Using Reinforcement Learning to Size Tasks for Scientific Workflows

Nicolas Zunker

n.zunker@campus.tu-berlin.de

September 30, 2021

BACHELOR'S THESIS

Mobile Cloud Computing Chair

Technische Universitaet Berlin

Examiner 1: Prof. Dr. Odej Kao

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 selbstaandige und eigenstaandige Anfertigung versichert an Eides statt:

DATE

Nicolas Zunker

Acknowledgment

I would like to thank the following persons for their great support when writing my thesis at the Mobile Cloud Computing Research Group (MCC) at Technische Universitaet Berlin.

NZ

Abstract

The growth of computational power and big data has lead 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 need. 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 can make the entire process more efficient. This is naturally of particular interest to the users of these workflows since it will either lower their costs or speed up the entire workflow or both.

This thesis aims to investigate the allocation of resources to individual tasks and specifically how reinforcement learning can be applied to improve the allocation to be as optimal as possible. The implementation 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 against both eachother as well as the default configuration of resources.

Should such an approach prove useful and provide an improvement in resource usage it would naturally indicate this is an area which should be explored further and that the use of these methods can improve scientific workflows. This would be helpful to both the scientists which use workflow managers as well as the administrators of the execution platforms on which the workflows are 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 [**ResourceProvisioning**] 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 pick the right 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 this it also quite common that the users may not have the requisite knowledge to properly size the tasks themselves or may make a poor estimation [**Predictability**] of what a task's resource requirements may

be. And 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, a university cluster or in a cloud computing set-up) and the computational 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 both very complex and very difficult.

However it is also obvious that sizing tasks properly is of critical importance. Not only are greater efficiency or greater speed always desirable for any situation but specifically for large scale, distributed computations it can make a significant difference because 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 paid for).

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: since agents using reinforcement learning 'learn on the go' by trying to discover the optimal policy for their goal through interaction with the environment and the problem. 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 for any form of machine learning which depends on that is ruled out. Secondly, reinforcement learning is adaptable: since it only ever aims to learn an optimal policy though interactions a reinforcement learning agent is constantly gathering feedback on its actions (even once they have reached a point which 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 initial or default configurations.

1.3 Structure of the Thesis

The remainder of the thesis is structured as follows: first in 2 some background information is given about the ideas and technologies relevant to this paper, then in 3 some related works are discussed which addressed similar topics. After that the approach to the problem is described in 4 and finally in 5 the results are presented and analysed.

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 2.5. Finally, in 2.6 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 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 own equipment, which quickly becomes

infeasible for large and difficult tasks, 2) using a cloud computing service, 3) using a dedicated machine with more resources or 4) using a dedicated 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 paper by [TANNENBAUM et.al] defines a distributed system as one in which "multiple autonomous processors do not share memory but cooperate by sending messages over a communications network". 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 the vast resources available in the system.

When speaking of distributed systems two prominent architectures are grids and clusters.

TODO: batch processing?

In this thesis a cloud computing service is considered a service with an aim to be available all the time and from anywhere, and to provide configurable computing resources to support the needs of the user. The critical component of that description is the word configurable. Cloud computing provides a flexible amount of computing resources so that users only pay for what they need. The most common resources in demand are data storage and computational power. With the continuing growth of the internet and digitalization and the increasing demand for computational power, cloud computing services continue to grow in popularity. Since the resources provided by such services can also located in almost any region they provide a great degree of power and flexibility for users and also liberate from having to

acquire such resources themselves. A given user or group of users no longer need to own and manage their own systems of 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.

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 its other version and nor will any additional configuration be necessary in order to enable them to run on the same host. 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 any 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 the container. The poster child for containerization is Docker. Their philosophy is "build once, deploy anywhere", and many cloud computing services only need to be provided a Dockerfile (the configuration) 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 becomes relevant for scientific workflows because by using containers a workflow can be reliably reproduced on various different execution platforms.

2.3 Scientific Workflows

2.4 Workflow Management Systems

2.5 Nextflow

2.6 Reinforcement Learning

Popularized in the seminal book (TODO: cite!) by Sutton and Barto, reinforcement learning presents a framework for an agent to learn the optimal policy for achieving a given goal in an uncertain environment by interacting with the problem and the environment. And most importantly it is 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 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, to achieve the goal of its designers.

Central to reinforcement learning are the policies and how best to evaluate them. In order to achieve its goal an agent needs to develop a policy that maximizes its reward, then as the agent encounters similar situations it simply follows the policy it has learned. To evaluate a policy it must be compared to the ideal version of itself. For any given problem and its reward function there exists at least one policy which maximizes the reward. At its core, reinforcement learning aims to enable the agent to continuously refine its current policy so that it approaches the optimal policy.

Even more important than the concept of policies is the idea of exploration vs. exploitation. Since the agent initially knows nothing about its environment it must attempt to learn about its environment through exploration. By trying different choices and receiving different rewards the agent can construct a policy that always makes the right choice. But in order to know what the right choice is the bandit must also make the wrong choice so that it can learn not to make it again. Trying different things is the "exploration" and using the knowledge gained from this to make the right choice is the "exploitation". An agent cannot simultaneously explore and exploit. This is 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, and using its current policy to increase its rewards.

In this thesis two specific types of reinforcement learning are considered. First are gradient bandits. The term "Bandits" is a framework for solving problems in which an agent repeatedly returns to an unchanging situation in which there are several choices, each of which lead to unknown results. The analogy used by Sutton and Barto is of a room with several levers with unknown effects (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 which yields the maximum reward from the pulling of certain levers. Applying this to the case of nextflow 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 this (where the policy could be minimizing runtime or maximizing resource usage). Gradient bandits solve this problem of finding the best policy by using the gradient of the reward function to learn a preference for each of the levers. Using gradient ascent the bandits take small steps in the direction of a the ideal preference for each lever which would maximizes the reward. Mathematically speaking this is done through stochastic gradient ascent, for which the formula is : TODO use equation $H_{t+1} = H_t(a) + \alpha \frac{\delta E[R_t]}{\delta H_t(a)}$. This formula aims to increment the preference proportional to the increment's effect on performance. These preferences influence the probability of choosing a given action. Crucially, after choosing an action and receiving the 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} after taking action A_t and receiving reward R_t its preference is updated as follows TODO use begin equation: $H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \hat{R})(1 - \pi_t(A_t))$ and $H_{t+1}(a_t) = H_t(a_t) - \alpha(R_t - \hat{R})\pi_t(a_t)$ for the preferences for each action $a_t != A_t$, where $\hat{R}$ is the average of all the rewards so far. It can be proven that this formula eventually approximates the formula for gradient ascent.

TODO: states and Q or TD learning

Chapter 3

Related Work

This chapter discusses other works in related areas and considers the nuances of their approaches and how they differ from ours.

3.1 Reinforcement Learning to Schedule Tasks/Jobs

In the LearningToSchedule... paper the researchers attempted to 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 IO 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 whereas this thesis looks at resource allocation.

In the SCARL paper researchers use reinforcement learning for scheduling jobs among machines with heterogeneous resources. Their approach 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.

3.2 Reinforcement Learning to Allocate Resources

SmartYARN applied 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 and the actions available to the agent were 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 originally 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 of more resources versus the potential increase in performance they bring. 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.

VMConf 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. One 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 approach 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 achieve 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 spend exploring the state action space for optimal rewards and spend more time exploiting this to achieve better results.

Finally there is the DeepRM paper on resource management using deep reinforcement learning. These 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 complete 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. A key reason that the agent was able to achieve this increased performance was that it learnt 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 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. This is an idea which could be interesting as a topic of further research if one were to expand on this thesis.

3.3 Significant Takeaways from these Works

All of the papers discussed above 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. The majority of the papers also utilized neural networks or deep reinforcement learning. It is also worth noting that most of these papers used runtime in their reward functions. A key difference between all of the above papers and this thesis is that these papers handled the allocation of resources among machines in their system whereas in nextflow the jobs are given a resource allocation and may then be passed on to a platform or cluster management software which handles the actual allocation of the job to a machine.

Chapter 4

Approach

This chapter covers how the problem was addressed and how the proposed solution was integrated into a scientific workflow manager and tested. This chapter will first look over the software architecture of the approach and where it was inserted into the source code of nextflow, then the exact nature of the gradient bandit and the Q-learning approaches will be considered. Finally the structure of the tests performed to try and evaluate these approaches will explained before the results from those tests are presented and discussed in the next chapter.

4.1 Integrating a Solution into a Workflow Manager

Nextflow is a very 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 [**TracingAndVisualisation**]. Nextflow is free open source software and for this thesis a fork of the 20.12.0 version was used. In order to integrate a reinforcement learning approach with the nextflow source code a simple structure was designed which externalised the storage of the data of previous tasks and workflow runs to

a database and when a task needed to be scheduled, the reinforcement learning agent would use the historical data for that task to inform its decision. The structure can be seen in the following figure.

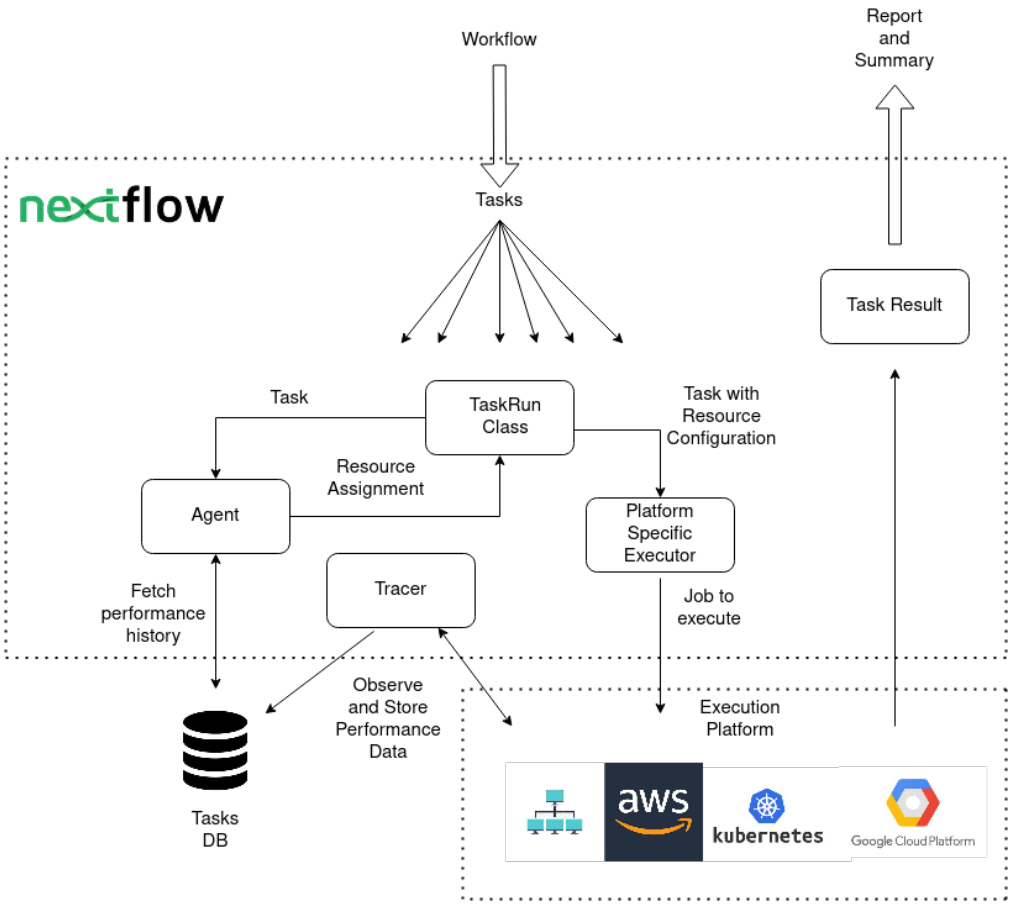


Figure 4.1: High Level Design: Integration of a Reinforcement-Learning Agent into Nextflow

Before a task is ready to be scheduled it is first sent to the 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 select 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 this data, the agent 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. It is important to note that the execution platform will also have its own system for managing, scheduling and executing jobs and processes but from the perspective of the nextflow/agent system all it does is pass on the task with the resource allocations which were chosen. As the task is executed nextflow's tracer module will gather performance data, i.e. peak CPU usage, peak RSS and other such things [**TracingAndVisualisation**]. 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 task scheduling and collecting data about a task is clear, it is time to delve into the specific approaches tried.

4.2 CPU Bandit

4.2.1 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 the task. Its reward was based on a very straightforward function:

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

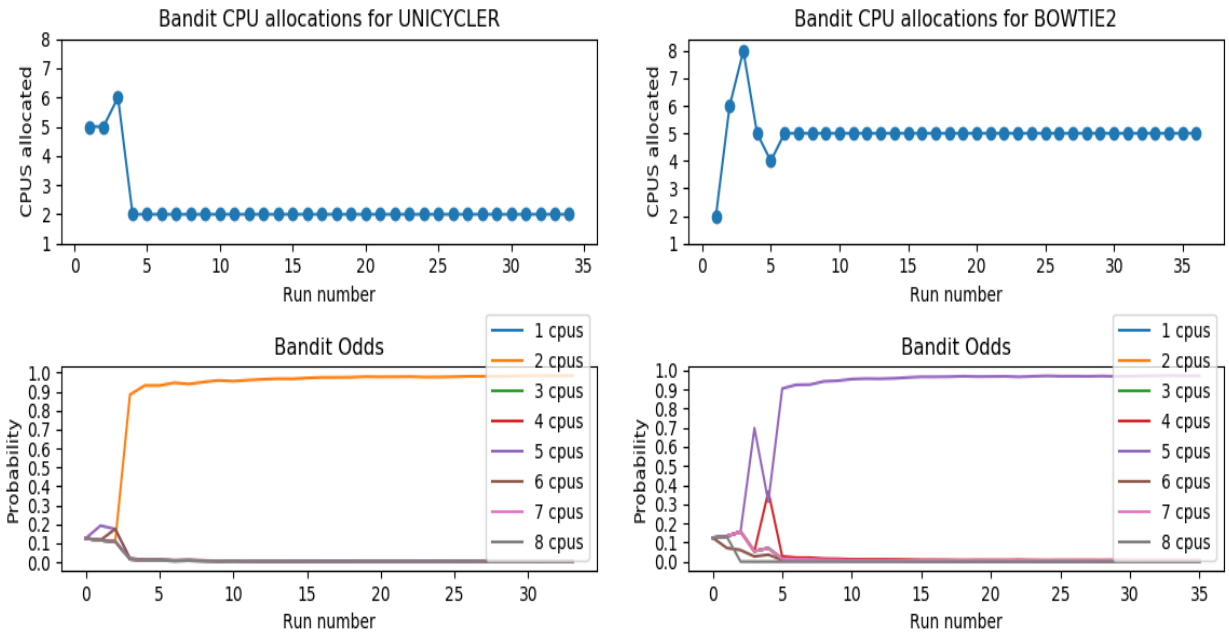
In this equation cpu_usage is a value in percent of the number of single core CPUs used by the task. If a task were assigned 4 CPUs and only effectively used two of them then cpu_usage

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 that the agent is incentivised 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. Additionally, should a task have 100% usage then its penalty is not 0 but is just the time that it ran. 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 immediately minimises the amount of resources wasted, and ultimately the tasks and their workflows would be incredibly slow.

4.2.2 Picking the Right Step Size

One immediate issue encountered with the CPU bandit 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 maximum less than 20 or a function which is known to have a standard deviation less than that. However the reward function used in 4.1 has the bounds $[\ -time * (cpus + 1)\ ,\ -time\ ]$ and as such is effectively unbounded since time can be arbitrarily large (all tasks are of course configured with a maximum runtime but these are user estimates and are always several orders of magnitude larger than the normal time it takes, e.g. maximum runtime is usually an order of days whereas the maximum runtime seen for any task in the course of these experiments was an order of minutes). Additionally over the course of the experiments in this thesis the standard deviation which was observed for the reward function was usually too large for this step size as well. The danger in using an inappropriate constant step size for a given reward function is that when the bandit receives a reward which is larger or smaller 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 with the exploration phase 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. For example if one allocation receives a reward of -150 and and the first two allocations had had an average reward of 350 then for a step size of 0.1 the preference for the new allocation will increase by 5. Because of the exponential nature of the soft max distribution function used on the preferences this increase could result in the bandit converging to always pick the third allocation without any more exploration.



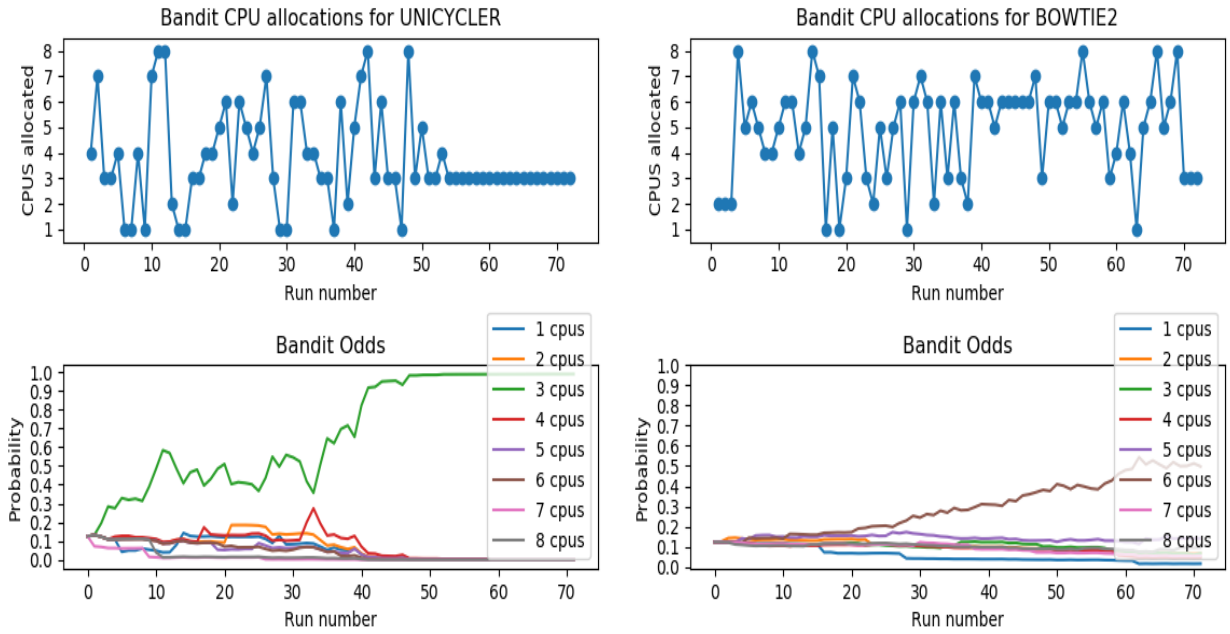
(a) Bandit 1 (b) Bandit 2

Figure 4.2: Example of Two Bandits Converging Too Fast

These two examples are of two tasks which were exhibiting the behaviour described previously. The figures 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 the probability for a given action rises

far too quickly and no other actions are attempted. This is occurring because the step taken by the bandit in the direction of a single actions preference is too large and the resulting probability for that action (probability of choosing a given action is based on a soft max distribution of the preferences for the available actions) grows too fast so that only that action is being chosen.

The solution to this problem is to adjust the step size so that it is not a constant value but is instead based on the average time (historically) that it takes the task to complete. So tasks which have longer average runtimes are given smaller step sizes. The reference equation used was $step\_size = 1/max(reward)$, which for the reward function in 4.1 translated to $step\_size = 1/avg(time)$ since the maximum possible absolute value of the reward would be close to $cpus * avg\_time$. This reference equation is based on the Gradient Bandit example in the Sutton and Barto book, where they used a step size of 0.1 for a reward function which had a statistical mean of 4.



(a) Bandit 1

(b) Bandit 2

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

The above figure shows the same two tasks and their bandits but with the changes to the 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 possible allocation) it does not become so preferred as to be dominant until much later and the bandit continues trying other options as well.

It should be noted however that these changes mean that for optimal performance of the bandits there should already be amount of 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. Since some reference data is needed to compare to the bandits, in order to evaluate their performance, it makes sense to first collect the reference data about the performance of the default configurations and then run the workflows with the bandits active.

4.3 Q-Agent

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 first on the maximum resources of the system and secondly on the default allocation given to the task by the developers of the workflow. When the task was scheduled by the agent for the first time it would start in the default state.

The pseudo code for the Q-learning agent can be seen in 1.

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

Algorithm 1: Q-learning Agent Pseudocode

For the Q-Agent a different reward function was used than for the gradient bandit- primarily because it also had to incorporate memory but also as part of an attempt to try slightly different reward functions and approaches.

$$r = -max(0.1, cpus - cpu\_usage/100) \times (t/avg\_t) \times (mem \times (1 - max(0.75, mem\_usage/100))) \tag{4.2}$$

Here the *memory* variable refers to the memory allocated to the process and *mem_usage* is the value of of the peak RSS of the process 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. 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 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. There are some slight modifications though. 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 amount 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 unused CPUs, slowdown and 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 wind up in different states.

4.4 Testing the Agents

In order to evaluate the performance of these agents they were tested on a 34-core, 128GB 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 tasks take since the reinforcement learning agents need this. After that the agents are tested. The gradient bandit is run 50 times for each workflow and the Q-learning agents 100 times. The inputs for the workflows used are based on custom combinations of different input files from each workflow's full example datasets and their dummy datasets which run very quickyl and are normally used for testing the basic functionality of their workflows. 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.

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.

Chapter 5

Evaluation

This chapter looks at the results achieved and considers their origins as well as their implications going forwards. In 5.1 the results are presented and analysed. Then in **??** considers what could be improved or done differently . Finally 5.3 looks forwards and asks what implications this could have and what the next steps could be.

5.1 Analysis of Results

5.2 Potential Improvements

5.3 Implications

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 [**paxos1998**]. 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 [**paxos2001**] 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.

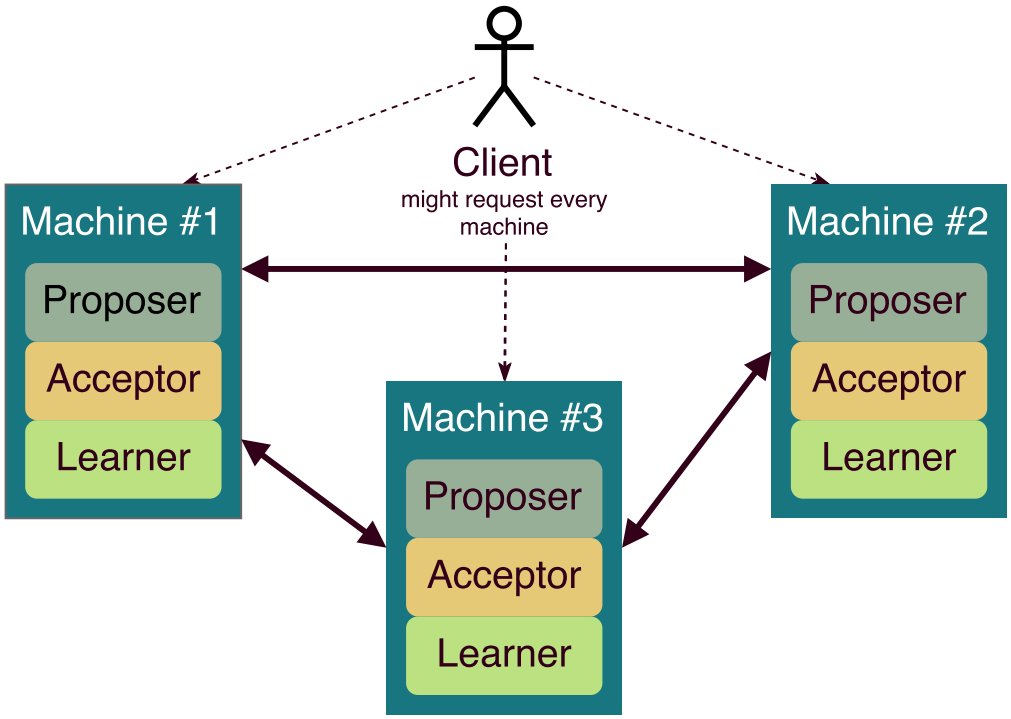


Figure A.1: Example Setup of Three Machines Agreeing on Values Based on the Paxos Algorithm

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 [**versionVektor**]. 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

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

reconciliation is most times not trivial, that's why tools such as Cassandra delegate this task to the application layer [**cassandra2010**].

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 [**lamport1982byzantine**]. 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