university of groningen

faculty of mathematics and natural sciences

Adaptive provisioning of heterogeneous resources for processing chains

Master's thesis

25th October 2017

Student: M. Kollenstart

Primary supervisor: Prof. dr. A. Lazovik

Secondary supervisor: Dr. V. Andrikopoulos

Primary supervisor TNO: E. Harmsma, MSc

Secondary supervisor TNO: Ir. Ing. E. Langius

Abstract

Efficient utilisation of resources plays an important role in the performance of batch-based task processing. In the cases where different types of resources are used within the same application, it is hard to achieve good utilisation of all the different types of resources. By adaptively altering the size of the available resources for all the different resource types the overall utilisation of resources can be improved. Eliminating the necessity of doing trial runs to determine the desired ratio between resources or having knowledge on the different steps on beforehand. With the current developments in cloud infrastructure, enabling dynamic clusters of resources for applications, this can improve throughput and decrease lead times in the field of computing science.

In this thesis a solution is proposed that tries to come up with the right calculations necessary to create an adaptive system that provisions the right resources at run-time. The solution aims to provide a generic algorithm to estimate the desired ratios of instances processing tasks as well as ratios of the resources that are used by these instances.

To verify the proposed solution a reference framework is provided that tries to eliminate underutilisation of virtual machines in the cloud, where functionally different virtual machines are used in a CPU intensive calculation job. Experiments are conducted based on use-case in which the probability of pipeline failures is determined based on the settlement of soils. These experiments show that the solution is well capable of eliminating large amounts of underutilisation. Resulting in increased throughput and lower lead times.

Acknowledgements

I would like to express my gratitude to all thos who have made it possible for me to complete this research. Especially, my daily supervisors at TNO, Edwin Harmsma, MSc and Ir. Ing. Erik Langius, for their guidance and expertise during the thesis. I would also like to thank my supervisors at the RUG, Prof. dr. Alexander Lazovik and Dr. Vasilios Andrikopoulos for their feedback and discussions on the subject and the written work. The supervision helped me greatly during my research.

Furtermore, I would like thank TNO Groningen, for providing the ability to perform the research using their models and computational resources.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Efficient utilisation of resources is a classical problem in a broad variety of research fields, like logistics and production. In the context of computing science, this means to efficiently use hardware. Currently, physical data-centres often have low overall utilisation, as the data-centres are designed to handle estimated peak loads at the time the data-centres are built. To improve resource utilisation, resources are only required to be available at the moment there is demand. With current developments in cloud computing, renting hardware becomes more and more affordable[1]. The time it takes to acquire new resources in the cloud is currently on a level that within minutes new resources can be acquired, compared with the time it takes to decide new hardware should be present in a physical data-centre and the resources actually being available is several orders of magnitude larger. This developments in cloud computing enables operations engineers to change the size of their resource cluster while running applications. However, managing different types of resources is still a difficult and a mostly manual process. Most cloud providers have functionality to easily scale a group of resources based on utilisation thresholds[2][3], but adaptively altering the size of the resource group based on application demands is not widely available. Especially when different types of resources are used in one resource cluster. To efficiently use the available resources, the demand for different types of resources has to be known at design time. Leaving no room for dynamic resource demand at runtime.

Estimating the idle-time of resources by determining the demand of resources at runtime and taking decisions on the size of the resource cluster leads to lower lead times and thus lower costs. For instance, compute intensive data processing applications, where data is processed via series of compute intensive components, benefits from such an automated decision process regarding the allocation of resources. Linking components, with different resource needs, in a chain leads to a mixed resource demand for different types of resources within the processing chain, resulting in a

[1] Public cloud competition prompts 66% drop in prices since 2013
[2] Microsoft Azure Virtual Machine Scale Sets scaling
[3] Amazon Auto Scaling

heterogeneous resource demand.

The management of heterogeneous resources is a widely researched topic in the field of manufacturing. Issues with utilisation of production capacity and the reduction of waste can be related to utilisation of computing resources. For instance, managing the resources of each manufacturing step of a product chain is an important factor for the profitability of a company. Production rates of intermediary steps should match throughout the processing chain to prevent piling up of semi-finished products. A proven concept to decrease lead times and reduce inventory levels throughout the manufacturing process is just-in-time manufacturing (JIT). Henry Ford described this manufacturing process in his book *My Life and Work*[1]:

> "*We have found in buying materials that it is not worthwhile to buy for other than immediate needs. We buy only enough to fit into the plan of production, taking into consideration the state of transportation at the time. If transportation were perfect and an even flow of materials could be assured, it would not be necessary to carry any stock whatsoever. The carloads of raw materials would arrive on schedule and in the planned order and amounts, and go from the railway cars into production. That would save a great deal of money, for it would give a very rapid turnover and thus decrease the amount of money tied up in materials.*"

The most famous example of just-in-time manufacturing is the Toyota Production System[2]. In which Toyota addressed the main problem of overproduction and the waste introduced by high inventory levels. In Figure 1.1 a graphical interpretation of the Toyota Production System is displayed. The key concept in this figure is that the production reacts on the demand of the dealer, and the different links in the product chain are only processing cars when there is a demand for them. Preventing the production of intermediate products that might never be used for actual cars.

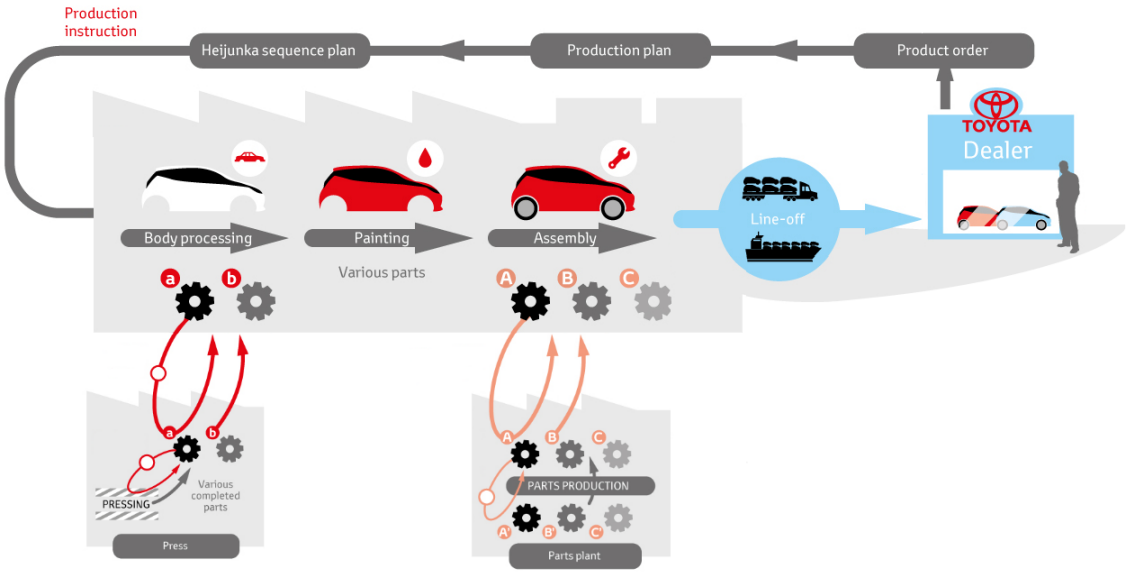


Figure 1.1: Toyota Production System[4]

To be able to generalise the kind of problems addressed in this research, the Toyota Production System example is used to describe the key elements that can indicate the applicability of this research on other types of problems.

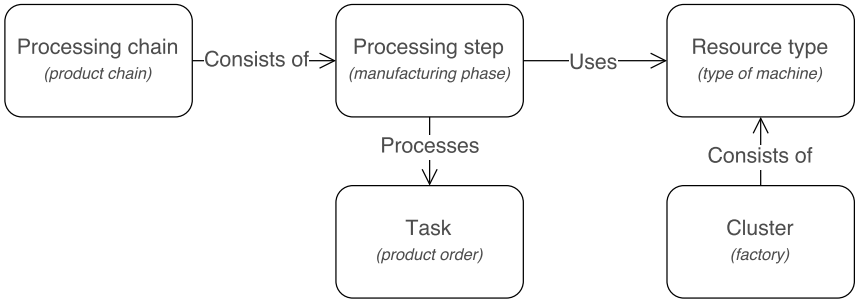


Figure 1.2: Relationships between definitions with their corresponding analogies from the Toyota example.

The relationship between the definitions described below is shown in Figure 1.2. Where arrows indicate one-to-many relationships, so a processing chain consists out of multiple processing steps and a processing step processes multiple tasks.

Processing chain

A chain of *processing steps* that solve a problem based on a certain input. In the Toyota case five processing steps are distinguished: dealer, body processing, painting, assembly, and line-off.

[4] https://www.toyota-europe.com/world-of-toyota/this-is-toyota/toyota-production-system

Processing step

A link in the chain that independently processes input from the previous step and outputs it for the next step. A *processing step* can have requirements on the type of resources needed to complete the process. For instance, paint guns are required to successfully execute the painting *processing step*. Each *processing step* can be run in parallel. So, for instance, multiple cars can be assembled when there are multiple assembly stations.

Task

Element, or sequence of elements, that flow as atomic unit through the *processing chain* transforming at each step. A product order is a task at the beginning of the *processing chain*, the task flows through the *processing chain* transforming into the desired output of a car.

Resource type

A type of resources needed to fulfil a *processing step*. Each processing step can have a different set of requirements on *resource types*. For instance, the painting processing step requires paint guns to be able to paint the car and the body processing step needs welding machines to form the body. In this research, we focus on reusable resource types, like the machines and personnel. And not on resource types used once, like products bought from other manufacturers that are used in the car, i.e. tyres or windows acquired from outside the processing chain.

Cluster

All of the *resources* available to the *processing chain*, partitioned into distinct sets of *resource types*. So for instance, the cluster consists of all the paint guns and the welding machines, each in their own distinct set.

Managing the cluster of resources available to the processing chain is the key element of this research. The distribution of resource types in the cluster is an important factor on how the tasks flow through the processing chain. When the resources required by a specific processing step are in abundance two things can happen:

1. All of the resources are used and too much intermediate results are created for the next processing step to handle.

2. Not all resources are used by the processing step wasting available resources.

For example, lets assume the body processing needs 1 welding machine for 3 hours to process one body and the painting step needs 1 paint gun for half an hour to paint the body. When there are 9 welding machines and only 1 paint gun available to the processing chain, either 3 welding machines won't be used, or a queue of welded chassis is piling up before the painting step increasing with 1 chassis per hour. As the welding machines are capable of handling 9 chassis per 3 hours, and the paint gun only handles 6 chassis per 3 hours. To get an optimal flow through the system, with using the painting step fully, 6 welding machines are needed per paint gun. This way, the 6 welding machines can be used to process 6 chassis per 3 hours and

the single paint gun can also be used to process 6 chassis in 3 hours. In this example the ratio can be determined very easily, as the ratio of welding machines to paint guns is 6 to 1.

The more interesting case is when this ratio cannot be determined easily and the ratio can dynamically change due to different input characteristics or other external effects. For instance, when Toyota produces different types cars in the same factory and the body processing of the different types of cars differs. In this case, the fixed ratios of the previous example do not hold any longer, as it depends on the types of cars requested by the Toyota dealer. An automated system could be able to decide the ratio between resource types needed in the processing chain. Resulting in a fine-grained resource distribution changing over time, instead of having a fixed cluster for the whole lifespan of the processing chain.

This research introduces an approach on how decisions on the distribution of resource types can be made automatically during the lifespan of the processing chain, using information of the utilisation of the resources and information of the processing chain and processing steps. This approach is verified by applying it to the use-case of STOOP, where compute intensive models for soil settling and pipe stresses are used to determine the chances of gas- and water-pipeline failures.

1.1 TNO

The research is done at the Dutch organisation for applied scientific research (TNO), aimed at creating a generic solution with the project Sensortechnology applied to underground pipelineinfrastructures[5] (STOOP) as case study.

TNO is an independent Dutch research organisation. The goal of TNO is to apply scientific research in practice. The mission of TNO is to connect people and knowledge to create innovations that enhance the competitive strength or industry and the well-being of society in a sustainable way.

The STOOP project is executed at the Monitoring and Control Services department of TNO, the goal of the STOOP project is to monitor underground pipeline networks in order to be able to make an assessment of the integrity of the pipelines. So that asset managers can make an objective assessment whether or not a pipeline should be replaced. The STOOP project is be discussed further as case study in the evaluation in Section 7.1, where the STOOP project provides meaningful scenarios to evaluate.

[5] Innovative techniques for monitoring infrastructures

1.2 Research question

The main focus of this research lays in the decision process for an efficient distribution of different resource types. Combining information on both the application level and the infrastructure level could allow us to take more informed decisions. Important is the fact that the decision taking should be a continuous process while applications run on a cluster, as the resource demand is not known beforehand and can be variable throughout the life-time of applications.

Two different types of distributions should be computed: the distribution of processing step instances, and the distribution of resource types. There is a close relation between these two distributions, so the decision taking should consider this relationship.

This leads to the main research question:

"How to adaptively provision heterogeneous resources for compute intensive data processing?"

In order to answer this main research question, several sub-questions have to be answered:

1. *Which metrics are needed to provide a clear overview of the status of the processing chain?*

   Different metrics are needed to determine where in the processing chain over- and underutilisation occurs. What information do we need from both the application level and the infrastructure level to have a clear overview of the whole processing chain?

2. *How to continuously decide efficient distributions of processing step instances?*

   Subsequent processing steps should perform at the same throughput of tasks to prevent over- and underutilisation of these steps. How can the metrics be used to determine an efficient distribution of processing step instances for the whole processing chain at a certain point in the lifespan of the processing chain.

3. *How to continuously decide efficient distributions of resource types in the cluster?*

   Like the decision for efficient distributions of processing step instances, efficient distributions for the resource types in the cluster should be decided on. How can the cluster be distributed in the case of possibly multiple running processing chains on the same cluster.

1.3 Contribution

This research focuses on introducing a solution for the problems that are discussed in Chapter 3. The solution gives an new approach on how to efficiently use heterogeneous resources in an adaptive way. An algorithm is proposed to determine the ideal distribution of processing steps and resources, based on metrics on application level as well as on the resource level. The solution gives a general approach that has applicability on a broader level than just the computing science field.

A reference implementation provides a basic framework that supports the solution so that the solution can be verified. The framework is not a fine-tuned and optimised implementation, but given the dynamic nature of problems being handled this does not influence the verification of the solution.

1.4 Document structure

This chapter briefly introduced the topic of this research, with the main focus and research question. In Chapter 2 the state of the art of different research fields subject are discussed. The problem statement is discussed in Chapter 3, where the problems related to the research question is elaborated upon. Chapter 4 provides a general solution that solves the research question and the problems stated in Chapter 3. The solution is worked out into a reference architecture in Chapter 5. An implementation based on the architecture is briefly discussed in Chapter 6, where the implementation and realisation of the reference framework are introduced. In Chapter 7 the reference framework is tested and evaluated against test scenarios from the STOOP project. Chapter 8 provides some directions for future work of this research. And finally, in Chapter 9 a conclusion is given based on the evaluation and the applicability of the framework.

Chapter 2

Related work

In the different fields that are connected to this research, like dynamic provisioning, data processing, and task scheduling, a lot of research is done that can contribute to this research. Most research focuses on single research fields and less research on the wide topic of this research, where dynamic provisioning of heterogeneous resources is used to support data processing. The related work is split up into different aspects of the research, corresponding with the blocks identified in Figure 1.2: processing chain, processing steps, and the resource cluster. The related work is often on the topic of two or more blocks, the discussion of the related work addresses the research in the block the main contribution lays to get a clear overview on the state of the art.

2.1 Processing chains

Processing chains can be seen as the logic for processing tasks, with processing steps transforming input tasks into the desired input for the next processing step. Therefore, it is important to identify how such a processing chain is designed so that processing steps are able to process tasks. Depending on the kind of problems that are solved by the processing chain, there can be different constraints on the processing guarantees that define how faults in processing tasks by processing steps are handled.

2.1.1 Chain design

To be able to process tasks through processing chains, processing steps are required to have great scalability. As the number of processing steps must be able to change easily without introducing large overheads when large numbers of processing step instances are deployed.

Dean and Ghemawat[3], have published their research at Google for distributed and scalable processing of data. The MapReduce programming model is introduced, which is a simple model that enables developers to design highly scalable and distributed programs that can be run on a large scale cloud infrastructure. The abstraction of the model is inspired by the map and reduce primitives originating from functional languages, like Haskell. For the map primitive a single function is applied to all entries in a list. And for the reduce primitive all these records are reduced to a single entry. The reduce operation always reduces two elements into one element of the same type. Another important aspect of the MapReduce model is that all the functions are stateless and always give the same result when called with the same input variables. Due to this stateless nature of the functions, they are very well suited for distributed execution on partitions of a data-set. Looking at the Toyota example from Chapter 1, this principle can be applied to the automotive sector. Where, for instance, the body processing step is a map phase that for each order executes a function, i.e. creating the metal body of the car. However, reducing a two assembled cars into one does not make sense. But when shipping cars the set of cars in the cargo ship can be seen as the reduction of cars to a single entity.

To be able to run the MapReduce model well on heterogeneous clusters, Ahmad et al.[4] introduced Tarazu. The MapReduce model is evaluated against heterogeneous clusters, in this case the heterogeneity lays in different performance of different types of resources. A cluster mixed with Intel Xeon processors and Intel Atom processors is tested, naturally the Xeon processors outperform the Atom processors significantly. In the MapReduce model the notion of heterogeneous clusters is absent, a homogeneous set of workers is assumed so the tasks can be scheduled on the different workers evenly. A suite of optimisations to improve MapReduce performance on heterogeneous clusters is introduced, consisting of: load balancing of map operations, scheduling of map operations, and load balancing of reduce operations.

Available solutions

In the field of computing science, there are several largely used solutions that provide a design language and a runtime platform that executes the designed applications.

Hadoop MapReduce MapReduce is one of the core components of Apache Hadoop. It is the open-source implementation of the programming model introduced by Dean and Ghemawat[3]. Heterogeneous resources cannot be used efficiently by Hadoop MapReduce, the Tarazu solution can assist in the case where the heterogeneity of the resources lays in the performance of different resources. But when processing steps are strictly limited to use a specific resource and can't run on other resources the Tarazu solution does not help.

Apache Spark Apache Spark[1] is a processing engine that supports batch processing as well as streaming processing, in the form of micro-batch processing. The performance of Spark is significantly better than Hadoops MapReduce, because it uses in-memory transfer of data between subsequent stages instead of using much slower disks. Therefore, Spark is currently preferred over Hadoops MapReduce by most developers. Spark is often used in combination with Hadoops Distributed File System (HDFS) to support parallel data processing while taking data locality in mind.

One of the biggest drawbacks of Spark, for this research, is that it assumes that all workers are homogeneous and makes, therefore, no scheduling difference for tasks. This is a clear design decision that works very well for cases where homogeneous resources are used, but limits the possibilities to support models that do not run on the platforms Spark is designed for, like models that require Windows.

Apache Storm Apache Storm[2] is a parallel and distributed streaming data framework, for which Toshniwal et al.[5] described how Storm is used at Twitter. It uses micro-batching to process tasks through a series of processing steps, called bolts. It is mainly focused on near-real time processing of tasks.

Just like Apache Spark, Storm assumes homogeneous resources. Complicating the deployment of processing chains with processing steps that have strictly different resource constraints.

2.1.2 Processing guarantees

In distributed data processing the chances on errors in the processing are significantly larger, as failures in network transmission occur and the chance that at least one resource fails increases when more resources are introduced to the cluster.

In the description of how Apache Storm is used at Twitter[5], two of the most interesting processing guarantees for distributed processing are discussed:

at least once
: The at least once guarantee can be simply be realised by adding an acknowledgement when a piece of data is processed correctly. If after a certain time-out a piece of data is not acknowledged it can be rescheduled. If an element takes longer to process than the time-out is this means that the element will be handled twice in the pipeline.

at most once
: The at most once guarantee means that the piece of data is scheduled once and will not be rescheduled. In case of a failure in processing a piece of

[1] Apache Spark
[2] Apache Storm

data, this element will be dropped. This means that there is no necessity of acknowledging of pieces of data.

Another guarantee one could think of is that each piece of data is processed exactly one in each processing step. This however requires complete synchronisation of data in-between processing steps, introducing an exponential bottleneck when applications are deployed on a large scale.

Depending on the processing chain a choice should be made on the guarantee of the data processing.

2.1.3 Summary

The MapReduce programming model is an interesting principle to start from, as it is widely used and a lot of research is done on the distributed data processing with for instance Apache Spark and Apache Storm.

However, almost all of the current research and available solutions assume a homogeneous set of resource. Or at least a homogeneous set of resources that are compatible with each other, so that tasks can be processed by all of the available resources. In this research we want to investigate whether it is possible to have strictly separate types of resources that are not compatible with each other.

In this research a reference design is made based upon a newly created framework, as the dependency of the existing frameworks on homogeneous workers is too strong to easily modify these frameworks. In the newly created framework design elements of mainly the MapReduce programming model is incorporated.

2.2 Processing steps

The discussion of research on processing steps contains mainly the runtime aspects, as the design of the processing steps has been handled at the discussing on processing chains. In this section the balancing of processing steps is discussed, as well as methods on how to scale and deploy distributed processing steps.

2.2.1 Balancing

To be able to estimate resource demands of processing chains it is important to have a good knowledge of the running processing chain and the resources it uses, while avoiding introducing too much stress on the resources for gathering load metrics.

There are a lot of different approaches to measure the load on a resource instance, or the load on a cluster of resources. Both at application level as well as on operating system or hardware level.

For time series streaming data, Xing et al.[6] introduced a greedy algorithm for push-based continuous streaming that aims at avoiding overload and minimising end-to-end latency. The greedy algorithm tries to correlate stream rates to create a balanced operator mapping plan where the average load variance is minimised or the average load correlation is maximised, i.e. don't synchronise the load burst of two input streams. This way the load is balanced by not letting the inputs create high peaks at the same time and be nearly idle in between these peaks. This method can be mainly used when different processing chains run on the same platform with a burst like character.

Shah et al.[7] have introduced Flux, a data-flow operator between producers and consumers. The goal of Flux is to re-partition data coming from producers so that an even distribution is created for the consumers attached to it. This way the possibility of bottlenecks in the data are prevented.

A back-pressure method is described, by Collins et al.[8], to handle congestion on a single machine with multi-core processor. Processing steps, or filters, are moved around across the available cores and the ratio between filters is altered through the back-pressure algorithm. This algorithm indicates which filters work faster than others, by letting filters ask an amount of tasks that they are able to receive in their buffers. This way there is no buffer overflow and are no tasks dropped. The paper focuses on computations on a single machine, with a case study of the JPEG encoder, but the principle can be applied to flows distributed over multiple machines.

2.2.2 Distributed deployment

The research on scientific workflows, currently mainly discussed in the field of Biology, is closely related to the set of problems that we try to solve in this research. Where chains of operations are applied on input data. Containerisation is a powerful tool to isolate instances of different components of workflows. Which can be applicable to our processing chains, where for instance each processing step resides in an isolated container. And the resource types are different kind of virtual machines, on which these containers can run.

Zheng et al. [9] discusses the importance of isolation for scientific workflows, mainly because of the reproducibility that containerisation gives. As the environments are clearly described in an image. Several options of using Docker are investigated: wrapper scripts, worker in container, container in worker, and shared containers. The performance of these options are tested too see the overhead of using Docker in scientific workflows.

A popular biology workflow system, Galaxy, is integrated into a flexible container-based solution by Liu et al.[10]. This enables the workflow system to fully utilise cloud resources. To make it easier for scientists without computing-science background, different types of using docker are investigated. Namely, Docker in Docker, Sibling Docker, and Tool in Docker.

Challenges that arise when taking ordinary scientific workflows to the cloud is discussed by Zhao et al.[11]. A generic reference platform is presented that can be used to alter current workflow platforms to use in the cloud. An example is given with the Swift workflow management system that is integrated to be used in the cloud.

2.2.3 Task scheduling

As tasks are distributed over a set of instances, the tasks should be scheduled evenly over all the available instances. This also means that instances that have less performance should receive less tasks than well performing instances.

Work stealing is an effective method for scheduling fine-grained tasks, mainly focused on multi-core processors within one machine. The algorithm dates back to 1981 when Burton and Sleep[12] proposed the idea. The algorithm works with using concurrent queues for each core in the processor. Each core processes tasks on its own queue, unless the core is underutilised, at that moment the core attempts to steal tasks directly from the task queue of other cores. This algorithm has many variants that are implemented over time, for example Acar et al.[13], where the queues are strictly private and stealing of work occurs via message passing.

The scheduling done in MapReduce, described by Dean and Ghemawat[3], works in a relative simple way. The master node is responsible to schedule tasks on worker nodes, when a worker node is able to receive new tasks it will notify the master that it has a slot available for a new task. The master node will then check if there are tasks that are local to that slave node, so that the data transfer overhead will be minimised. To overcome the problem of tasks that take too long to execute, i.e. stragglers, reducing the performance of the complete processing chain, the master will start speculative tasks which compute the same operation on the same data. Using this technique the speculative tasks can finish earlier than the original task, resulting in a lower overall execution time. To indicate if tasks are stragglers MapReduce will look at the progress of a operation and the time it took to reach that progress, this will be compared to the mean of the execution times for that specific operation. The open-source implementation of MapReduce uses thresholds of task execution times instead of monitoring the task progress metrics.

The scheduling mechanism in MapReduce and Hadoop assumes homogeneity of their workers. Meaning that all nodes work at roughly the same rate and that progress of tasks occurs at a constant rate. However, when run on rented hardware in clouds these assumptions do not always hold. For instance, the number of virtual machines that are deployed on a single machine can have a great impact on performance when all the virtual machines compete for resources on the machine. Zaharia et al.[14] proposed a scheduler for Hadoop that tries to overcome this kind of heterogeneity of resource performance. This scheduler gives tasks a ranking based on the estimated time the task will end if it is scheduled now, this way the scheduler has more opportunity to schedule speculative tasks during the execution of the job. Also, the

scheduler tries to schedule speculative tasks on faster nodes instead of on all nodes, to relieve slow nodes as much as possible.

2.2.4 Instance scaling

To make sure that processing step instances have access to the right resource that is allocated to them, a cluster manager is needed that is able to match the resources to processing step instances. Looking at the opportunity of isolated containers, scaling up instances should have little impact on the performance.

In an extensive survey, by Peinl et al.[15], several aspects of cluster management are discussed on available management solutions for running containerised software in a cluster: image registry, container management, cluster scheduler, orchestration, service discovery, storage, software defined networks, load-balancer, monitoring, and management suites. The paper is published in April of 2016, most of the discussed solutions are still available and maintained, but a lot of new developments have been made in this field. For this research the cluster schedulers, the orchestration, and software defined network solutions are the most interesting.

Researchers at Google published, by Verma et al.[16], their Borg. Google's Borg platform is a cluster manager that Google uses for their own applications. Hundred thousands of jobs, from many thousand of different applications across a number of clusters each with up to thousands of machines. Google's Borg system is open-sourced as Kubernetes, which is currently one of the most used cluster manager and container provisioning system for Docker.

Available solutions

To be able to launch containers on multiple hosts while communication between containers across hosts is still possible, container cluster managers create a layer of abstraction for the cluster and provide tools to manage and provision containers in the cluster.

Docker Swarm The integrated cluster manager of Docker is Docker Swarm[3]. It is integrated in the command line interface and API of Docker. One of the most significant features Swarm offers is the multi-host networking, especially the multi-host networking with Linux nodes as well as Windows nodes. Currently Swarm is the only solution that offers this functionality.

Swarm is built around services, container declarations for which a desired amount of instances can be set. Swarm ensures that the service is running and that the amount of replicas matches the description. For these services service discovery, by using DNS entries for services and instances, is available.

[3] Docker Swarm mode

Kubernetes As discussed before, Googles Borg system was open-sourced as Kubernetes[4]. The principal of Kubernetes differs quite a bit with Docker Swarm. Kubernetes has a modular design, and plugins are available that change the behaviour. For instance, Kubernetes doesn't have multi-host networking support built-in, but supports a large number of Container Network Plugins that provide this functionality.

The basis concept of Kubernetes is the pod structure, a collection of containers tightly scheduled together. Containers inside a pod are likely to have quite a lot of communication between each other and are able to share data volumes. To be able to expose pods to other pods or to the host machine, services are used to expose ports.

Rancher Rancher[5] is an abstraction on top of other cluster managers. By default, it uses Cattle, an alternative to Docker Swarm and Kubernetes, but it also supports Docker Swarm and Kubernetes as back-ends for Rancher to run on. Rancher provides a very powerful web interface, that enables users to setup a cluster very fast and get a good overview of the cluster and the services.

However, Rancher seems to be caught up by Docker Swarm and Kubernetes. Which are more robust and have the same capabilities as Rancher.

Apache Mesos & Marathon Where the previous container cluster managers discussed are built upon Docker and share quite the same functionality. Apache Mesos[6], the cluster manager, and Apache Marathon, the container orchestrator, have a slightly different approach. Mesos focuses on isolating resources, providing an abstraction layer for computing elements, able to handle larger amounts of hosts.

Mesos can run dockerised applications as well as "ordinary" programs. But a lot of functionalities that are available at default in other managers have to be configured and managed manually.

Marathon is the layer on top of Mesos that enables orchestration of containers. It is designed to handle long-running applications, with possible stateful applications.

2.2.5 Summary

Several techniques are discussed that are able to balance the flow through a processing chain, by using intelligent scheduling algorithms or via back-pressure. These techniques can be interesting to use or adapt to prevent in-balance of processing steps. For usage in CPU-intensive tasks, the back-pressure algorithm seems to be

[4] Kubernetes
[5] Rancher
[6] Apache Mesos

the most interesting, as it has great scalability and is able to handle differences in throughput of instances.

For the deployment of processing step instances, containerised solutions tend to be very powerful as each container is fully independent of other instances. This requires the processing steps to be stateless so that the distributed deployment does not impact the scalability of the processing chain. In the field of Biology, many efforts are made to make it as easy as possible for biologists to deploy and run their own application. For this research this is not an important factor, therefore, approaches like Docker in Docker or other exotic forms of Docker usage are not being used. But containerisation itself is very powerful as it provides easy deployment of large scale applications as well as improving the reproducibility of scenarios.

For the scheduling of tasks in the processing chain a combination of the scheduling as proposed in MapReduce and the back-pressure algorithm seems to be a powerful combination. As the number of instances of each processing steps should be able to change, using only back-pressure is not recommended as each instance is then required to know which instances are available to request tasks. Therefore, asking a master for a certain amount of tasks which in turn distributes that to appropriate instances seems to be a promising approach.

To manage these containers, mostly via the Docker ecosystem, a large number of products exists that abstract the notion of hosts individually. By using a container cluster manager, all the resources of the cluster are available via a single interface. Allowing large number of containers to run on a cluster of resources with the capability of the containers communicating with each other on a shared network space. Given the fact that most container cluster managers share a basic set of functionalities, there will not be made a decision on the cluster manager until the realisation of the reference framework.

2.3 Resource clusters

The cluster of resources available to the processing chain is supposed to scale up and down the different types of resources individually, creating a cluster that fits the current processing chain deployed on the cluster. Two aspects of the resource cluster are interesting to investigate, namely the scaling of resource types and the ability of having heterogeneous clusters that are able to communicate freely between each other.

2.3.1 Resource type scaling

Adaptive provisioning is an important factor of this research. Managing the set of resources is a key aspect in preventing over- and underutilisation of resources by processing chains.

In the field of scientific workflows, comparable to the processing chains discussed in this research, Ostermann et al.[17] proposed four provisioning algorithms for managing virtual machines in the cloud: Cloud start, instance type, Grid rescheduling, and Cloud stop. These algorithms make sure the right amount of machines are provisioned, as well as the most suitable machines for the processing chain being executed. The use of cloud resources as extension upon existing grid infrastructures is investigated. A just-in-time workflow scheduler is used to be able to utilise the changing cluster fully. The provisioning rules are based on execution time of tasks combined with fuzzy resource descriptions, based on the virtual machine instance types at Amazon expressed in EC2 Compute Units.

The Aneka platform is introduced by Buyya et al.[18]. Aneka is a resource provisioning platform that is able to provision virtual machines on Microsoft Azure as well as on Amazon EC2. A deadline-based algorithm is used to provision resources in order to meet deadlines based on time.

Zhang et al.[19] have presented Harmony, a dynamic resource provisioning scheme for the cloud. The K-means clustering algorithm is used to divide workload into distinct task classes with similar resource and performance requirements. A dynamic capacity provisioning model for heterogeneous resources is used to determine the resource needs of a pipeline. Harmony tries to balance between energy saving and scheduling delays, while considering the reconfiguration costs of provisioning virtual machines.

Zhang et al.[20] discusses a method of CPU demand approximation to predict the needed resources for multi-tier applications. The method works on multi-tier applications with a network of queues placed between the different tiers. The model is capable of modelling diverse workloads with changing CPU demand over time. Approximation of resource demands is done by using statistical regression based on the CPU demand along all the tiers in the system.

To distribute data between multiple data-centres to utilise energy price differences, Xu et al.[21] proposed capacity allocation and load shifting schemes. Not only the costs are considered, but also the outage probability is taken into account as data-centres have to keep enough resource buffers to handle outages. The ratio-based load swapping scheme, where data-centres can shift a portion of its load to other data-centres, seems to be the most interesting.

Kansal et al.[22] have studied whether or not they can reduce carbon emission by turning off underutilised servers. An overview of load balancing techniques for the cloud is given, with the corresponding load metrics that are used by these techniques. The goal of this paper is to indicate whether or not data-centres can be more ecologic friendly by reducing carbon emissions when shutting down idle servers. But none of the discussed load balancing techniques uses energy consumption or carbon emission as metric.

2.3.2 Heterogeneous clusters

Heterogeneous clusters covers a wide area of clusters with heterogeneous hardware or software. Most of the research is done in the field of heterogeneous hardware, where heterogeneous hardware in the cloud lead to varying execution times.

An extension on OpenStack is published by Crago et al.[23], this extension let users provision VMs with more heterogeneity than currently is available in OpenStack. With the possibility to, for instance, request a number of accelerators or CPU architectures. This is done in a similar way as currently is done by cloud providers, with predefined resource sets. But the extension allows users to give key-value pairs with additional requirements for the resource. The publication mainly focuses on the technical implementation in OpenStack.

To be able to utilise less powerful resources, Thai et al.[24] introduced a scheduler for heterogeneous clusters. The heterogeneity discussed is in the performance difference between resources. Service Level Objectives are used to determine priority for jobs. By running jobs on less powerful resources a significant cost saving realised with reduction of deadline violations.

Another approach for utilising a heterogeneous cluster with machine with and without GPU is to probe with small tasks and calculating statistics about this probe step. Chen et al.[25] used this approach to accelerate the MapReduce paradigm using a cluster with nodes with GPU, with unknown characteristics, i.e. there is no information about the exact performance of the GPUs. The statistics in this case are used to dynamically change the task block sizes to fully utilise the available GPUs in the cluster.

Like the previous approach Shirahata et al.[26] presented a method to perform a similar way of probing with MapReduce and Hadoop on a mixed GPU and non-GPU cluster. By using Hadoop to schedule jobs, separate CPU and GPU binaries are used. A new job is started firstly with both CPU and GPU capabilities, and metrics of the CPU usage and GPU usage is monitored to see if the task is meant to run on a CPU or a GPU. These metrics are then used to determine the CPU-GPU ratio for that type of task.

Hassaan et al.[27] introduced an extension on Apache Spark to run Spark jobs on a heterogeneous cluster. The end-to-end framework introduced uses SparkGPU. The framework is able to process streamed data on a heterogeneous cluster of CPUs and GPUs. To be able to run programs on the GPU from within Java or Scala applications, Java Native Access is used for communication between user program and GPUs. Also, they investigated if a manager application on the host machine is useful to reduce the overhead that JNA introduces.

2.3.3 Summary

The current adaptive provisioning of resources is mainly focused on scaling a resource type up or down based on the request load of the applications running on the resources. For batch-based applications, most likely, all the requests are available at the beginning of the application. This makes it harder to scale up resource types, as in most cases this would mean that the amount of instances would increase heavily when a processing chain is deployed. Resulting in the fact that the overhead of provisioning resources increases, which is harmful when the costs and the lead time of the processing chain are considered important features of the processing chain. This is why the solution provides a new algorithm to estimate the demand for resources, which is not depend heavily on a threshold value for the utilisation.

For heterogeneous clusters, most research is done based on heterogeneous hardware that is able to run all tasks, where the different resource types have different performance. This approach makes it easier to schedule tasks, as all tasks are able to run on each resource. In this research the focus lays on heterogeneous clusters based on the software layer, where applications can only run on one of the resource types. So processing chains are split up into containers that are required to run on specific groups of nodes, thus, making approaches like probing tasks to check for differences in processing throughput less effective. So part of the architecture proposed in Chapter 5 covers the scheduling of tasks and instances on different resource types.

Chapter 3

Problem statement

In this chapter the problem is elaborated upon, giving background information on the research question. The focus lays on describing what the problem in adaptive provisioning of heterogeneous resources is with respect to compute intensive data processing. As discussed in Section 1.2, the research question can be split up into three separate parts. The main research question is: *"How to adaptively provision heterogeneous resources for compute intensive data processing?"*. To be able to reason about the problem being tackled in this research, the kind of applications that will run on the adaptively provisioned cluster are discussed. Afterwards, the different aspects of the research question are discussed through the sub-questions identified earlier.

1. Which metrics are needed to provide a clear overview of the status of the processing chain?
2. How to continuously decide efficient distributions of processing step instances?
3. How to continuously decide efficient distributions of resource types in the cluster?

The theoretical part of the research can be solved by answering these sub-questions. The implementation side of the research question is handled in the architecture and the realisation (Chapters 5 and 6). However, the decisions taken there do not effect the core of the research.

3.1 Compute intensive data processing

Processing chains have as property that tasks flow through the processing steps, with a fixed order between the processing steps. A connection between two steps indicates that the producing processing step provides the input for the consuming processing step. As there is no intermediate entity that receives the complete task, the two processing steps must be compatible regarding the output of the producing

step and the input of the consuming step. Each processing step acts as consuming processing step to receive new tasks, as well as acting as producing processing step for successfully processed tasks in that processing step. However, the first processing step in the chain only produces tasks and receive or retrieve tasks from outside the processing chain. As well as, the last step in the processing chain only consumes tasks and distribute the results outside of the processing chain.

Each processing step can be parallelised individually, so there is no fixed connection between an instance of the consuming processing step and an instance of the producing processing step. For each task the destination of the task can change to another instance of the receiving processing step. With these requirements for the processing chains, a more extensive illustration of the kind of processing chains discussed in this research can be made. In Figure 3.1, an example of the flow of tasks through the processing chain is shown. With three processing steps linked sequentially in a processing chain, the first and the last processing step have three instances and the second processing step has two instances. Eight tasks are introduced at *Processing Step A* that are being processed by the instances before the intermediate result is sent to *Processing Step B*. The result of processing in *Processing Step B* becomes then the input for the *Processing Step C*. The tasks remain the same throughout the flow, only the data they are carrying changes at each processing step. So the blue *Task 1* is the same task as the purple *Task 1*, only the data carried by the task changes.

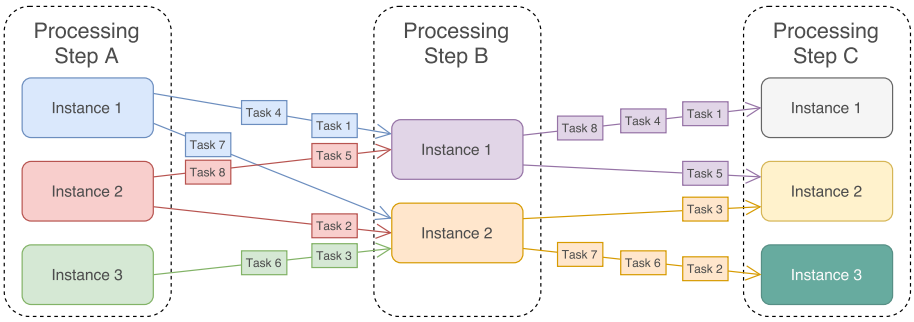


Figure 3.1: Flow of tasks through a processing chain

To achieve a efficient flow of tasks through the processing chain it is required that each processing step handles roughly equal amounts of tasks per unit of time. As in this case the tasks do not pile up in front of slower processing steps, and faster processing steps are not idle too long. Instances of processing steps are isolated entities that do not depend on other instances in that processing step, therefore increasing the amount of instances should result in a linear increase of potential task throughput of the processing step. To be able to reason on the throughput of processing steps, we assume that processing a task by processing step instances takes in general the same amount of time for all the instances of that processing step. When at a certain point in time a processing step handles 100 tasks per minute with 10 instances, we assume that 20 instances of the same processing step are capable of handling 200 tasks per minute.

To identify the processing steps that are under- or overutilised, information is needed about the flow of tasks in the processing chain. When an overview of the current state of the processing chain is made, with different kinds of information, important decisions should be made for future time steps to try to create a more efficient distribution of processing step instances.

To make sure that the scale of the different resource types support the distribution of processing step instances, another set of information is needed about the utilisation of the different types of resources. With the information on the different types of resources, decisions can be made if the current distribution of resources corresponds with the distribution of processing step instances.

When looking at the Toyota example from 1, lets assume the following partial scenario:

- 50 body processing instances
- 10 painting processing instances
- 60 welding machines
- 18 paint guns
- Body processing takes 3 hours per task per instance
- Painting takes 0.5 hour per task per instance

Lets start by analysing the body processing step. Each body processing instance needs one welding machine to successfully process a task. So the resource capacity of 60 welding machines for 50 body processing instances is $\frac{60}{50} \cdot 100\% = 120\%$, so at all times there are 10 welding machines not in use by any body processing instance. For the painting processing step, each instance needs 2 paint guns to successfully process a task. The resource capacity in this case is $\frac{18}{10 \cdot 2} \cdot 100\% = 90\%$, which means that always one painting processing instance is idle and not capable of handling new tasks. For both processing steps the resource capacity is not optimal, as the aim is to have 100% utilisation of resources.

When looking at the relation between the two processing instances it can be determined how many tasks the body processing step and the painting processing step can handle per hour. The body processing step is able to handle $\frac{50}{3} = 16\frac{2}{3}$ tasks per hour, and the painting processing step is able to handle $\frac{9}{0.5} = 18$ tasks per hour. The efficiency of the relation is $16\frac{2}{3}/18 = \frac{25}{27} = 0.93$, so the rate of tasks produced by the body processing step is only capable of utilising 93% of the painting processing instances, not including the idle painting processing instance that has no paint guns to use. To achieve an optimal distribution of processing instances the ratio between the body processing step and the painting processing step should be $50 : 9 \cdot \frac{25}{27} = 50 : 8\frac{1}{3} = 6 : 1$, which corresponds to the ratio of time needed to process one task per instance. To level the current ratio of processing step instances two options are available: increasing the amount of body processing instances, and

decreasing the amount of painting processing instances. For the resource type distribution, there is no efficient usage of resources for both processing steps. Adjusting the resources for the current situation is relatively easy, as removing 10 welding machines and adding 2 paint guns result in an possibility of optimal resource usage when all the processing step instances have tasks to process. But the interesting fact is estimating the resource type distribution for future time steps, as the processing step distribution may change over time. For example, due to changes in input data that requires more or different calculations in one of the processing steps, leading to a changed demand from the processing chain. So interaction is needed between both distributions to be able to result in an efficient processing chain.

In this example all of the information was clear and available, but the interesting case is when this information is not yet available. Retrieving the information from each processing instance individually is a costly procedure, especially when the amount of instances grows to thousands or more. The kind of metrics that need to be gathered to create a clear overview of the system is discussed in the next section. The sections after that explain how these metrics can be used to determine the correct distributions.

3.2 Metrics gathering

To be able to identify which metrics are needed to deduce interesting metrics for the decision, first needed are the useful types of information for both the processing chain as well as the resource type utilisation.

For the processing chain it is important to know how the flow of tasks is in the chain. Intuitively, the throughput of tasks at each processing step is a good indicator of the performance of a processing chain, as this information could indicate processing steps that are under- or over-performing. However, when the buffer of tasks in-between processing tasks reaches it's maximum capacity this buffer will limit the throughput of the producing processing step. This would be the case when the producing processing step process tasks at a higher rate than the consuming processing step. But also, when the consuming processing step is capable of handling more tasks than the producing processing step is producing, the throughput of the consuming processing step will be capped at the throughput of the producing processing step.

Therefore, more information specific for the consuming processing steps and the producing processing steps is needed to have a clear overview of the processing pipeline. For producing processing steps it could be advantageous to look at the overall buffer between the producing processing step and the consuming processing step. When these buffers contain low amounts of tasks for a longer time period, it indicates that tasks are consumed almost instantly. Resulting in two possibilities, either the consuming processing steps has an equal throughput of tasks, which is the desirable configuration. But it can also mean that the consuming processing

step has a greater potential throughput and instances are waiting for the producing processing step to process tasks. Also looking at the derivation of the buffer usage can indicate the trend of the buffer usage, which can be used to identify the difference in throughputs of the two subsequent processing steps when the buffer is not nearly empty or nearly at full capacity. To prevent gathering of metrics regularly at a certain interval, the producing step can indicate for each task the fraction of time it has waited due to a full local buffer. Ideally this would approach zero, indicating that the task did not wait to be placed in the local buffer. When this value increases it indicates the buffer is full, and there are too much producers for the amount of consumers. This metric, however, is only capable of indicating that the local buffer is actually full but is not able to indicate whether or not the the available buffer capacity is increasing or decreasing. For this, the relation between processed tasks per time unit and the increase or decrease of the local buffer can be investigated. For instance, when the processing step is processing 100 tasks a minute, but the local buffers increase with 25 tasks a minute. This indicates that the local buffers eventually fill up, to prevent waiting for this limit to be reached this relation can be used to predict the correct ratio between producers and consumers. This metric can be described as the relative delta of local buffers, as the delta of the local buffers indicate the change in the availability of buffer capacity and this delta is expressed relative to the amount of tasks processed in the delta window.

Consuming processing steps are either limited to the amount of tasks it is able to consume from the producing processing step, or limited to its own potential capacity. For the first case, not all consuming step instances are used fully. This can be measured by recording the average time a processing step instances waits before it is able to receive a new task. For instance, when a processing step instance is waiting on average 2 minutes for a task and processes a task on average for 4 minutes, the utilisation of that processing step is $\frac{4}{2+4} = \frac{2}{3}$. Such an utilisation indication gives a very good starting point to calculate the necessary ratio of instances of both processing steps.

Also for the utilisation of resource types additional information is needed. However, it depends on the type of resources what is needed. When the usage of a resource instance is binary, i.e. the resource instance is either used fully or not at all, an estimation of the utilisation of the group of instances of that resource type can be made by calculating the percentage of time the resource instances are used. So, for instance, a paint gun is either used to paint one car or it is not, it is not partially used for one car and partially for another car at the same time. When 10 paint guns have a total utilisation of 6 hours in a 1 hour window, it means that the paint guns are idle for 4 hours. Resulting in an utilisation factor of $\frac{6}{10} = \frac{3}{5}$.

Another possibility is that one resource instance is capable of handling multiple tasks in parallel, in that case the utilisation of the resource instance is already a fraction. Namely, the fraction of processing steps using the resource versus the potential capacity of that resource instance. In that case, aggregating the utilisation of the group of resource instances can be simply done by, for instance, averaging the

utilisations.

Because resources and processing step instances are not likely to be created instantly and without costs, information is needed on the time it takes to start processing step instances. This delay introduced in the creation of new instances has implications on the desired window metrics should be gathered and the decision making. When the delay of creating new instances is large with respect to the total running time of the system, we need to be very sure we need that extra instance. But when the delay is small it can be advantageous to start new instances with less certainty, as new instances that turn out to be under-utilised can be removed without large costs to the system.

Summarising, the following metrics would be needed to be able to reason about the flow of tasks through the processing chain and the utilisation of resources by that processing chain are shown in Table 3.1.

Processing chain metrics

Metric code	Description
PublishWait	Fraction of wait times to publish tasks in the local buffer of producing processing steps, relative to the processing time of the task.
BufferChange	The fraction of the amount of tasks being published to the local buffer versus the amount of tasks transferred to consumers.
ConsumeWait	Fraction of wait times versus processing times of consuming processing steps.

Resource utilisation metrics

Metric code	Description
Utilisation	Fraction of utilisation, calculated by resource specific metrics.

Table 3.1: Metrics definitions

3.3 Processing chain distribution

With the processing chain metrics, imbalances in the distribution of instances of processing steps can be identified. To be able to do this specific states are identified between subsequent processing steps. These three possible states are: fast producer and slow consumer, slow producer and fast consumer, and balanced consumer and producer. Where a fast producer and slow consumer is a state where a processing step that with the current amount of instances is able to processes more tasks than

the consuming processing step is able to handle with its current amount of instances. For the slow producer and fast consumer, this logically means that the producing step processes tasks at a lower rate than the consuming processing step is able to handle. And the balanced consumer and producer is the desired state, that is tried to achieve for each pair of subsequent processing steps in the processing chain.

For each pair of subsequent processing steps we have a set of metrics as defined before, with these metrics we can identify in which category the pair of steps belong to. In Chapter 4 solving the question on how the distribution should be determined is answered.

Mapping the states and the metric characteristics gives the following overview:

- Fast producer and slow consumer

  PublishWait $> 0 \ \wedge$ **BufferChange** $\approx 1 \ \wedge$ **ConsumeWait** ≈ 0
  The producer has a significant wait time for each tasks to be placed in the local buffer. Therefore, the relative buffer delta is zero, as the producer is not producing more tasks than there are tasks consumed from the local buffer. The consumer can in this case only be fully occupied with consuming tasks, otherwise there could be a fault in transferring tasks from the consumer to the producer.

  PublishWait $\approx 0 \ \wedge$ **BufferChange** $> 1 \ \wedge$ **ConsumeWait** ≈ 0
  The producer is producing more tasks than are consumed, with a local buffer that is not yet full. As the wait times are still approximately zero, but the relative buffer delta indicates an increase in the buffer usage. Just like before, this means that the consumer should have no wait times, as the buffer is increasing.

- Slow producer and fast consumer

  PublishWait $\approx 0 \ \wedge$ **BufferChange** $\approx 1 \ \wedge$ **ConsumeWait** > 0
  The consumer is waiting for tasks to be published, combined with approximately zero wait times of the producer and an even relative buffer delta this indicates that the local buffer of the producer is empty and when new tasks are published they are consumed directly by an available consumer.

  PublishWait $\approx 0 \ \wedge$ **BufferChange** $< 1 \ \wedge$ **ConsumeWait** ≈ 0
  The consumer is fully occupied with consuming tasks from the publisher, but the local buffer of the producer is decreasing. Indicating that in a certain amount of time the local buffer of the producer will be empty, resulting in a state where consumers are waiting for tasks.

- Balanced producer and consumer

  PublishWait $\approx 0 \ \wedge$ **BufferChange** $\approx 1 \ \wedge$ **ConsumeWait** ≈ 0
  This is the ideal situation, both the producer and consumer are not waiting on each other and the buffer remains roughly the same, indicating

that both processing steps are processing tasks at the same rate.

		BufferChange		
PublishWait	**ConsumeWait**	< 1	≈ 1	> 1
≈ 0	≈ 0	SP-FC	Balanced	FP-SC
≈ 0	> 0	-	SP-FS	-
> 0	≈ 0	-	FP-SC	-
> 0	> 0	-	-	-

Table 3.2: Producer-Consumer categorisation, indicating in which category the producer and consumer belong to. With the following possibilities: Fast producer slow consumer (FP-SC), Slow producer fast consumer (SP-FC), balanced producer and consumer (Balanced), and the erroneous category (-)

The combinations of metrics that are not described by the above overview indicate a faulty state, as there is a relation between the metrics. For instance, not both the producer and the consumer can wait half of the time, as for the producer to wait the local buffer should be full and for the consumer to wait the local buffer should be empty. In Table 3.2 all the possible values of the metrics with their corresponding category are shown.

3.4 Resource type distribution

Making decisions on the distribution of processing step instances is only part of the question that is solved. Matching the available resources with the processing chain is an important factor in achieving an overall efficient usage of resources with respect to the processing chain.

Determining an efficient resource distribution at a specific time is relatively straightforward, as we have the resource utilisation metrics for the different resource types. This can simply be done by multiplying the current amount of resource instances by the utilisation metric. For instance, when there are 100 welding machines with an overall utilisation of 0.8 and 30 paint guns with an overall utilisation of 0.9, the ratio between the two resource types should be: $(100 \cdot 0.8) : (30 \cdot 0.9) = 80 : 27 = 1 : 0.3375$.

However, when processing step instances are competing for a resource that is utilised fully there will be processing step instances that have to wait before using a resource. The waiting on resources cannot be covered by determining the processing step distribution solely. Take for instance the case where there are 100 welding machines with a utilisation of 1.0, this could happen when 100 processing step instances use each one welding machine. But when there are 120 welding machines each requiring one welding machine, 20 processing steps instances will be halted at each moment as there is no resource available to fulfil tasks. Or another possibility is that, when resources can be shared by multiple processing step instances, the performance of

the processing step instances decreases. Resulting in disturbing the processing step distribution, as when resources are not fully occupied the processing step instances will increase their performance.

Therefore, interaction is needed between the determinations of both the resource type distribution and the the processing step distribution. To avoid exhaustive usage of resources by processing step instances and to create efficient and sustainable distributions that the processing chain is able to use. As when a resource is exhausted, the resource will most likely throttle the processing step instances, resulting in lower throughput of the instances which results in wrong distributions when the is scaled up. As the instances will return to their actual throughput in that case, and the deployed distribution is in-balanced.

Chapter 4

Solution

In this chapter solutions on the problems stated in Chapter 3 are discussed. Focusing on the theoretical approaches on the calculations of the different distributions.

The chapter is divided into three main sections, firstly to calculate the optimal processing chain distribution based on the metrics gathered from the processing chain. With this distribution the translation to the actual instance count of the different processing steps has to be determined to fully utilise the resource available to the cluster at that moment. And lastly, calculating the distribution of resources in the cluster and resource instance counts is described based on user defined strategies.

An worked out example, based on the Toyota example, clarifies the solution.

4.1 Processing chain distribution

In Section 3.3 a categorisation of pairs of processing steps is made to determine if the relation between the processing steps is: fast producer and slow consumer, slow producer and fast consumer, or balanced producer and consumer.

When the processing steps are categorised into the different states, the determination of the ratio between the steps should occur. For this the currently deployed ratio between the processing steps is needed, provided by equation 4.1, where n is the amount of instances for either the producing processing state or the consuming processing state.

$$r = n_{producing} : n_{consuming} \tag{4.1}$$

The calculation of the new ratio depends on the category the pair of processing steps falls in, as well as the **BufferChange**. The category determines if there should be more or less publishers relative to consumers, and the buffer change indicates whether or not the buffer is full or empty. If the buffer is not at it's full or empty level, the change in the buffer can be used to determine the optimal ratio between processing steps. The **BufferChange** is 2 when the producers process 2 times more

tasks than the consumers is consuming and is 0.5 when the consumers consume tasks 2 times faster than the producer is producing. Multiplying the amount of consumer by the **BufferChange** results in an increase of workers when the producer is faster and a decrease of workers when the consumer is faster, which is the goal of calculating the ratios between processing steps.

When the buffer full, either the the producer has a significant wait percentage to publish tasks in its buffer or the consumer is exactly on par with the production of the producer. In the case the producer is waiting, it makes sense to lower the amount of producer instances with the percentage that the producers are waiting on average. The same holds up for the case when the buffer is empty and consumers are waiting significantly for new tasks, in that case the consumer instances should be lowered.

In the equations 4.2-4.5 this description is translated into formulas that can be used in algorithms later on.

$$\text{Fast producer} \ \wedge \ \delta \approx 1: \quad r' = n_{producing} \cdot (1 - \ pw \ ) : n_{consuming} \tag{4.2}$$

$$\text{Fast producer} \ \wedge \ \delta > 1: \quad r' = n_{producing} : n_{consuming} \cdot \ \delta \tag{4.3}$$

$$\text{Slow producer} \ \wedge \ \delta \approx 1: \quad r' = n_{producing} : n_{consuming} \cdot (1 - \ cw \ ) \tag{4.4}$$

$$\text{Slow producer} \ \wedge \ \delta < 1: \quad r' = n_{producing} : n_{consuming} \cdot \ \delta \tag{4.5}$$

$$\text{Balanced} \ \wedge \ \delta < 1: \quad r' = n_{producing} : n_{consuming} \tag{4.6}$$

where $\delta = \textbf{BufferChange}$, $pw = \textbf{PublishWait}$, $cw = \textbf{ConsumeWait}$

With the new set of ratios for the processing chain, the distribution can be calculated with an recursive function displayed in equation 4.7.

$$D_n = D_{n-1} \cdot r' \tag{4.7}$$

$$D_0 = 1 \tag{4.8}$$

For each processing step in the processing chain the position in the chain determines which D belongs to that processing step. The first processing step has index 1 and the last processing step has index N. For clarification D is normalised against the sum of D, following equation 4.9

$$\hat{\mathbf{D}} = \frac{\mathbf{D}}{|\mathbf{D}|} \tag{4.9}$$

The normalised distribution $\hat{D}$ now contains the fraction of instances a processing step needs with respect to the total amount of instances.

4.2 Processing instance counts

With the normalised distribution $\hat{\mathbf{D}}$ calculated, this can be used to determine the amount of instances each processing step that will improve the resource utilisation. For this, the resource utilisation metric **Utilisation** is needed. This determines the capability of each resource to be used by extra processing step instances. To be able to use the utilisation in formulas a function is introduced in equation 4.10 that returns the fraction of utilisation of an input processing step, which corresponds with the utilisation of the resource that is needed for that processing step.

$$U(p) = \text{utilisation of resource needed for } \ p \tag{4.10}$$

This function is used to determine the scaling factor to result in the total amount of instances that at least can run on the specific resource, by dividing the current amount of instances by the utilisation fraction. As the current amount of instances do not produce more load than the current utilisation fraction, scaling the amount of instances by 1 over the utilisation the instances cannot produce more than the current load. As processing steps that have an calculated distribution fraction that is higher than the current deployed distribution fraction are used fully, otherwise the scaling up wouldn't have effect at all. Equation 4.11 shows the calculation of the scale factor, which is determined for all processing steps that have an higher distribution factor than currently deployed.

$$\begin{aligned} s_p &= \frac{1}{U(p)} \\ s &= \min_{\forall p} s_p \\ p_{scale} &= \operatorname*{argmin}_{\forall p} s_p \end{aligned} \tag{4.11}$$

As all of the processing steps are scaled based upon the same scale factor the lowest scale factor should be used, as this is the scale factor that leads to no overutilisation of the processing steps. With these factors calculated, the new processing step instance count can be determined. Following equation 4.12, the new set of processing step instance counts can be calculated.

$$\begin{aligned} \mathbf{N}'_{p_{scale}} &= \mathbf{N}_{p_{scale}} \cdot s \\ \mathbf{N}' &= \mathbf{N}'_{p_{scale}} \cdot \frac{\hat{D}}{D'_{p_{scale}}} \\ \text{where } \ \mathbf{N}_p &= \#\ \text{ current instances of step } \ p \end{aligned} \tag{4.12}$$

This set $\mathbf{N}'$ can now be used to scale the processing step instance. As $\mathbf{N}'$ contains fractions, the set must most likely be converted to a set of integers by either rounding, ceiling, or flooring the fractions, depending on the characteristics of processing steps instances and the kind of resources used in the processing chain.

4.3 Resource provisioning

At the moment the processing step instance count is stable, meaning that the resource usage will be fairly constant, the optimal resource distribution can be determined. By multiplying the amount of resource instances by the utilisation factor, the optimal amount of instances for the current situation can be calculated. As the number of processing step instances $\mathbf{N}'$ will not use more than 100% of the resource, no resources is overutilised at this moment. Like the calculation of the normalised distribution of processing step instances $\hat{\mathbf{D}}$, for the resource distribution a normalised distribution is used to make the calculations using the distribution easier, as shown in equation 4.13.

$$\begin{aligned} \mathbf{R}'_r &= \mathbf{R}_r \cdot \mathbf{U}_r \\ \hat{\mathbf{R}} &= \frac{\mathbf{R}'}{|\mathbf{R}'|} \end{aligned} \tag{4.13}$$

where $\mathbf{R}_r = \#$ current instances of resource r, $\mathbf{U}_r =$ utilisation of resource r

To be able to know how much the resource instances should be scaled up or down, a budget B of the maximal resource allocation is needed. Calculating the amount of resource instances that is needed to create an even resource utilisation is a simple task, as shown in equation 4.14.

$$\mathbf{R}'' = \hat{\mathbf{R}} \cdot B \tag{4.14}$$

This new optimal resource instance count $\mathbf{R}''$ is fractional, as resources are most likely scaled based on integers, the fractions should be converted to corresponding integers.

4.4 Example

To clarify the formulas above, a scenario is worked out based on the Toyata example of Chapter 1. Where four processing steps are distinguished: Dealer, Body processing, Painting, Assembly. In Table 4.1, the metrics of the processing steps at a certain point in time are shown.

	Instances	**PublishWait**	**BufferChange**	**ConsumeWait**
Dealer	1	0.75	≈ 1	0.25
Body processing	20	0.1	≈ 1	≈ 0
Painting	5	≈ 0	1.2	≈ 0
Assembly	10	≈ 0	≈ 1	≈ 0

Table 4.1: Scenario metrics

Each of the processing steps use a distinct resource, so four utilisation metrics are shown in equation 4.15, with their corresponding scale factors.

$$\begin{aligned} U(Dealer) &= 0.1 & p_{Dealer} &= 10 \\ U(Body\ processing) &= 0.4 & p_{Body\ processing} &= 2.5 \\ U(Painting) &= 0.2 & p_{Painting} &= 5 \\ U(Assembly) &= 0.3 & p_{Assembly} &= 3.33 \end{aligned} \tag{4.15}$$

The initial ratios are determined from the number of instances from Table 4.1, as shown in equation 4.16.

$$\begin{aligned} r_1 &= \text{Dealer} \rightarrow \text{Body processing} &&= 1:20 \\ r_2 &= \text{Body processing} \rightarrow \text{Painting} &&= 20:5 \\ r_3 &= \text{Painting} \rightarrow \text{Assembly} &&= 5:10 \\ r_4 &= \text{Assembly} \rightarrow \text{Dealer} &&= 10:1 \end{aligned} \tag{4.16}$$

Now, given the initial ratios and the metrics at a certain point in time, the new processing step distribution can be calculated. All the ratios are in the form of $1:n$, so that in the following calculations these numbers can be used more easily. Equation 4.17 shows these calculations and their results.

$$\begin{aligned} r'_1 &= 1 \cdot (1 - 0.75) : 20 &&= 0.25 : 20 &&= 1 : 80 \\ r'_2 &= 20 \cdot (1 - 0.1) : 5 &&= 18 : 5 &&= 1 : 0.2778 \\ r'_3 &= 5 : 10 \cdot 1.2 &&= 5 : 12 &&= 1 : 0.4167 \\ r'_4 &= 10 : 1 &&&&= 1 : 0.1 \end{aligned} \tag{4.17}$$

Now the distribution of processing steps can be determined, as shown in equation 4.18, where the values of D'_n correspond with the consumers of each pair of processing steps. So the value of D'_1 corresponds to the body processing instances, and D'_2 corresponds to the painting instances. The last value, D'_4 corresponds to the consumers of the first processing step, namely the dealer.

$$\begin{aligned} D'_1 &= 1 \cdot r'_1 &&= 80 \\ D'_2 &= D'_1 \cdot r'_2 &&= 22.222 \\ D'_3 &= D'_2 \cdot r'_3 &&= 9.259 \\ D'_4 &= D'_3 \cdot r'_4 &&= 0.926 \end{aligned} \tag{4.18}$$

Normalising this distribution gives the fractions of each processing step, as shown in equation 4.19, which sum up to 1. Which indicates that in this case 71.17% of all the processing step instances should be of the body processing type.

$$
\begin{aligned}
\hat{D}_1 &= 1 \cdot r'_1 &&= 0.7117 \\
\hat{D}_2 &= D'_1 \cdot r'_2 &&= 0.1976 \\
\hat{D}_3 &= D'_2 \cdot r'_3 &&= 0.0824 \\
\hat{D}_4 &= D'_3 \cdot r'_4 &&= 0.0082
\end{aligned}
\tag{4.19}
$$

Now that we have the new distribution, the processing step instance count can be determined. First the starting point for the scaling has to be determined, by looking at the resource that is utilised the most. Which is the resource belonging to the body processing step, as shown in equation 4.20

$$
\begin{aligned}
s &= \min_{\forall p} s_p &&= 2.5 \\
p_{scale} &= \operatorname*{argmin}_{\forall p} s_p &&= \textit{Body processing}
\end{aligned}
\tag{4.20}
$$

Filling in equation 4.12 with the calculated values gives the new processing step instance distribution. In equation 4.21, the values of the desired instance counts are calculated.

$$
\begin{aligned}
\mathbf{N}'_{p_{scale}} &= \mathbf{N}_{p_{scale}} \cdot p_{scale} &&= 20 \cdot 2.5 = 50 \\
\mathbf{N}' &= \mathbf{N}'_{p_{scale}} \cdot \frac{\hat{D}}{D'_{p_{scale}}} &&= 50 \cdot \frac{\{0.0082,\ 0.7117,\ 0.1976,\ 0.0824\}}{0.7117} \\
& && = \{0.579,\ 50,\ 13.883,\ 5.789\}
\end{aligned}
\tag{4.21}
$$

As these numbers are fractional and it is most likely that we can not use half a paint gun, these values are rounded up to the next integer. Table 4.2 shows the number of processing step instances that should be provisioned.

	# instances
Dealer	1
Body processing	50
Painting	14
Assembly	6

Table 4.2: New instance counts

When the number of instances for each processing steps stops changing the resources can be scaled accordingly, by looking at the utilisation fractions and using equation 4.13 and 4.14

Chapter 5

Architecture

This section proposes a conceptual architecture based on the solution introduced in Chapter 4. This conceptual architecture provides the basis for the implementation to verify the solution. The high-level architecture is the starting point for the rest of this section. Some individual aspects of the architecture are discussed in more detail to clarify relevant parts of the architecture with respect to the adaptive approach.

The focus of the architecture lays on the case where processing chains are chains of algorithms and models that are run on compute instances that act as resources for the processing chain.

This chapter starts with the introduction of a set of high-level requirements that provides the basis for the architecture. Based on these requirements a high-level architecture is discussed, of which several parts are elaborated upon in later sections.

5.1 Requirements

Based on the research question and the solution, a set of high-level requirements can be deduced. The high-level requirements are split up into functional requirements, i.e. what the solution is supposed to do, and non-functional requirements, what quality attributes the solution must fulfil. The requirements are prioritised following the MoSCoW method. Providing the following four categories:

Must Requirements that are critical for the final deliverable to be successful

Should Requirements that are not essential for the final deliverable to be successful. These requirements can be as important as MUST requirements, but often do not have the necessity to be included in the first deliverable

Could Requirements that are nice to have when there are enough time and resources left. Often improving user experience.

Won't Requirements that have are the least critical

5.1.1 Functional Requirements

In Table 5.1 the functional requirements are displayed, with their identifier, priority, title, and description.

FR-1	**Must**	**Adaptive provisioning**
	The platform must be able to estimate the needed resources demand of running applications over time and dynamically provision that estimate.	
FR-2	**Must**	**Gather processing chain metrics**
	The platform must be able to gather metrics of processing chains to be able to make a decision for FR-1, by estimating the processing steps that are bottlenecks in the processing chain. Based on the processing chain metrics discussed in Section 3.2.	
FR-3	**Must**	**Gather resource metrics**
	The platform must be able to gather resource metrics from the cluster to be able to make a decision for FR-1, by estimating what resources are under- or overutilised in the cluster. Based on the resource metrics discussed in Section 3.2.	
FR-4	**Must**	**Language library**
	The platform must provide a scalable language library that developers can use to create user applications. This language library is the needed layer of abstraction for developers to create user applications as intuitively as possible.	
FR-5	**Must**	**Run user applications**
	The platform must be able to run processing chains written by external developers. Processing chains written with the language library (FR-4) should have the right communication protocols and runtime capabilities to be executed on the platform.	
FR-6	**Must**	**Schedule tasks**
	The platform must be able to schedule user applications on the platform. This includes the scheduling of tasks, i.e. partitions of the input data that flow through the processing chain, on the available processing step instances.	
FR-7	**Could**	**User Interface**

		The platform should provide an user interface in which users are informed about decisions made by the platform. Also, the platform should provide statistics of running and finished processing chains.
FR-8	**Won't**	**Data locality**
		The platform could provide a way to schedule tasks closely to where the input data resides. So that data-heavy processing chains can be executed on the platform, with limited performance losses. Based on the scope of this research, this requirement won't be embedded in the architecture, leaving this for future work.
FR-9	**Won't**	**Flow optimisations**
		The platform won't optimise processing chains, for instance by combining processing steps to limit the necessity of transferring data.

Table 5.1: Functional Requirements Requirements

Requirements FR-1, FR-2, and FR-3 are introduced based on the research question as stated in the introduction, following the three sub-questions. To be able to have a meaningful reference framework as it is intended to use, by having processing chains that are dynamic and diverse, the requirements FR-4, FR-5, and FR-7 provide the ability to create other applications that are run on the platform easily.

Scheduling of tasks is an implicit requirement, as due to the distributed nature of the framework, scheduling (FR-6) plays an important role in the performance of the framework.

Requirements FR-8 and FR-9 are added based on the related work in the field of distributed data processing. But are left out of the scope of this research due to time limitations.

5.1.2 Non-functional Requirements

In Table 5.2 the non-functional requirements are displayed, with their identifier, priority, title, and description.

NFR-1	**Must**	**Distributed**
		The platform must be able to execute user application distributed on a cluster of resources.
NFR-2	**Must**	**Scalability**

		The platform must be designed with scalability in mind, the overhead introduced when more resources are added should be minimised. So that processing chains that are required to have low lead times are able to provision a large amount of resources that are used effectively. Furthermore, the platform must be able to handle at least hundreds of resource units.
NFR-3	**Must**	**Fault tolerance**
		The platform must be able to handle faulty processing steps, resource malfunction, or packet loss.
NFR-4	**Should**	**Adaptability**
		The platform should be designed in such a way that new and other techniques that replace existing functionalities can be added to the platform easily. Like, for instance, adding other resource providers or resource types.

Table 5.2: Non-functional Requirements Requirements

The chosen non-functional requirements are common for these types of frameworks. They are present because insufficiently meeting these requirements limits the ability of reasoning on the results of the framework.

5.2 High-level architecture

The high-level architecture of the platform gives an overview of the components in the platform and the relations between these components. The logical centre of the platform is the monitor, which monitors running processing chains and make decisions for provisioning processing step instances as well as resource instances. The monitor has no direct notion of the resources and processing step instances, separate managers provide a layer of abstraction for these components. Thus, letting these managers take care of the low-level matching between processing step instances and resources, and letting the monitor take decisions on the global overview of the processing chains. Providing the opportunity to change resource managers or processing step managers to other vendors.

The platform consists of four components that each take care of a part of the solution:

Monitor
: The monitor is responsible for the deployment of processing chains by providing the resource manager and the processing step instance manager enough information, so that they are capable to provision the cluster with the right

processing step instances. This is an ongoing process that monitors the progress and other metrics of the processing chain to see if changes should be made in the amount of resources or amount of processing step instances used by the processing chain. Therefore, the monitor can be seen as an controller of a control loop that receives sensor output from the resource manager and the resource manager, and controls the amount of instances of processing steps and resource types. The monitor is responsible for FR-5, and for the provisioning decisions for FR-1.

Resource manager
The resource manager is responsible for providing resources to processing step instances, enabling processing step instances to process data. Metrics of the usage of resources are shared with the monitor so that it is able to make the decision to scale up or scale down the number of resources. Making the resource manager responsible for FR-3 and partly for FR-1.

Processing step manager
The processing step manager is responsible for instantiating processing step instances of the right type for processing chain. Also the communication between processing step instances is taken care of by the processing step manager. It has a notion of the available resources to the pipeline, so that it can match processing step instances to resource instances. Thus, making it responsible for FR-2 and partly for FR-1.

User interface
The user interface component provides users the ability to start new processing chains and to monitor their running processing chains to see the progress and the decisions made by the monitor. Giving the user the ability to watch the performance of his processing chain. The user interface communicates only with the monitor, making it only responsible for FR-7.

These components are shown in a schematic overview of the platform in Figure 5.1, where the relations between the components are illustrated.

Giving this four main components of the platform, only FR-4 and FR-6 are not accounted for yet. The architecture supporting these requirements is discussed later in this chapter.

To create a separation of concerns the monitor is not responsible for all of the tasks, but the resource manager and the processing step instance manager are each responsible for their own part of the system. The monitor is the component that takes decisions retrieved from both the resource manager and the processing step manager, leaving the logic on how to provision resources and processing steps to their managers.

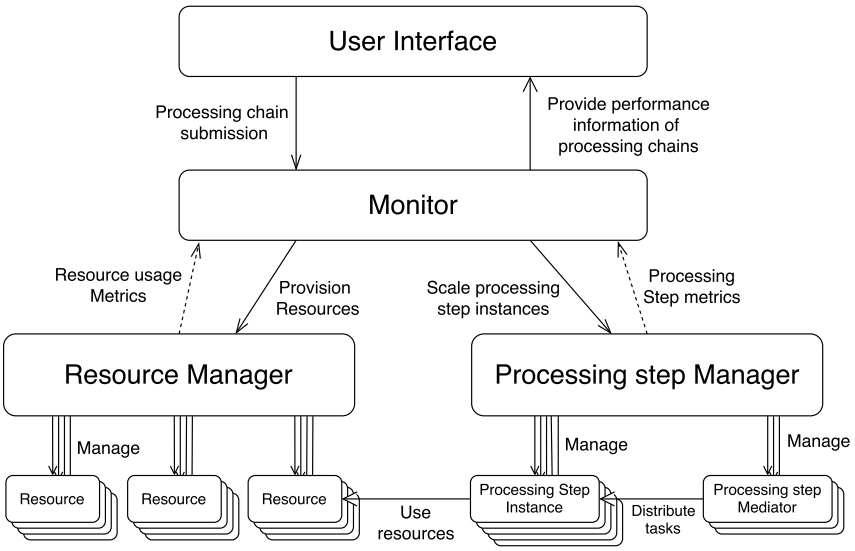


Figure 5.1: High-level overview

5.3 Language library

The language library is the tool developers use to create processing chains, consisting of individual processing steps that can process tasks in parallel. The essence of the language library is that it has to be easy to use without large complex configurations, the developer should only worry about the pragmatics of the processing chain. To support FR-4, which requires a clear and simple abstraction for developers to create applications that can be run on the platform.

The MapReduce principle is very convenient to use for this types of problems, as the different functions can be executed in parallel easily. As, for instance, the map function executes a function on a partition of data independently of other partitions. This stateless design of the MapReduce principle, therefore, is well suited for executing tasks on a cluster of resources.

As the normal MapReduce functions do not take into account heterogeneous resources to run tasks on, the principle is extended with possibilities for developers to indicate on which resource a function is required to run. Making it necessary to be able to run the language library on all resource types that ought to be supported in the processing chains.

The language library contains the runtime requirements, so that when a processing chain is written by a developer it can be packaged so that the processing chain can be used in runtime. Implementing logic that enables the packaged applications to transfer tasks between processing step instances.

5.4 Runtime overview

The runtime environment consist of 3 major components: the platform monitor, the processing step instances, and the processing chain mediators. The platform monitor orchestrates processing chains, by starting and monitoring processing chains initiated by users. Also, the dynamic provisioning is provided by the platform monitor, as this component gathers metrics of the resources and the processing step instances. The architecture of the run-time provides a solution for FR-6, also the run-time architecture provides information on how the processing chain metrics are gathered.

Processing chains need processing step instances to execute functions on input data, for managing the distribution of tasks over the processing step instances processing chain mediators are introduced. These mediators are responsible for distributing tasks it receives to processing step instances for the next processing step. To prevent the mediator to become a bottleneck when all data travels through the mediators, only task descriptions are shared with the mediators. These task descriptions are small packages indicating that the processing step that published the task description has a task available. Processing step instances are then themselves responsible to transfer the data between each other. The reason the mediators are situated between the processing step instances is because the platform monitor is required to know how tasks are travelling through the system so that it has a view of the running processing chain, used to make provisioning decisions. The mediators share the knowledge they have about the processing chain section they are responsible for. In Figure 5.2 the communication steps between processing step instances and mediators is shown. The dotted lines indicate that the amount of network traffic used is small compared to the solid line. As these messages are only the requests or the tasks metadata.

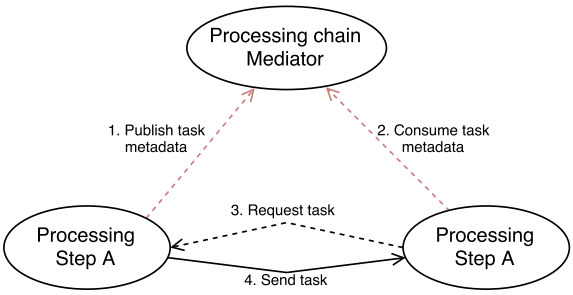
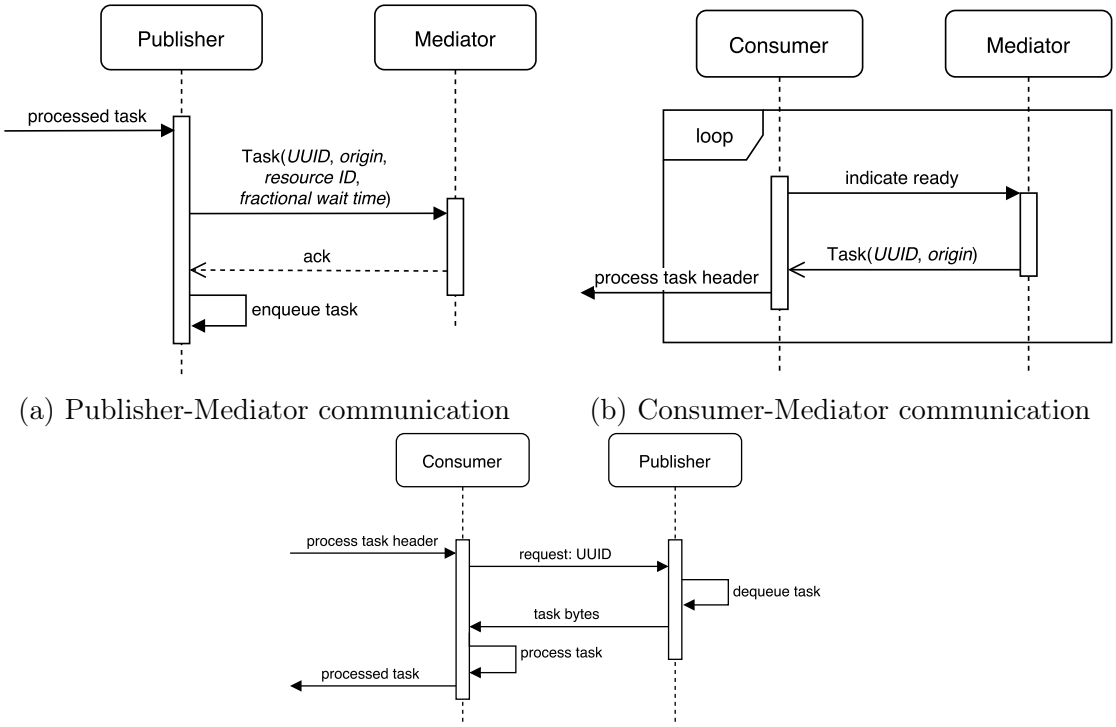


Figure 5.2: Runtime task distributing

The information the monitor gathers from the mediators is also used to inform users of the status of their running processing chain. For instance, information about the progress of the processing chain or information on the utilisation of the processing step instances for each processing step. So that users can be informed if the processing chain is in a stable state, where there is no or little changes in the distributions.

Figure 5.3 shows the sequence diagrams for the communication between publisher-mediator-consumer. Indicating the communication protocol between these entities and which entity is responsible for which part of the communication.



(a) Publisher-Mediator communication

(b) Consumer-Mediator communication

(c) Consumer-Publisher communication

Figure 5.3: Task processing and distribution sequence diagrams

In these sequence diagrams it becomes clear which components in the platform are responsible for the task distribution. Where the mediator plays an important role in matching publishers and consumers, so that they are able to transfer intermediate results in the chain.

5.5 Control loop

The control loop that periodically decides on the distribution of processing steps instances and resource instances is mainly situated at the Monitor. The Monitor controls the number of instances for both the processing steps as the resources, by sharing the calculated instance numbers to respectively the processing step manager and the resource manager. To be able to calculate new instance counts it requires the information discussed in Chapter 4, which is runtime feedback based on the instance counts deployed at that moment.

The resource manager takes a double role in this control loop, as it provides the abstraction for the monitor to provision resources as well as providing utilisation

metrics of the resource to the monitor. The feedback of the processing chain is acquired from the mediators.

The rate the control loops calculates new distributions depends mainly on external factors from resource providers. As the addition of resources can depend heavily on the type of resources that are provisioned. Also, metrics of resources are most likely not available instantaneous. For instance, metrics from virtual machines can have delays of several minutes before they arrive at the Monitor.

5.6 Dynamic provisioning

Dynamic provisioning of processing step instances and resources is one of the key contributions of this research. A part of the processing step instance provisioning is taken care of by the cluster manager, which decides where to start the actual processing step instances. But an important factor of the processing step instance provisioning, the amount of instances that are deployed per processing step still has to be determined. The key determination estimating the ideal ratio of processing step instances for the different types of processing steps in the processing chain. Using the formulas proposed in Chapter 4, the ideal distribution of processing step instances can be determined. As the instances are provisioned based on integers only and not on fractional values, the number of processing steps instances provisioned have to be rounded up to the next integer. Resulting in slightly too high processing step instance counts. However, this effect is not likely be of a high impact, as the resource usage won't be 100% in this case and assuming the instances won't be occupying the resources when they are waiting.

The resource provisioning is partly taken care of by an external resource provider, but again, like the processing step instance provisioning, the estimation on the distribution of resources types has to be made according the formulas proposed in Chapter 4. As most batch-based use-cases tend to have a fixed upper-bound of resource budget, rounding off resource instances to the next integer can result in an violation of the resource budget. Therefore, the resource instances are be rounded off to the nearest integer. In this case there is also a possibility of budget violation. For instance when resource A is estimated at 4.5 instances and resource B at 5.5 instances, rounding off the instances results in 5 respectively 6 instances. Whereas, the budget is 10 instances. Therefore, the last resource for which instances are determined is given the amount of instances that are left in the budget.

Chapter 6

Realisation

This chapter elaborates on the implementation details of the framework used for verifying the adaptive approach, and how the setup is created for performing tests.

Firstly, the language library is discussed, which forms the basis of applications that run on the platform. The runtime-components, i.e. the processing step runtime, the mediators, and the monitor, are discussed separately. As these components are the most important ones for platform that provide the information to make the decisions.

Also this chapter discusses the deployment of the platform on cloud infrastructure via Docker Swarm and Microsoft Azure.

6.1 Language library

The language library supports developers to create applications to run on the platform. This library must be easy to use. With the usage of Scala as programming language the functional idea behind the MapReduce model can be easily implemented, as Scala is an object oriented functional programming language.

In Listing 6.1 an example of an simple application is shown. This example is one of the scenarios that is evaluated in Chapter 7. All the steps, **data**, **dSettlement**, **wTube**, and the return value, are instances of **ProcessingStep**. This ProcessingStep class contains basic functionality on steps, like the *map* operation used in the example. The operations supported by the **ProcessingStep**s is currently limited to *map* and *reduce* operations, but the implementation is done in such a way that this can be easily extended with new functionalities.

Map operations are relatively large, as no complex directed acyclic graph generator that aggregates map operations that can run together is implemented. Ideally, the settlement map operation would be split into separate map operations for generating the input (line 8), executing the external python library (lines 9 and 10), and

retrieving the interesting pieces of the result (line 11). Making each map operations solely responsible for a small part of the total computation.

```
1  object TestScenario1 extends ProcessingChain {
2
3    def chain(): ProcessingStep[_, _] = {
4      val data = new RandomDatasource()
5
6      val dSettlement = data map { case (x, y) =>
7        import sys.process._
8        val input = DSettlement.xml(x)
9        val settlementResult = ("python .//core-wrap//run.py"
10                                #< stringToInputStream(input)).!!
11       (x, y, settlementResult.split(",").last.toFloat)
12     } setEnvironment VirtualMachine(Windows())
13
14     val wTube = dSettlement map { case (x, y, load) =>
15       val input = WTube.xml(load)
16       val pipelineStresses = WTube.calculate(input)
17       (x, y, load, WTube.retrieveMaxVonMises(pipelineStresses))
18     } setEnvironment VirtualMachine(Linux())
19
20     wTube map { case (x, y, load, maxVonMises) =>
21         // Store in permanent storage
22     }
23   }
24 }
```

Listing 6.1: Application example

Together with the chain the data source are the only mandatory implementations a developer has to develop to have an application that is able to run on the platform. The random data source is shown in Listing 6.2, which shows that only the initialisation of the data source has to be implemented. In which the initial values are filled in, and the desired amount of partitions that are generated from the initial values. In this case random numbers between 2.0 and 22.0 are generated, while the amount of iterations can be changed at the time of deployment. The amount of partitions are in this case equal to the amount of iterations, as the calculations take a decent amount of time in the order of half a minute.

```
1  class RandomDatasource(environment: Option[Environment] = None)
2                  extends DataSource[Float](environment = environment) {
3
4    override def initialize(args: Array[String]): Unit = {
5      val size = if (args.length > 0) args(0).toInt else 100000
6
7      initValues = List.range(0, size) map {
8          _ => 2.0f + 20 * math.random.toFloat
9      }
10     partitions = size
11   }
12 }
```

Listing 6.2: Data source example

Following these simple examples, the language library enables developers to create processing easily without having to know how the platform actually works. Only the amount of chosen partitions for the processing chain is an open question for each developer, as having too much partitions introduces more overhead in the system and having too few partitions leads to insufficient information at the mediators about the performance of the chain.

6.2 Processing steps

The implementation of processing steps, with the communication with other processing steps and the mediators can be split up into two categories: collectors, and workers. The collectors provide the data source and manage the initial distribution of tasks and collect successfully executed tasks. Workers are the intermediate processing steps that actually transform the data in each step.

So for each processing chain one processing step is a collector and the other steps are workers in the chain.

The communication, such as shown in Figure 5.3, is implemented with ZeroMQ. Which is a lightweight library on top of simple TCP sockets, enabling easy communication between the components. For the processing steps only simple request and reply sockets are used. Communication with the mediators occurs with request sockets, while the communication for requesting a task from a previous processing step happens in a standard request-reply fashion. The holder of the tasks acts as reply server for the next processing step to request the task.

For serialisation and deserialisation of tasks the default Java serialisation from the `java.io` package is used. When only simple primitives are used this is not a very efficient solution, but the introduced overhead is ignored for this research. This can however be optimised in future work.

6.2.1 Workers

Worker instances are split up into receiver threads and distributor threads. The receiver threads receive new tasks from a mediator, after which the actual task is retrieved. The receiver is also responsible for actually executing the function that is implemented for that processing task. The receiver is implemented as an infinite loop that receives task headers, after which it contacts the instance responsible for the task header via a temporary socket to retrieve the task data.

The distributor thread is a server that listens for requests for tasks, so the distributor thread and the receiver thread have a necessity of a shared store of completed tasks that not yet have been transferred to the next processing step.

6.2.2 Collector

For the test executed in Chapter 7 the kind of input data is a simple list of random loads. Therefore, the current implementation only focuses on this use-case. Leaving the implementation of other data source, such as input from distributed file systems as HDFS or databases via query languages, for future work.

The collector is implemented with a separate task management and a receiver thread, so that these responsibilities can run at the same time. The task management thread is responsible for distributing tasks to the first mediator, this also means that the task management thread is responsible for resubmitting tasks that are likely to be lost in transmission or lost in a scale down of a processing step. The receiver thread acts in the same fashion as for the workers, except from the fact that no operations have to be done on the data.

The task management thread and the receiver thread share a store of tasks that still have to be fulfilled. At the moment that store is empty the collector finishes, which indicates that the processing chain successfully processed the batch and the monitor is allowed to clean up the started resources and processing step instances.

6.3 Mediators

The mediators are relatively simple entities, that is built up with three elements: the receiver of task headers, the distributor of task headers, and the metrics calculation and sharing. The basis of the mediator are two queues: the task queue, and the worker queue. The task queue contains the tasks that are published to the mediator but are not yet distributed, and the worker queue contains the workers waiting for new tasks.

The implementation of the mediator is self-explanatory given the architecture in Chapter 5.

6.4 Monitor

The monitor implements the management of processing chains, including the control loop for the decisions on the number of processing step instances and resource instances. For this it gathers metrics from mediators in processing chains, together with resource utilisation from the resource manager. In the control loop the monitor acts as controller, processing the available information and taking decisions on the number of instances for both processing steps as resources.

As the monitor requires a good overview of the running processing chains, it provides the API for the user interface. Such way, the user interface can retrieve interesting information on processing chains, to inform users on the state of the platform.

6.4.1 Metrics gathering

Two types of metrics are gathered by the monitor: processing chain metrics from the mediators, and resource metrics from the resource manager.

The mediators send their collected metrics on a set interval to the monitor. Therefore, the monitor implements a server that is able to receive the metrics of mediators. The monitor itself has no dynamic control on the interval mediators send their metrics, but is able to set a specified interval at the start of a processing chain. Readings of metrics received from the mediators are in the form of the case class in Listing 6.3.

```
case class MediatorReading(
                tasksDone: Int,
                taskQueueSize: Double,
                workerQueueSize: Double,
                numWorkers: Int,
                publishWaitTime: Float
)
```

Listing 6.3: Mediator readings format

Gathering resource metrics occurs via a pull-based system, as the resource metrics are retrieved from cloud providers. This way no additional servers have to run to gather metrics and send these to the mediators. Metrics gathered from the cloud provider usually occurs on a per VM basis, introducing the need to aggregate these metrics into averages for the resource types.

A Scala trait is introduced so that implementations for other cloud providers can be added easily to the system. This trait is a simple interface that requires a `gatherMetrics` method to be implemented, which should result in a sequence of VMMetrics, as shown in Listings 6.4 and 6.5.

```
trait VMMetrics {
  def gatherMetrics(): Future[Seq[VMMetric]]
}
```

Listing 6.4: VM metric trait

```
case class VMMetric(
                instance: String,
                os: OperatingSystem,
                timestamp: Date,
                cpuPercentage: Double,
                memoryPercentage: Double
)
```

Listing 6.5: VM metric format

The implementation of the gathering of VM metrics can either occur periodically or on a request basis in the control loop. At this point the gathering is done periodically every minute, as the metrics from Microsoft Azure are updated each minute.

6.4.2 Control loop

The control loop for determining the right amount of processing step instances, as well as, determining the right distribution of resource types consists of five main tasks:

1. Checking if all instances are running properly for a certain amount of time, to ensure previous decisions are visible in the retrieved metrics.
2. Determine the desired processing step instance counts.
3. Determine the deployable processing step instance counts.
4. Determine if changes in the resource distribution are needed.
5. Provision the calculated processing step instance counts.

As not all information is available instantaneously, the algorithm has to check if the needed information is available to the monitor. This happens based on predefined timeouts that, for instance indicate the time it usually takes for processing step instances to work fully, or the time it usually takes before resource usage is picked up fully by the resource manager. These timeouts cannot be defined on beforehand for all scenarios, different cloud providers can have different timeouts before resource utilisation can be measured correctly. In future work, these timeouts can be estimated at run-time by looking at trends in the metrics.

The determination of desired processing steps follows the solution and architecture (Chapter 4 and Chapter 5) completely.

Docker Swarm is used to provide the cluster manager functionality, unfortunately Docker Swarm has no intelligent algorithm implemented to decide where each container in the cluster should be deployed. It looks, at this moment, only at the number of containers running on each host to determine the best suitable host. In our case, this is not desirable because it can occur that multiple resource intensive containers are run on one host and containers that are mostly waiting on another host. Ideally, a cluster manager should provision based on the utilisation of host, and selecting the host that is underutilised first for deploying new containers. To overcome this problem, processing tasks that have a high resource demand reserves resources, preventing too much heavy containers to run on one host. This means that when determining the amount of processing step instances may not exceed the maximum amount of reservable resources.

Like the calculation of processing step instances, the calculation of the resource distribution follows the solution and architecture. The only additional feature that is built in is a cool down period after a change in resource distribution. To prevent continuously changing the resource distribution in the cases when the resource demand is not close to integers.

6.4.3 Application programming interface

The provided API of the monitor is implemented using the http4s library. The API provides functionality in the following categories:

- Starting and stopping processing chains
- Control loop logs
- Processing chain metric statistics
- Resource metric statistics
- Docker cluster information, including service and tasks information, and service logs.
- Export functionality of processing chain metrics and resource metrics to comma-separated values (CSV) files.

As the user interface is implemented in Javascript with AngularJS, the API provides all information in Javascript Object Notation (JSON) format, except for the CSV-files for exporting metrics.

The API right now only supports traditional HTTP calls, but in future work a WebSocket or HTTP long polling approach could be advantageous as the amount of API calls will be reduced drastically this way.

6.5 Microsoft Azure

For the evaluation of the adaptive approach, Microsoft Azure is chosen as cloud provider. All major cloud providers have the same functionality that are needed for the platform, as the requirements for the cloud provider are quite common.

The cloud provider must act as resource manager, and therefore, provide an interface to scale up or down resources types. As well as, providing utilisation metrics of the virtual machines in the cluster. For managing the resource types, deployed as Azure Virtual Machine Scale Sets, the Azure Virtual Machine Scale Sets REST APIs are used. For measuring the utilisation of virtual machines, the Azure Diagnostics Extension is used. This is an extension that runs on the actual virtual machines that records usage of the virtual machine on a 1 minute interval.

For deploying a cluster as used in Chapter 7, Azure templates are used to simplify the deployment of a scale set for each resource type and the management virtual machine. On the management virtual machine the monitor runs, as well as instances that do have some state, like processing chain collectors. As the scale sets themselves are scaled up and down, there is always some information lost in the case of deallocating virtual machines, which is not desirable for the processing chain collectors.

6.6 Docker & Docker Swarm

The usage of Docker enables components of the platform to run isolated and behave the same regardless the host configuration the Docker containers are run on. Enhancing the reproducability of the tests in the platform as configurations are clearly defined in the so called `Dockerfiles`. This way the installed libraries are the same for each run, and are others able to run the same tests without having to install specific libraries on host systems.

To deploy Docker containers over a cluster of machines, Docker Swarm is used. The main reason behind the decision for Docker Swarm is that it enables cross platform networking, so Linux Docker containers are able to communicate with Windows Docker containers via a shared subnet in an overlay network. Other functionality that is used from Docker Swarm is present in all major container cluster management solutions.

Chapter 7

Evaluation and results

In this chapter the adaptive approach is evaluated with a use case from the STOOP project. A comparison is made with the existing approach used in the STOOP project on performing calculations on models with different resource needs.

7.1 Case Study

The data and models used to determine the chances of failures of a pipeline are very diverse and are not all developed internally at TNO. Currently, an Apache Spark workflow is used in the STOOP project to perform statistical calculations.

The STOOP workflow is simplified by the pseudo-code in Listing 7.1. The input of the workflow is a list of pipeline segments. A probabilistic method is used on each of these segments. For the probabilistic method stochastic variations are determined and run in parallel. For each variation then the local calculation scenario for that combination of segment and variation is determined. Then a number of ground settlement and pipeline soil spring calculations are done, the number of these calculations depends on the combination of segment and variation used. The results are then used to calculate the segment stress on parts of the segment, the results of these calculations indicate whether or not the pipeline is likely to fail for these variations.

```
1 for every segment:
2   while not probab.converged()
3     for every variation:
4       constructRegionLayout()
5       doSettlementCalculations()
6       doPipelineCalculations()
7       combineResults()
8       calculateSegmentStress()
9       hasFailed()
```

Listing 7.1: Stoop workflow pseudo-code

Looking at this simplified code of the STOOP case, it is clear that there is a lot of dynamicity in the workflow. As the probabilistic method does not always converge after the same amount of iterations. And for each segment and even each variation the amount of settlement, pipeline, and segment stress calculations can change.

On top of these flow characteristics, also the environments the different calculations should be run on are different. Namely, the settlement and pipeline calculations are packaged into a Windows DLL, and the rest of the flow is run on Linux.

7.2 Models

For evaluating the adaptive approach, a part of the STOOP case study is tested. The models for calculating settlements and for calculating pipeline segment stresses are combined into a simple processing chain. For the settlement calculations a random amount of load is introduced on a fixed composition of soils, for which the settlement is calculated after 90 years. This settlement is used to calculate the stress that this settlement introduces on a pipeline segment, which can be used to determine the failure probability.

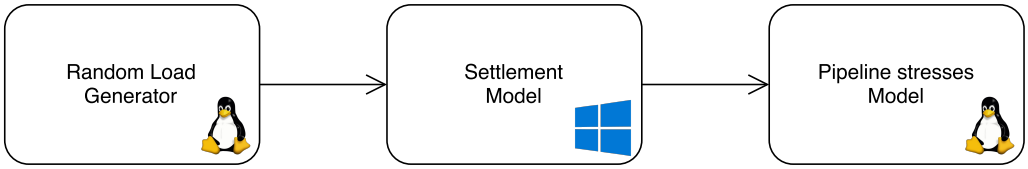


Figure 7.1: Test scenario

In this test case, we see dependencies on both Windows as well as Linux, where both of the compute intensive models are black box models that are used as is.

The lead times of the models are dependent on stochastic values, which makes it next to impossible to estimate the ratio between the models on beforehand.

The current solution used in the STOOP project makes use of Apache Spark. Where a Linux Spark cluster is deployed, with a Windows partner virtual machine for each Linux worker. At the moment the Linux workers are supposed to do a Windows calculation, a pipe with the partner virtual machine is setup to delegate the calculation. This setup with dedicated partner virtual machines inherently has an static and 1 to 1 distribution of resources.

7.2.1 Scenarios

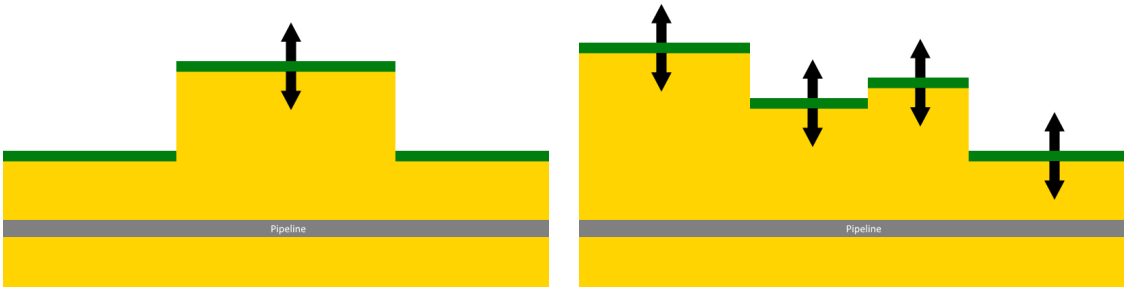
For testing the effectiveness of the platform different scenarios are tested, based on the models discussed. Because the solution is trying to reduce underutilisation of resources, the scenarios are chosen in a way that there is a significant underutilisation

present in the Spark scenario. This means that the main focus in the results lays in the improvement of the utilisation and if the adaptive approach is able to perform close to 100% utilisation of resources. The pipeline stress model takes roughly 2 times the time the settlement model takes per task, based on the stochastic values used in the scenarios.

The two main scenarios are as follows:

1. In one area additional load is introduced, while the surrounding areas above the pipeline are be constant. So one calculation is needed to predict the settlement of the additional load after 90 years, and one calculation is needed to determine the stress on the pipeline.
2. Four areas above the pipeline are of variable load, for each of these areas a settlement calculation has to be performed and with this information one pipeline stress calculation can be done. Each iteration of the scenario contains four random loads.

In Figure 7.2, the two scenarios are shown in simple graphics to clarify the difference between the scenarios.



(a) Scenario 1 (b) Scenario 2

Figure 7.2: Test scenarios

The first scenario has a higher demand for Linux, meaning that the Windows nodes will be underutilised. For the second scenario this is turned around, with a higher demand for Windows.

To be able to see the behaviour of the platform when the kind of load changes throughout a test, a combination of the two scenarios is introduced. In this combination, the first half of iterations is performed with one settlement calculation, the second half of iterations contains four settlement calculations.

For the tests a fixed amount of iterations is performed that vary the load. Giving an indication on what values for the loads are acceptable and do not introduce too much stress on the pipeline. The number of iterations per test are chosen in a way that the total lead time is large enough for the adaptive approach to be able to provision new resources. The three different scenarios are tested with 20 VMs, with 5 runs for each test. The first scenario, with one settlement calculation per pipeline

stress calculation, is also performed with 100 VMs for 3 runs for each test. The total test setup is shown in Table 7.1.

Scenario	# Cores (VMs)	# Iterations	# Runs
1	40 (20)	15000	5
2	40 (20)	7500	5
3	40 (20)	15000	5
1	200 (100)	150000	3

Table 7.1: Test overview

The tests are done on the Azure cloud platform, with Standard D2 v3[1] instances that are equipped with two cores and 8GB RAM. All VMs are deployed in the same Azure region to have the best network connectivity between instances.

The throughput of the Spark setup is measured by logging the task time of each task.

7.3 Results

The results of the tests are grouped based on the number of VMs used. The tests with 20 VMs are discussed based on the average of the 5 runs. Whereas, the tests with 100 VMs are discussed briefly with the average result of the 3 runs and a closer look is performed on the first run of the adaptive approach to show the decisions that are taken.

The results of the adaptive approach can be shown fine-grained, as the gathered metrics are stored. The results of the Spark setup are determined based on task duration, where each task contained 125-150 iterations.

7.4 20 virtual machines

In Figures 7.3, 7.4, and 7.5 the results of the tests with 20 VMs are displayed. Each of the tests is run 5 times and the average of these 5 runs is shown in the figures. For the adaptive approach the cluster starts with 2 Windows VMs and 2 Linux VMs.

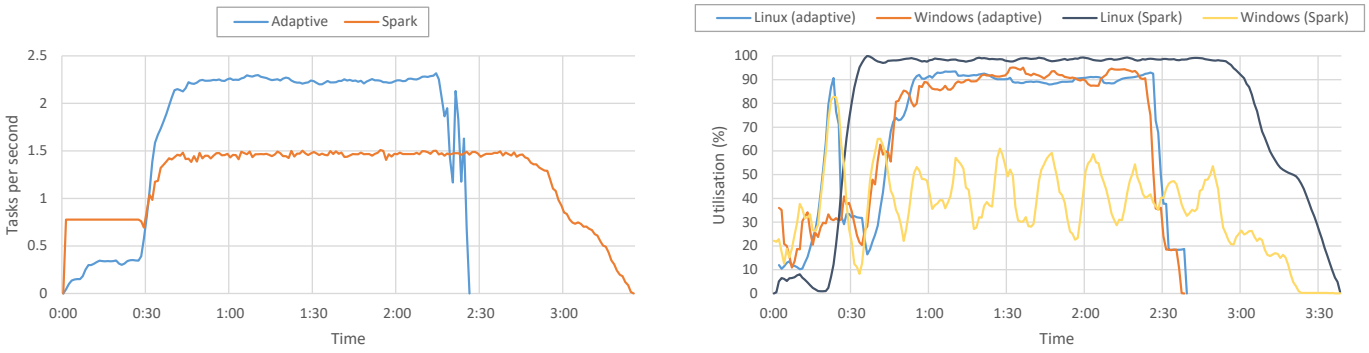
When taking a look at the results of the first scenario, it is clear that the adaptive approach outperformes the Spark setup significantly when the resources are scaled up to the resource budget of 20 VMs around the 30 minute mark. The Spark setup initially has a lower throughput then later in the test, this can be explained by the fact that in the Spark setup when the first tasks are handled the settlement

[1] Microsoft Azure instance overview

model must be loaded. This takes around 10 minutes to complete, therefore, the throughput of the Spark setup increases after the first tasks are completed after 30 minutes.

When looking at the difference in throughput when both approaches reached their maximum throughput, we see an improvement of roughly 50%. Also the lead time of the complete scenario is reduced by 59 minutes on a lead time of 206 minutes, which is an reduction of 29%.

The resource utilisation of both approaches show the reason behind these improvements. Spark is able to maximise the resources of the Linux VMs, but the Windows VMs have an average of 38% utilisation over the scenario. The adaptive approach is not able to maximise one of the resources, but both resource types have an average utilisation of 70%. When looking at the stable part of the scenario, between 60 and 120 minutes, we see an average cluster usage of 91% for the adaptive approach and 70% for the Spark approach.



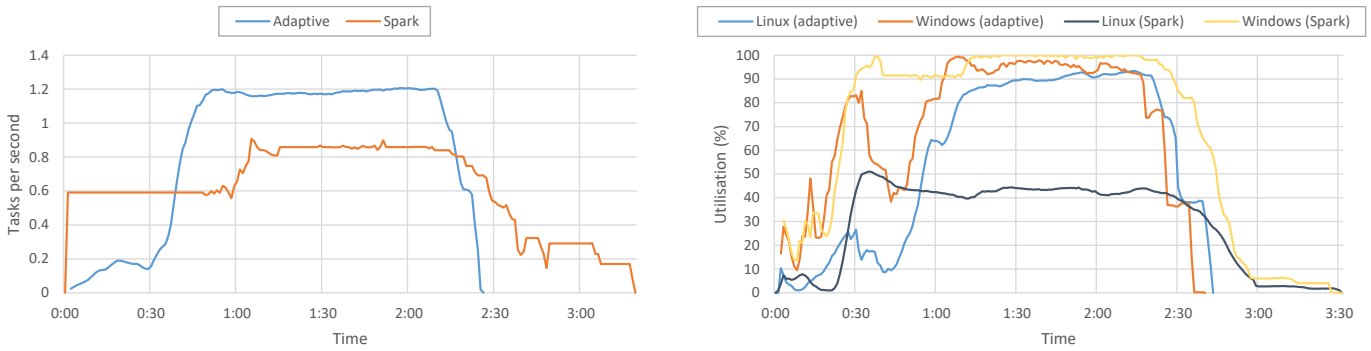
(a) Throughput

(b) Resource utilisation

Figure 7.3: Results of scenario 1 with 20 VMs, averaged over 5 runs

The results of the second scenario show a similar picture as the first scenario. However, the differences between the approaches is less than in the first scenario. But still an improvement in throughput when the approaches are stable of 40% is made. Also an reduction in lead time of 53 minutes is realised, based on a lead time of 200 minutes this is an improvement of 27%.

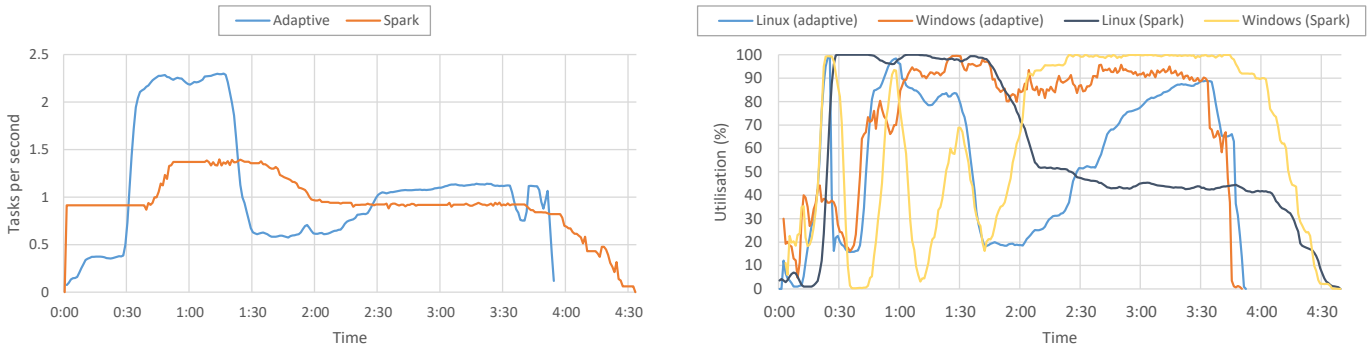
The resource utilisation also shows an similar picture, the window the adaptive approach is stable is, however, smaller than in the first scenario. Looking at this stable windows from 70 minutes to 120 minutes, the average resource utilisation of the adaptive approach is 92% and for the Spark approach it is 71%. Which is comparable to the first scenario.



(a) Throughput (b) Resource utilisation

Figure 7.4: Results of scenario 2 with 20 VMs, averaged over 5 runs

The results of the third scenario show how well the approaches are able to adapt to changing resource demands throughout the scenario. The first 90 minutes show roughly the same results as the first scenario, where the adaptive approach outperformes the Spark scenario. At this 90 minute mark the adaptive approach reaches the shift in demand, a large dip in the throughput is noticeable. At that moment it tries to find a new resource distribution to fit the demand. At the 150 minute mark, the adaptive approach starts to outperform the Spark approach again. The difference in lead time between the approaches is 40 minutes, which is a 14% improvement. The overall resource utilisation of the adaptive approach is 65%, whereas the Spark approach achieves an resource utilisation of 60%.



(a) Throughput (b) Resource utilisation

Figure 7.5: Results of scenario 3 with 20 VMs, averaged over 5 runs

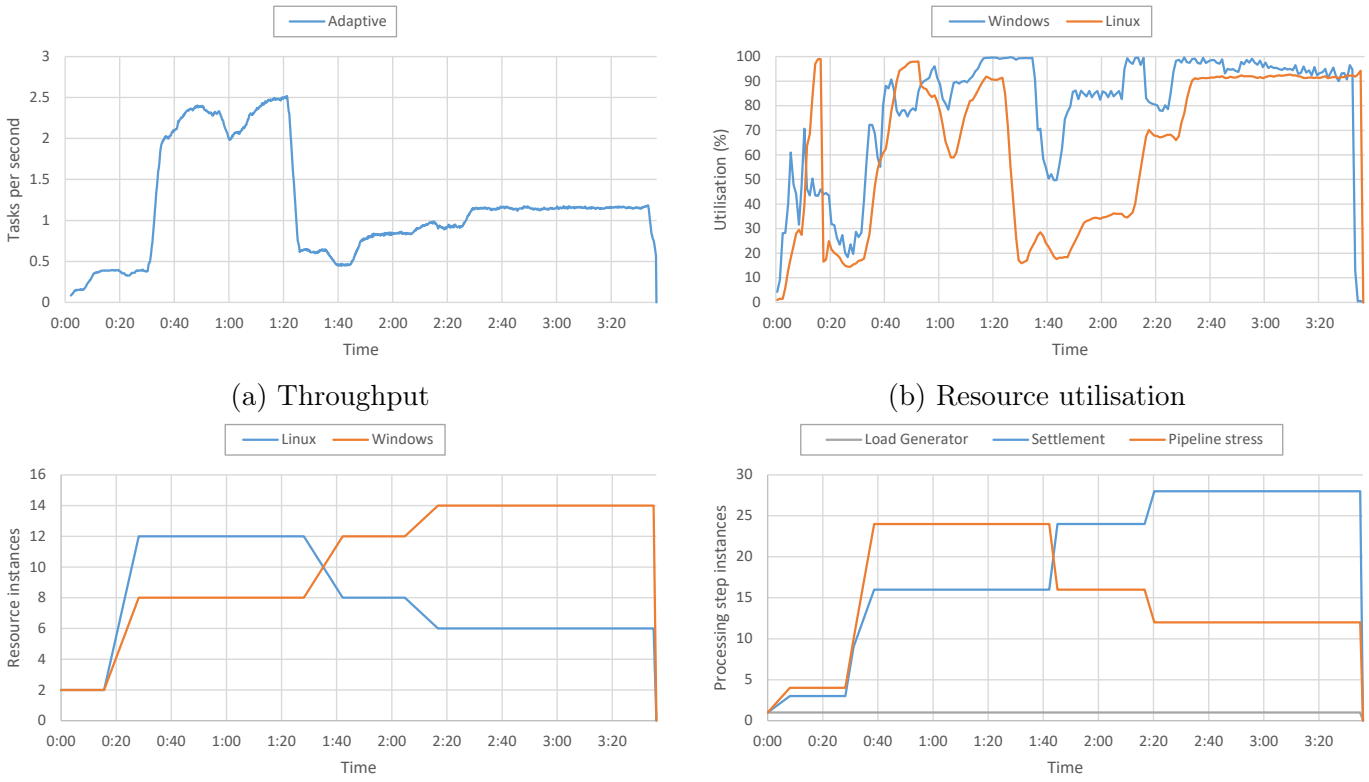
As this scenario is able to show how well the adaptive approach is able to handle changing needs in the processing chains, a closer look is taken at the first run of the adaptive approach with 20 VMs for this scenario. In Figure 7.6, the throughput and resource utilisation are shown together with the resource instances and processing step instances.

It is clear that around 1:25 the scenario flips to the four settlement calculations,

which is also clearly visible in the resource utilisation. The adaptive approach makes its first decision shortly after the dip in resource utilisation of the Linux cluster, when it detects a large difference in utilisation between Windows and Linux. The processing step instances are scaled just shortly after the resources are fully provisioned.

Interesting too see is the dip at 1:00, which was introduced by an error in a Windows VMs that resulted in a loss of network connections. After a while the containers on that host were restarted and worked the cluster properly again. This error did not have any effect on the adaptive approach in the rest of the test.

Due to this dip no clear stable period can be found for the first half of the calculations. However, for the second half of the calculations a clear stable period is present from 2:30 until the end of the run around 3:30.



(a) Throughput

(b) Resource utilisation

(c) Resource instances

(d) Processing step instances

Figure 7.6: Results of the first run of scenario 3 with 20 VMs

Summarising the results of the test with 20 VMs, Table 7.2 shows the improvements of the adaptive approach relative to the Spark approach. The throughput and resource utilisation improvements are measured when both approaches are stable, except from the resource utilisation of scenario 3, as no clear stable windows can be determined.

These percentages show a significant improvement in all facets of tests. Also when taking into account that the adaptive approach has to provision new resources at

Scenario	Throughput	Resource utilisation	Lead time
1	50%	30%	29%
2	40%	29%	27%
3	57% & 22%	8.3%	14%

Table 7.2: Improvements of the adaptive approach relative to the Spark approach

runtime, which introduces delays in computation, the lead time improvement is likely to approach the throughput improvement when the scenarios take longer to process.

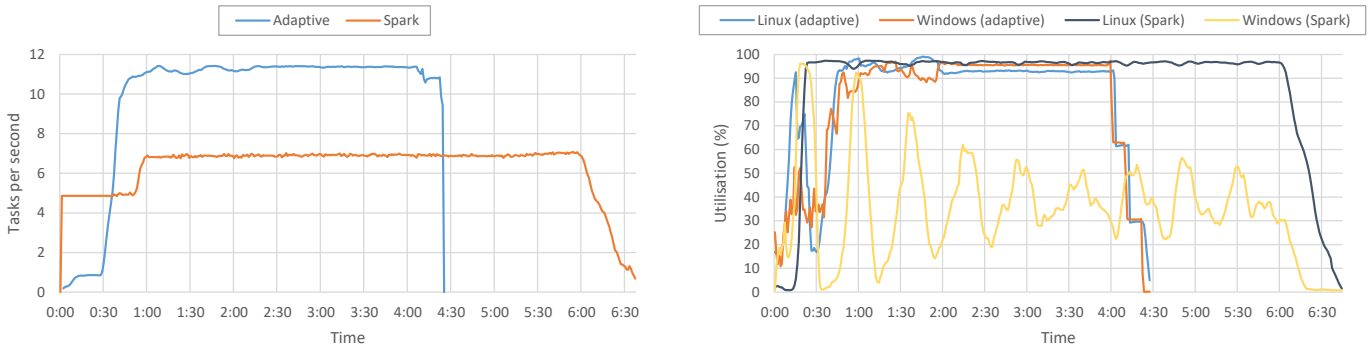
The improvement in resource utilisation is the result that is the most interesting, as the increased resource utilisation leads to the increased throughput and lead time.

7.5 100 virtual machines

The test with scenario 1 with 100 VMs shows largely the same result as the tests with 20 VMs. The throughput achieved by the adaptive approach when it stable, between 60 minutes and 240 minutes, is 11.3 iterations per second. The Spark approach reaches 6.9 iterations per second. This means an improvement of 64%, which is significantly larger than in the 20 VM case.

When comparing the throughputs of the 100 VM and the 20 VM cases, the 20 VM case has a throughput of 2.24, so the 100 VM case achieves a scalability factor of 1.008. Whereas, the Spark scenario achieves a scalability factor of 0.944.

Looking at the resource utilisation, an average utilisation when the approaches are stable of 94% for the adaptive approach and 67% for the Spark approach are reached.



(a) Throughput

(b) Resource utilisation

Figure 7.7: Results of scenario 1 with 100 VMs, averaged over 3 runs

A closer look at the processing step distribution (Figure 7.8) and resource distribution (Figure 7.9) of the first run of the adaptive approach show that after 22

minutes the first resource provision step is made. From a 5:5 distribution of Linux and Windows, the decision is made to adapt to a 64:36 distribution. After this the adaptive approach changes the distribution several times, in order to reach a stable distribution after 105 minutes. The processing step distribution follows the resource distribution closely.

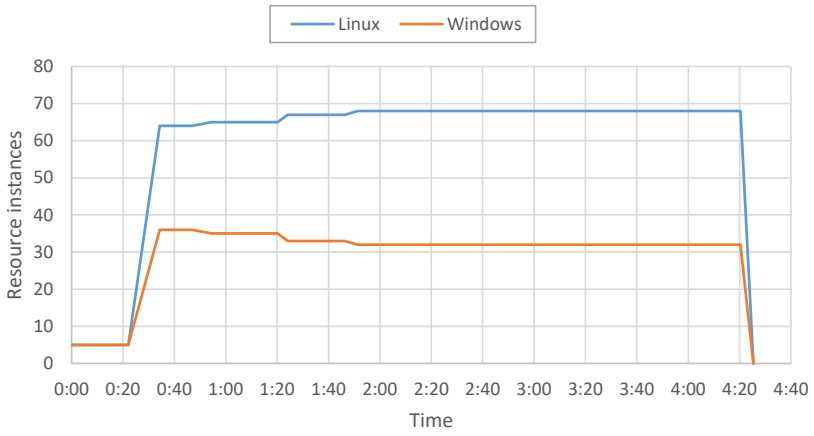


Figure 7.8: Resource distribution scenario 1 with 100VMs

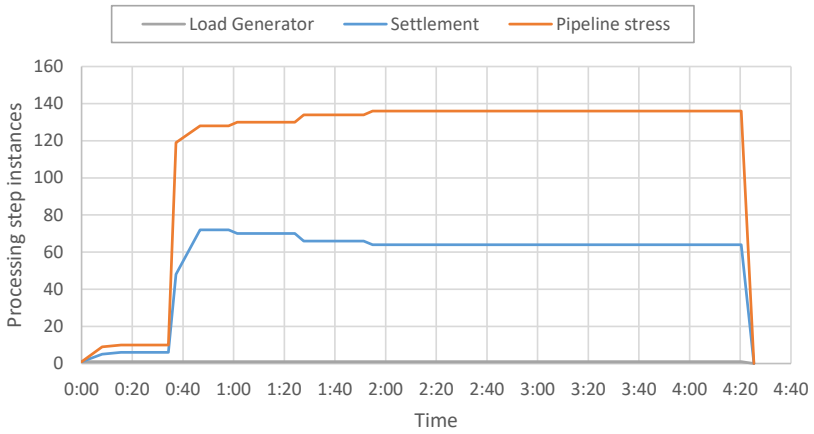


Figure 7.9: Processing step distribution scenario 1 with 100VMs

Figure 7.10 shows the overall utilisation of the different resource types. After the last resource provision step the utilisation stabilises at 92% for Windows and 89% for Linux.

The dip after 22 minutes in utilisation of Linux can be explained by the fact that provisioning Linux VMs takes less time than provisioning Windows VMs. When both resource types are provisioned correctly the algorithm carries on with calculating the processing step distribution, which results in an increase of the resource utilisation.

To estimate the cost improvement of the tests, the amount of VM hours can be determined. For the Spark scenario this is easy, as its tests ran for 404 minutes with

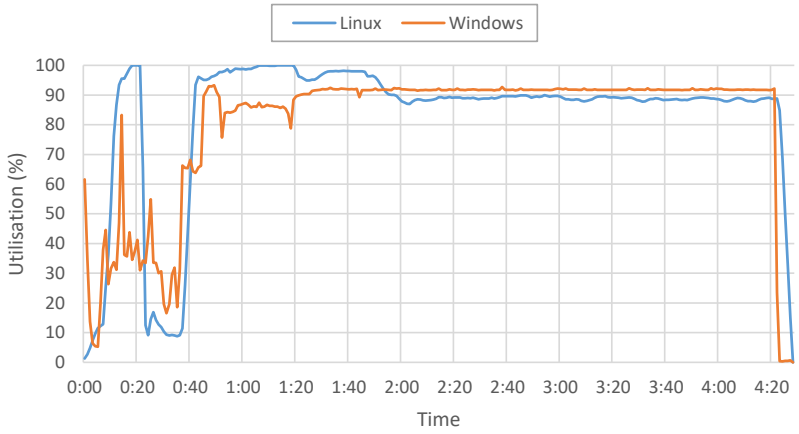


Figure 7.10: Resource utilisation scenario 1 with 100VMs

50 Linux VMs and 50 Windows VMs. For the adaptive approach a calculation has to be made for each window between provision steps. The results of this calculation are shown in Table 7.3. In this table also the costs are displayed, based on the pricing at the 27th of September. Which are $0.212 per hour for Windows VMs and $0.120 for Linux VMs.

Approach	VM hours (Windows, Linux)	Costs (Windows + Linux)
Adaptive	(135.33, Linux: 273.80)	$28.69 + $32.86 = $61.55
Spark	(336.67, Linux: 336.67)	$71.37 + $40.40 = $111.77

Table 7.3: Costs of different approaches on Microsoft Azure

The differences in costs are even greater than previous comparisons, with nearly halving the costs of the Spark approach, mainly because the adaptive approach used more Linux VMs which are cheaper due to reduced licence costs.

7.6 Test difficulties

When testing both approaches, some difficulties came up that made it harder to correctly test the different scenarios. Some of the larger difficulties are discussed.

The first and major obstacle that came up was the fact that when deploying larger amounts of VMs the chance of incorrectly provisioned VMs increases. On average, around 3% to 5% of the VMs did not provision correctly. To overcome this obstacle, overprovisioning tackled this problem largely. The virtual machine scale sets of Azure have the option to automatically overprovision VMs and deallocate the VMs that are added in abundance. For the Spark approach a normal set of VMs was used, so the overprovisioning had to be done manually. By deploying more VMs and stopping VMs just before starting the Spark job.

Another obstacle is the fact that Windows support for Docker Swarm is in a beta stage. This was visible the most in the networking stack of containers in the Swarm cluster. Sometimes containers do not have a network connection, which makes them unusable for distributed calculations. To overcome this problem, the processing step instances check if they have an active network connection when they are started, if not they exit and a new container is started. This solution works for the largest part, only when the network connection dropped when the processing step instance was running already this did not work. This is most likely the reason that the adaptive approach did not reached the 100% resource utilisation for one of the resource types.

Chapter 8

Future work

As for every project, and especially because of the fact that a new solution and platform is created in this thesis, there are still areas in which further research can be performed. The major areas that have been identified are:

Data locality: For the reference platform to be useful for big data applications, it is necessary to prevent data transfer through disks or even through networks. For big data applications this can influence the performance of the application greatly. When for instance a simple word count application needs to process terrabytes of data it is very inefficient to transfer this data through different machine a couple of times.

Improve efficiency: Improving the efficiency of the runtime components of the platform can result in the platform using one of the resource at near 100% levels, instead of the 90%-95% that is achieved right now. For instance, a good algorithm to create directed acyclic graphs of the processing chain and combining steps that have the same resource requirements. This would lower the number of hops the data has to travel through the processing chain. But also improving the stability of VMs in a cluster or improving the stability of Docker Swarm can help in achieving greater efficiency.

Distributed file systems and database queries: Right now only single data sources are supported by the platform. Supporting distributed file systems like the Hadoop Distributed File System or query languages to process data from databases on multiple data source instances can help in achieving better scalability of the platform.

Machine learning control loop frequency: Machine learning can be effective to reduce the currently static control loop intervals the platform uses. The lower these intervals are, while preserving the same decision certainty, the more the lead time decreases of processing chains run on the platform.

Chapter 9

Conclusion

In this thesis a new approach to adaptively provision resources based on metrics from both the application layer as well as the resource layer. By proposing a general approach and implementing a reference platform for the STOOP project significant improvements are made. The main research question posed is:

> *How to adaptively provision heterogeneous resources for compute intensive data processing?*

To be able to answer the main research question, the sub-questions identified in Section 1.2 have to be answered first. The first sub-question that is answered is:

> *Which metrics are needed to provide a clear overview of the status of the processing chain?*

In Section 3.2 the basis of the necessary metrics is identified. For processing chains, three metrics are distinguished: wait times to publish tasks, wait times for consuming tasks, and changes in the local buffers in between processing steps. Especially the wait times are extremely effective to estimate if an processing chain is in balance, as this would mean that none of the processing steps is waiting and thus fully utilised.

Metrics for resources are a bit less concrete than the metrics of processing chains. As it depends heavily on the kind of resources that are used which metrics are able to describe the utilisation of resources. Therefore, in the algorithms, a percentage of utilisation is used. This can be implemented differently for different types of resources. In the case of virtual machines with compute intensive models, the percentage of CPU usage is used to determine the utilisation of the resources.

These metrics are used to assist in answering the second sub-question:

> *How to continuously decide efficient distributions of processing step instances?*

The answer to this question is described in Section 4.1 and Section 4.2. A series of calculations based on the metrics identified earlier enables to calculate the desired

processing step distribution at the moment of when the metrics are gathered. And a way to convert this distribution to actual instance counts that can be deployed in the processing chain.

The last sub-question that is needed to answer the main research questions is:

> *How to continuously decide efficient distributions of resource types in the cluster?*

The answer to this question is relatively simple, as the utilisation of the different resource types should be on the same level. The only calculations that have to be made is the calculation of the distribution, which can be related directly to the utilisation metric. As well as, determining the actual counts of the resources by multiplying the distribution by the resource budget.

Now that the sub-questions are answered, the main research question can be discussed. By gathering the necessary metrics for the algorithm to estimate the desired distribution of processing step instances and thus the desired distribution of resources, we are able to create a control loop that is able to continuously adapt the processing chain and its resources to the estimated distributions.

In Chapter 7, this solution is evaluated with very promising results. An increase in throughput of up to 50% is achieved. Of course, the kind of scenarios tested plays an important factor in the results of the solution. When the demand for different resource types by an application is roughly equal, the adaptive approach will not achieve an improvement.

We can conclude that adaptive provisioning of heterogeneous resources for compute intensive data processing can be very beneficial for applications with an inbalanced resource demand. By using application metrics and resource metrics a control loop can be setup that continuously adapts to changing demand from the application.

Furthermore, the ability of the adaptive provisioning enables the creation of applications without the need of determining the correct ratio of resource types on beforehand. This could possibly save a lot of development time and time for trial runs. As developers are able to create and deploy applications without the necessity of estimating the ratio of processing step instances and resource instances.

Bibliography

[1] H. Ford and S. Crowther. *My Life and Work*. c, 1922.

[2] Taiichi Ohno. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1988.

[3] Jeffrey Dean and Sanjay Ghemawat. 'MapReduce: Simplified Data Processing on Large Clusters'. In: *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, pp. 10–10.

[4] Faraz Ahmad et al. 'Tarazu'. In: *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12*. ACM Press, 2012. DOI: 10.1145/2150976.2150984.

[5] Ankit Toshniwal et al. 'Storm@twitter'. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*. ACM Press, 2014. DOI: 10.1145/2588555.2595641.

[6] Y. Xing, S. Zdonik and J. H. Hwang. 'Dynamic load distribution in the Borealis stream processor'. In: *21st International Conference on Data Engineering (ICDE'05)*. Apr. 2005, pp. 791–802. DOI: 10.1109/ICDE.2005.53.

[7] M. A. Shah et al. 'Flux: an adaptive partitioning operator for continuous query systems'. In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. Mar. 2003, pp. 25–36. DOI: 10.1109/ICDE.2003.1260779.

[8] Rebecca L. Collins and Luca P. Carloni. 'Flexible filters'. In: *Proceedings of the seventh ACM international conference on Embedded software - EMSOFT '09*. ACM Press, 2009. DOI: 10.1145/1629335.1629363.

[9] Charles Zheng and Douglas Thain. 'Integrating Containers into Workflows'. In: *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing - VTDC '15*. ACM Press, 2015. DOI: 10.1145/2755979.2755984.

[10] Kai Liu et al. 'Flexible Container-Based Computing Platform on Cloud for Scientific Workflows'. In: *2016 International Conference on Cloud Computing Research and Innovations (ICCCRI)*. IEEE, May 2016. DOI: 10.1109/icccri.2016.17.

[11] Yong Zhao et al. 'Enabling scalable scientific workflow management in the Cloud'. In: *Future Generation Computer Systems* 46 (May 2015), pp. 3–16. DOI: 10.1016/j.future.2014.10.023.

[12] F. Warren Burton and M. Ronan Sleep. 'Executing functional programs on a virtual tree of processors'. In: *Proceedings of the 1981 conference on Functional programming languages and computer architecture - FPCA '81*. ACM Press, 1981. DOI: 10.1145/800223.806778.

[13] Umut A. Acar, Arthur Charguéraud and Mike Rainey. 'Scheduling parallel programs by work stealing with private deques'. In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '13*. ACM Press, 2013. DOI: 10.1145/2442516.2442538.

[14] Matei Zaharia et al. 'Improving MapReduce Performance in Heterogeneous Environments'. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 29–42.

[15] Rene Peinl, Florian Holzschuher and Florian Pfitzer. 'Docker Cluster Management for the Cloud - Survey Results and Own Solution'. In: *Journal of Grid Computing* 14.2 (Apr. 2016), pp. 265–282. DOI: 10.1007/s10723-016-9366-y.

[16] Abhishek Verma et al. 'Large-scale cluster management at Google with Borg'. In: *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*. ACM Press, 2015. DOI: 10.1145/2741948.2741964.

[17] Simon Ostermann, Radu Prodan and Thomas Fahringer. 'Dynamic Cloud provisioning for scientific Grid workflows'. In: *2010 11th IEEE/ACM International Conference on Grid Computing*. IEEE, Oct. 2010. DOI: 10.1109/grid.2010.5697953.

[18] Rajkumar Buyya and Diana Barreto. 'Multi-cloud resource provisioning with Aneka: A unified and integrated utilisation of microsoft azure and amazon EC2 instances'. In: *2015 International Conference on Computing and Network Communications (CoCoNet)*. IEEE, Dec. 2015. DOI: 10.1109/coconet.2015.7411190.

[19] Qi Zhang et al. 'Dynamic Heterogeneity-Aware Resource Provisioning in the Cloud'. In: *IEEE Transactions on Cloud Computing* 2.1 (Jan. 2014), pp. 14–28. DOI: 10.1109/tcc.2014.2306427.

[20] Qi Zhang, Ludmila Cherkasova and Evgenia Smirni. 'A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications'. In: *Fourth International Conference on Autonomic Computing (ICAC'07)*. IEEE, June 2007. DOI: 10.1109/icac.2007.1.

[21] Dan Xu, Xin Liu and Bin Fan. 'Efficient Server Provisioning and Offloading Policies for Internet Data Centers with Dynamic Load-Demand'. In: *IEEE Transactions on Computers* 64.3 (Mar. 2015), pp. 682–697. DOI: 10.1109/tc.2013.2295797.

[22] Nidhi Jain Kansal and Inderveer Chana. 'Cloud Load Balancing Techniques: A Step Towards Green Computing'. In: vol. 9. 1. IJCSI, Jan. 2012.

[23] Steve Crago et al. 'Heterogeneous Cloud Computing'. In: *2011 IEEE International Conference on Cluster Computing*. IEEE, Sept. 2011. DOI: 10.1109/cluster.2011.49.

[24] Long Thai, Blesson Varghese and Adam Barker. 'Algorithms for Optimising Heterogeneous Cloud Virtual Machine Clusters'. In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, Dec. 2016. DOI: 10.1109/cloudcom.2016.0033.

[25] Linchuan Chen, Xin Huo and Gagan Agrawal. 'Accelerating MapReduce on a coupled CPU-GPU architecture'. In: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2012. DOI: 10.1109/sc.2012.16.

[26] Koichi Shirahata, Hitoshi Sato and Satoshi Matsuoka. 'Hybrid Map Task Scheduling for GPU-Based Heterogeneous Clusters'. In: *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE, Nov. 2010. DOI: 10.1109/cloudcom.2010.55.

[27] Mohamed Hassaan and Iman Elghandour. 'A real-time big data analysis framework on a CPU/GPU heterogeneous cluster'. In: *Proceedings of the 3rd IEEE/ACM International Conference on Big Data Computing, Applications and Technologies - BDCAT '16*. ACM Press, 2016. DOI: 10.1145/3006299.3006304.