Sizing Tasks for Scientific Workflows with Reinforcement Learning

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 cloud computing and big data has lead to increased use of computing clusters to process large datasets. Particularly in the sciences this has become an enormous advantage for analysis and research and 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 computer clusters and cloud computing platforms. One of the most pertinent fields for this is Bioinformatics. Another result of the rise of cloud computing is that there has also been a paradigm shift towards designing programs to be as best suited as possible for cluster environments and this has also led to a shift towards containerization due to the inherent advantage of their 'write once, deploy anywhere' philosophy for such clusters. The result of this combination of factors is the increased popularity of scientific workflows and the frameworks or scripting languages which can be used to make such workflows scalable and reproducible, through containerization, and prepare them for deployment to various cloud computing platforms and management softwares (e.g. Kubernetes, AWS etc.). Nextflow is one such bioinformatics worfklow manager. It includes both a domain specific language to compose tasks into protable worfklows as well as the workflow management softtware for deployment to the various different execution platforms with which it is compatible.

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 differing computational resources and doing so can speed up the entire process and/or make it more efficient. This is naturally of particular interest to the scientists running these workflows since it will either lower the costs or speed up the process 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. Since the tasks within these scientific workflows will reoccur quite often the use of an algorithm which learns "on the go" presents a potential for near optimal resource allocation in the long run. The implementation will be integrated into the nextflow source code.

Differing approaches and reinforcement learning algorithms will be compared and their performance will be judged firstly by how they minimize runtime and secondly by their maximization of resource usage. The primary approach will use a simple gradient bandit and may be compared to other approaches such as TD-learning.

Should such an algorithm 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 will help make scientific workflows more efficient. This would be helpful to both the scientists which use workflow managers as well as the administrators of the execution platforms on which such workflow managers will run.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

This chapter gives a general introduction into 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 thesis in section 1.3.

1.1 Motivation & Problem Description

Thus, many executives feel fear to become unconscious [**TheInquisitiveMind**]. Also, I want to buy **smart home devices** such as the ones listed in table **??**.

1.2 Goal of the Thesis

The goal of this thesis is to XXXXX. The contributions are:

1. Analysis ...
2. Design ...

3. Prototypical implementation of the design.

1.3 Structure of the Thesis

The remainder of the thesis is structured as follows ... .

Chapter 2

Background

This chapter provides background information and explanations of the functionality and purpose of some of the technologies and concepts related to this thesis. To begin the history of cloud 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 target software, nextflow, of this thesis is introduced in section 2.4. Finally, in 2.5 reinforcement learning is touched upon and the benefits it could bring to the problem at hand are briefly explained.

2.1 Cloud Computing

The principle behind computer clusters is to pool the resources of several machines to speed up the execution of a program as well as to leverage specific data and or hardware resources of other machines in the cluster. This has given rise to the concept of cloud computing which takes this principle and aims to make these clusters of machines more accessible and configurable. Cloud computing aims to be available all the time and from anywhere, and to provide configurable computing resources to support the needs of the user. The most

common resources in demand are data storage and computational power. With the rise of the internet and the increasing demand for computational power, cloud computing continues to grow in popularity.

2.2 Containerization

Virtualization has existed for a long time however the overhead of running a virtual machine is often not worth the advantages they provide. Particularly for reproducible software development the most important qualities of the host machine beyond the architecture are the operating system and its filesystem. Since most applications these days need to run on multiple servers and it these can quickly become difficult to manage and because most servers use a similar architecture this has led to increased usage of containers. Containers do not virtualize the architecture of the host machine, instead they use namespaces and control groups to virtualize the operating system, network and the filesystem. This enables containers to isolate processes from the host machine and also from each other. Specifically for the deployment 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 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. 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.

2.3 Scientific Workflows

2.4 Nextflow

2.5 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 its preference is updated as follows TODO use begin equation: $H_{t+1}(A_t) = H_t(A_t) + \alpha(R_t - \hat{R}_t)(1 - \pi_t(A))$ and $H_{t+1}(a_t) = H_t(a_t) - \alpha(R_t - \hat{R}_t)\pi_t(a_t)$ for the preferences for each action $a_t != A_t$, where $\hat{R}_t$ is the average of all the reward 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 discusses the nuances of their approaches and results.

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 which it is scheduled on 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 the 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: 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 bandit 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.

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 %.

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 minimizing cost by using less resources against the need to increase runtime by using more resources. In this paper the researchers used the performance of the app 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.

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 state space was 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 modeling 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 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.

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, represented as images which could be fed into the neural network. The available actions were simply to allocate the jobs. 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 shortest job first metric and a very strong, heuristic based algorithm called Tetris were both outperformed by the researchers' approach. A key takeaway was that the agent achieved this increased performance by learning 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 distinctly 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.

3.3 Takeaways

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 then passed on to a platform or cluster management software which handles the actual allocation of the job to a machine.

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.

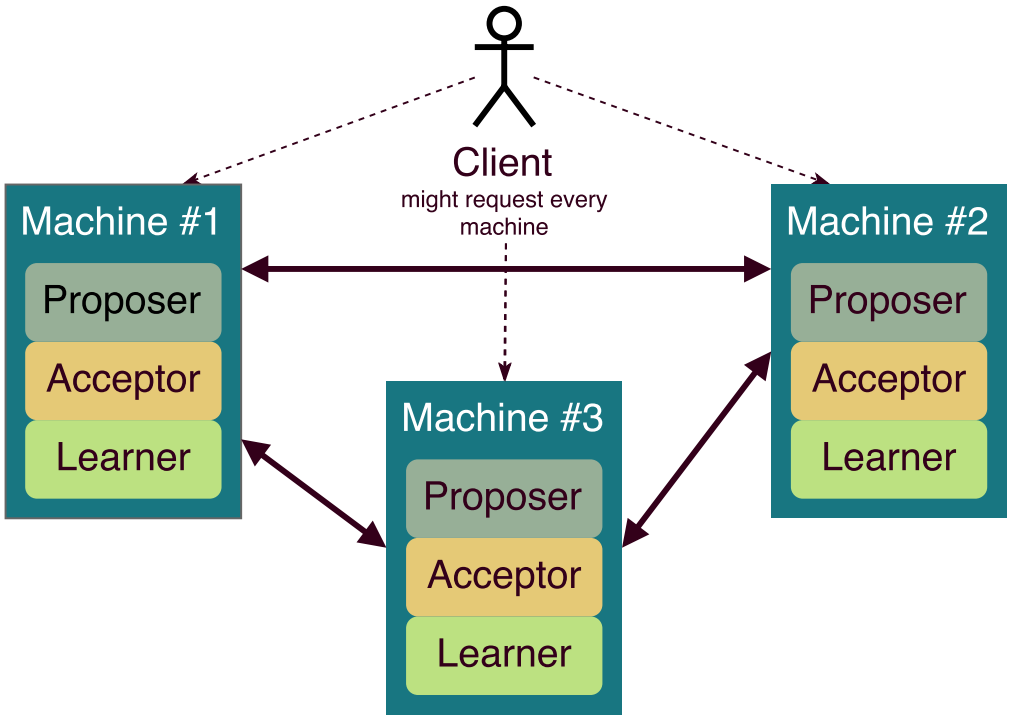

Figure A.1: Example Setup of Three Machines Agreeing on Values Based on the Paxos Algorithm

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