i by one, remove data item value
Reconcile version conflict	Set each v_i to maximum v_i from all vectors used for reconciliation. In addition, increment v_i of site that initiated reconciliation by one
Copy to new site	Augment vector to include new site

Table A.1: Influence of Operations on a Data Item's Version Vector

to the application layer [15].

Appendix B

Acronyms

AWS Amazon Web Services

IoT Internet of Things

Appendix C

Lexicon

Byzantine failure A malfunction of a component that leads to the distribution of wrong/conflicting information to other parts of the system is called Byzantine failure [18]. The name is based on the Byzantines Generals Problem, in which three Byzantine generals need to agree on a battle plan while one or more of them might be a traitor trying to confuse the others.

Appendix D

Listings

This is the appendix for code, that does not need to be provided directly inside the thesis.

D.1 Configuration for Node A

Bibliography

[1] Ilkay Altintas et al. "S04—introduction to scientific workflow management and the Kepler System". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing - SC '06* (2006). DOI: 10.1145/1188455.1188669.

[2] Jonathan Bader et al. "Tarema: Adaptive Resource Allocation for Scalable Scientific Workflows in Heterogeneous Clusters". In: *arXiv preprint arXiv:2111.05167* (2021).

[3] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. "Programming languages for distributed computing systems". In: *ACM Computing Surveys* (1989), pp. 32–2.

[4] Brendan Burns et al. "Borg, Omega, and kubernetes". In: *Communications of the ACM* 59.5 (2016), 50–57. DOI: 10.1145/2890784.

[5] Marc Bux and Ulf Leser. *Parallelization in Scientific Workflow Management Systems*. 2013. arXiv: 1303.7195 [cs.DC].

[6] Mukoe Cheong et al. "SCARL: attentive reinforcement learning-based scheduling in a multi-resource heterogeneous cluster". In: *IEEE Access* 7 (2019), pp. 153432–153444.

[7] Ewa Deelman et al. "Pegasus: A framework for mapping complex scientific workflows onto Distributed Systems". In: *Scientific Programming* 13.3 (2005), 219–237. DOI: 10.1155/2005/128026.

[8] Ewa Deelman et al. "Workflows and e-science: An overview of workflow system features and capabilities". In: *Future Generation Computer Systems* 25.5 (2009), 528–540. DOI: 10.1016/j.future.2008.06.012.

[9] Paolo Di Tommaso et al. "Nextflow enables reproducible computational workflows". In: *Nature Biotechnology* 35.4 (2017), 316–319. DOI: 10.1038/nbt.3820.

[10] Qin Ding, Cho-Jui Hsieh, and James Sharpnack. "An efficient algorithm for generalized linear bandit: Online stochastic gradient descent and thompson sampling". In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2021, pp. 1585–1593.

[11] Yolanda Gil et al. "Examining the challenges of scientific workflows". In: *Computer* 40.12 (2007), 24–32. DOI: 10.1109/mc.2007.421.

[12] Robert Grandl et al. "Multi-resource packing for cluster schedulers". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2014), pp. 455–466.

[13] Jim Gray. *Jim Gray on eScience: A Transformed Scientific Method*. 2007. URL: http://languagelog.ldc.upenn.edu/myl/JimGrayOnE-Science.pdf.

[14] Gideon Juve and Ewa Deelman. "Resource Provisioning Options for Large-Scale Scientific Workflows". In: *2008 IEEE Fourth International Conference on eScience*. 2008, pp. 608–613. DOI: 10.1109/eScience.2008.160.

[15] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[16] Leslie Lamport. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[17] Leslie Lamport. "The part-time parliament". In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169.

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401.

[19] Chee Sun Liew et al. "Scientific Workflows: Moving Across Paradigms". In: *ACM Computing Surveys* 49.4 (Dec. 2016). ISSN: 0360-0300. DOI: 10.1145/3012429. URL: https://doi.org/10.1145/3012429.

[20] Hongzi Mao et al. "Resource management with deep reinforcement learning". In: *Proceedings of the 15th ACM workshop on hot topics in networks*. 2016, pp. 50–56.

[21] A.W. Mu'alem and D.G. Feitelson. "Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling". In: *IEEE Transactions on Parallel and Distributed Systems* 12.6 (2001), pp. 529–543. DOI: 10.1109/71.932708.

[22] Nextflow. *Shootstrap Pipeline DAG*. Nextflow, 2021. URL: https://www.nextflow.io/docs/latest/_images/dag.png.

[23] nextflow.io, ed. *Tracing and Visualisation - Nextflow*. 2021. URL: https://www.nextflow.io/docs/latest/tracing.html#trace-report.

[24] D. Stott Parker et al. "Detection of Mutual Inconsistency in Distributed Systems". In: *IEEE Transactions on Software Engineering* SE-9.3 (1983), pp. 240–247.

[25] Jia Rao et al. "Vconf: a reinforcement learning approach to virtual machines auto-configuration". In: *Proceedings of the 6th international conference on Autonomic computing*. 2009, pp. 137–146.

[26] Matthew Shields. *Control- Versus Data-Driven Workflows*. Ed. by Ian J. Taylor et al. London: Springer London, 2007, pp. 167–173. ISBN: 978-1-84628-757-2. DOI: 10.1007/978-1-84628-757-2_11. URL: https://doi.org/10.1007/978-1-84628-757-2_11.

[27] Richard S. Sutton, Francis Bach, and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT Press Ltd, 2018.

[28] Lauritz Thamsen et al. "Mary, Hugo, and Hugo*: Learning to schedule distributed data-parallel processing jobs on shared clusters". In: *Concurrency and Computation: Practice and Experience* 33.18 (2021), e5823.

[29] Benjamin Tovar et al. "A job sizing strategy for high-throughput scientific workflows". In: *IEEE Transactions on Parallel and Distributed Systems* 29.2 (2017), pp. 240–253.

[30] Vinod Kumar Vavilapalli et al. “Apache Hadoop Yarn”. In: *Proceedings of the 4th annual Symposium on Cloud Computing* (2013). DOI: 10.1145/2523616.2523633.

[31] Christopher J. Watkins and Peter Dayan. “Q-learning”. In: *Machine Learning* 8.3-4 (1992), 279–292. DOI: 10.1007/bf00992698.

[32] Carl Witt, Dennis Wagner, and Ulf Leser. “Feedback-Based Resource Allocation for Batch Scheduling of Scientific Workflows”. In: *2019 International Conference on High Performance Computing Simulation (HPCS)*. 2019, pp. 761–768. DOI: 10.1109/HPCS48598.2019.9188055.

[33] Yu Xu et al. “Cost-efficient negotiation over multiple resources with reinforcement learning”. In: *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*. IEEE. 2017, pp. 1–6.