INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Parallel execution of pipelines using bioinformatics tools

CALMENELIAS PINO FLEITAS

(Licenciada)

Projecto Final para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Doutora Cátia Raquel Vaz
Doutor José Simão

Júri:

Presidente: Doutor Tiago Miguel Braga Da Silva Dias

Vogais: Doutor Carlos Jorge de Sousa Gonçalves
Doutor José Manuel De Campos Lages Garcia Simão

NOVEMBRO, 2019

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Parallel execution of pipelines using bioinformatics tools

CALMENELIAS PINO FLEITAS

(Licenciada)

Projecto Final para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Doutora Cátia Raquel Vaz
Doutor José Simão

Júri:

Presidente: Doutor Tiago Miguel Braga Da Silva Dias

Vogais: Doutor Carlos Jorge de Sousa Gonçalves
Doutor José Manuel De Campos Lages Garcia Simão

NOVEMBRO, 2019

Aos meus pais e familia.

Acknowledgments

Aos meus orientadores, por todo o apoio e força que me deram ao longo da realização desta dissertação. A todos os meus amigos que me acompanharam, ajudaram e animaram nesta jornada. Aos meus familiares por todo o apoio e confiança. E em particular, um grande agradecimento aos meus pais, sem eles nada disto seria possível.

Abstract

The project "Parallel execution of pipelines using bioinformatic tools", from now on referred to as *NGSPipesV2*, is part of a platform that allows the creation and execution of *pipelines* (set of tools to execute).

NGSPipesV2 extends the project "Infrastructure to support the execution of workflows for bioinformatics", from now on referred to as *NGSPipesV1*. *NGSPipesV1* project was developed within the final project of the Licenciatura em Engenharia Informática e de Computadores (LEIC) of Instituto Superior de Engenharia de Lisboa (ISEL).

The main goal of both projects is to help the scientific community to perform biological data processing using Next Generation Sequencing (NGS) techniques. For this purpose, both *NGSPipes* projects supports the creation and execution of *pipelines* (e.g. sequences of tasks), avoiding scientists to solve problems like installing tools and their dependencies and write *scripts* in order to execute *pipelines*.

NGSPipesV2 extends *NGSPipesV1* with the main objectives of adding support for the parallel execution of *pipelines*, orchestrate the execution of *pipelines* in remote clusters and improve the expressiveness of metadata for tools annotation.

Keywords: NGSPipes; Pipeline; Execution; Tool metadata.

Resumo

O projeto "Execução paralela de fluxos de trabalho usando ferramentas bioinformáticas", de agora em diante nomeado como *NGSPipesV2*, é parte de uma plataforma que permite criar e executar *pipelines* (conjunto de ferramentas para executar).

NGSPipesV2 é uma extensão do projeto "Infraestrutura de suporte à execução de fluxos de trabalho para a bioinformática", de agora em diante nomeado como *NGSPipesV1*. O projeto *NGSPipesV1* foi desenvolvido no contexto do projeto final da Licenciatura em Engenharia Informática e de Computadores (LEIC) do Instituto Superior de Engenharia de Lisboa (ISEL).

O objetivo principal de ambos os projetos *NGSPipes* é apoiar a comunidade científica a realizar o processamento de dados biológicos utilizando técnicas de Next Generation Sequensing (NGS). Para isto é suportada a criação e execução de *pipelines*, isto é sequências de tarefas, evitando que os cientistas tenham de resolver problemas tais como: a instalação de ferramentas e as suas dependências e escrever *scripts* para poder executar os seus fluxos de trabalho.

NGSPipesV2 estende *NGSPipesV1* com os objetivos principais de adicionar suporte para a execução de *pipelines* em paralelo, orquestrar a execução de *pipelines* em clusters remotos e melhorar a expressividade dos metadados para anotação das ferramentas.

Palavras-chave: NGSPipes; Fluxo de trabalho; Execução; Metadados das ferramentas.

Contents

List of Figures

List of Tables

Listing

Glossary

bioinformatics application of informatics techniques on biologist and health areas, to help analyse biological data. 1

data-parallelism is to use different files to perform the same command. 10

DNA deoxyribonucleic acid, is the hereditary material in humans. xxiii

NGS (Next-Generation Sequencing) are technologies (e.g. Illumina sequencing) that allows to produce DNA sequences. 1

nucleotide are the building blocks of nucleic acids. 9

pipeline is a sequence of commands executed sequentially, where an output from a command is used as input to the next one. 1

reads small sequences of DNA. 1

semi-automated let user decides when he wants to permit workflows to execute independent tasks in parallel. 20

SWS (Scientific Workflow System) are systems to help simplify workflows/ pipelines construction and execution. 2

task is a command in execution time or even a set of them. 1

task-parallelism is refer to tasks that are independent could execute in parallel. 2

workflow is a pipeline, non-linear, that support loops and branches. 1

1

Introduction

The amount of data available to research groups and companies in the health industry is constantly increasing. Bioinformatics is defined as the application of informatics techniques on Biology and health areas to help analyse biological data. Considering the use case of *bioinformatics*, either at research level or industry level, they use Next-Generation Sequencing (*NGS* [4]) techniques to produce small pieces of sequences from original DNA (Deoxyribonucleic acid) samples, known as *reads*. These *reads* are processed and analysed by a set of linked *tasks* of computational analyses and visualization. These set of tasks can be considered as a *pipeline*. A related term is *workflow* which can be transactional or cyclic. A *pipeline* is composed by a set of processes, usually acyclic, and involve the use of data resources in a staged fashion, with the output of one tool being passed to the next one. These tasks can run concurrently and each one has its own requirements (e.g. installation, required memory and execution) along with their parameters.

A common way to create and execute a *pipeline* is using scripts. When analysing how to create and execute a *pipeline*, some requirements can be defined:

1. choose which tasks to execute;
2. know how to use each task
    (a) installation;

(b) task parameters (types, which one is an input or a output);

(c) execution;

3. know how to execute the *pipeline* taking into account tasks order

   (a) create a script to execute the *pipeline* (requires knowledge on the scripting language);

   (b) execute manually each task on command line (requires to be pendent whenever each task ends to execute the next one) or use pipes [38].

On a biologist's perspective, without programming knowledge, it could be hard to build and execute a *pipeline*. Also, the execution of tasks that don't have dependencies could be parallelized taking advantage of available resources, which can result in time gains. *Pipeline* parallelization can be difficult to do without a solution that provides these facilities. Scientific Workflow Systems (*SWS*) helps to solve this problem. These systems main goal is to provide a solution that allows a scientist to create and/or execute a *pipeline* as transparent as possible. Transparency means that users don't have to deal with aspects such as tools resources (e.g memory, CPU) requirement, tools installation and manage execution (e.g. dependency, parallelism inference). So, to guaranty each task is well used, these systems need to infer and validate if tool parameters are used correctly and even infer how to execute each tool and the *pipeline* itself. For these purposes SWSs needs some tool's metadata and an infrastructure to manage the execution.

Execution infrastructures are a set of computers (nodes), generally connected through a local network, which work as single one. The main objective is to coordinate and use nodes to execute tasks in a short amount of time. SWSs uses these infrastructures to deal with the *task-parallelism* and distribution to gain time by balancing the nodes loads.

1.1 Aim of the thesis

This project objective is to help scientific community on the creation and execution of *pipelines* by supporting execution parallelization (multi-core, multi-task and data-partition execution) and tools metadata annotation. The overall objective is to offer a solution where the process described above could be as automated and easy as possible.This project extends the final year project *NGSPipes*

[43][9], from now on the old version will be refered as *NGSPipesV1* and new version as *NGSPipesV2*, by adding execution parallelization and improving tools annotation expressiveness (execution command construction, dependency between parameters). *NGSPipesV1* main features are: `Domain Specific Language` (DSL [16]), a language to specify *pipelines*; `Repository` of tools which describes tool's metadata and `Engine` where *pipelines* are executed.

To accomplish *NGSPipesV2* goals, the following changes were made:

1. *DSL*
   (a) enrichment of specific language primitives to support parallelism;
   (b) support to argument definition, which allow users to declare arguments that are specified at *pipeline* execution time. These arguments , making possible to support abstract *pipelines*;
2. *Repository* extended as *Tool repository*
   (a) update of tool metadata repositories;
   (b) update tool annotation mapping;
3. *Engine* execution environment with support to
   (a) multi-core;
   (b) multi-task;
   (c) data-partition.

Along with this project was developed another project thesis, "Bioinformatic pipeline specification language and sharing system" [10], which extends DSL and Repository (as *Pipeline repository*) features.

1.2 Outline

This document is divided in six chapters. Chapter 2 describes the use case that will be used to illustrate this project features. Chapter 3 presents a comparison among some relevant Scientific Workflow Systems (SWS) focusing on *pipeline* execution context. Chapter 4 presents the approach proposed architecture and principal concepts of this project. Chapter 5 describes solution implementation details, technologies and obtained results. Chapter 6 contains final remarks and discuss possible future work.

2

Case studies

This chapter explains the case studies in which this project is based and the principal terms used among this document within NGSPipes project context.

2.1 NGSPipes context

In this section are defined some used terms within *NGSPipes* project.

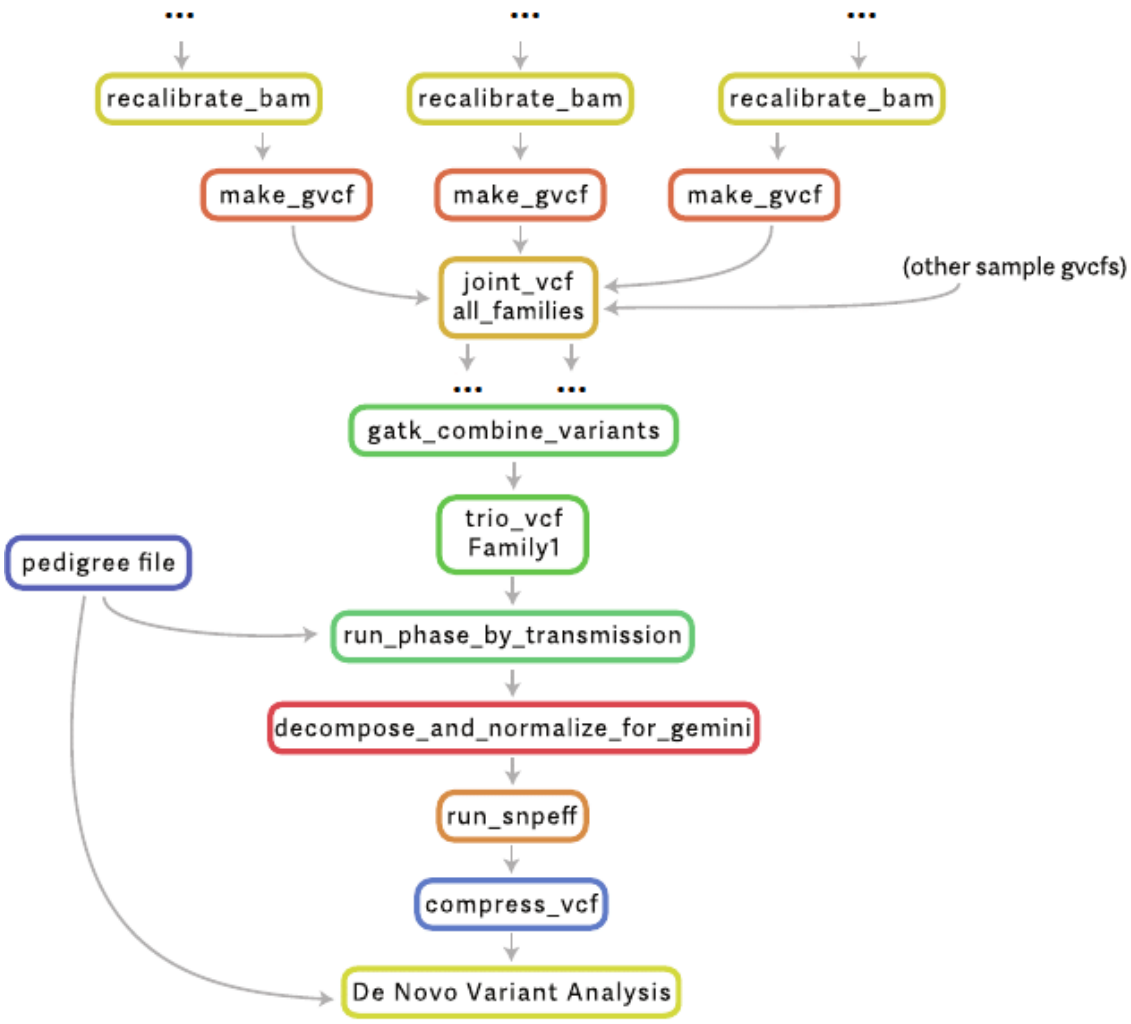
1. **Pipeline** - is known as a sequence of commands executed sequentially, where an output from a command goes as input to next one.
   Linux pipes are an example of *pipeline* implementation, see Listing 2.1. The character `|` divides each command within sequence (`command_1 | command_2 | .... |command_N`).

   Listing 2.1: Linux pipeline example.

```
1  ls -l | more
```

   In the bioinformatics context the term *pipeline* is used to describe a sequence of commands where an output from a command can be used as input of any other later command. Figure 2.1 shows fragments of a bioinformatic *pipeline*, taken from paper [27].

Figure 2.1: Bioinformatic pipeline fragments



2. **Workflow** - is a *pipeline*, non-linear, that supports loops and branches.

3. **Command** - is an instruction given to an application to execute a kind of function. An example of a command is `ls` on Listing 2.1.

4. **Task** and **Job** - are used to refer a command in execution time.

5. **Step** - is a term used to refer a command or another *pipeline* within a *pipeline*. In Figure 2.1 each box represents a step within the *pipeline*.

6. **Tool** - is an application composed by one or more commands. On next Section 2.2 each step of the case study is a tool.

7. **Parameter** - refers a definition each part of data that is provide to a command (in *NGSPipesV2*).

8. **Argument** - refers a definition each part of data that is provide to a command (in *NGSPipesV1*).

The term *pipeline* was chosen over *Workflow*, since *pipeline* is most used in bioinformatic field.

2.2 Case studies

Case studies are related to epidemiology vigilance, the data that is used came from the result of applying NGS techniques to genomes of pathogenic agents (e.g. Streptococcus pneumonia). When running a *pipeline* over this data is possible to identify the strain of the specie, the antibiotics which strain is resistant or the presence of a determinate virus.

The *pipeline* applied to each case study corresponds to the execution of the following tools:

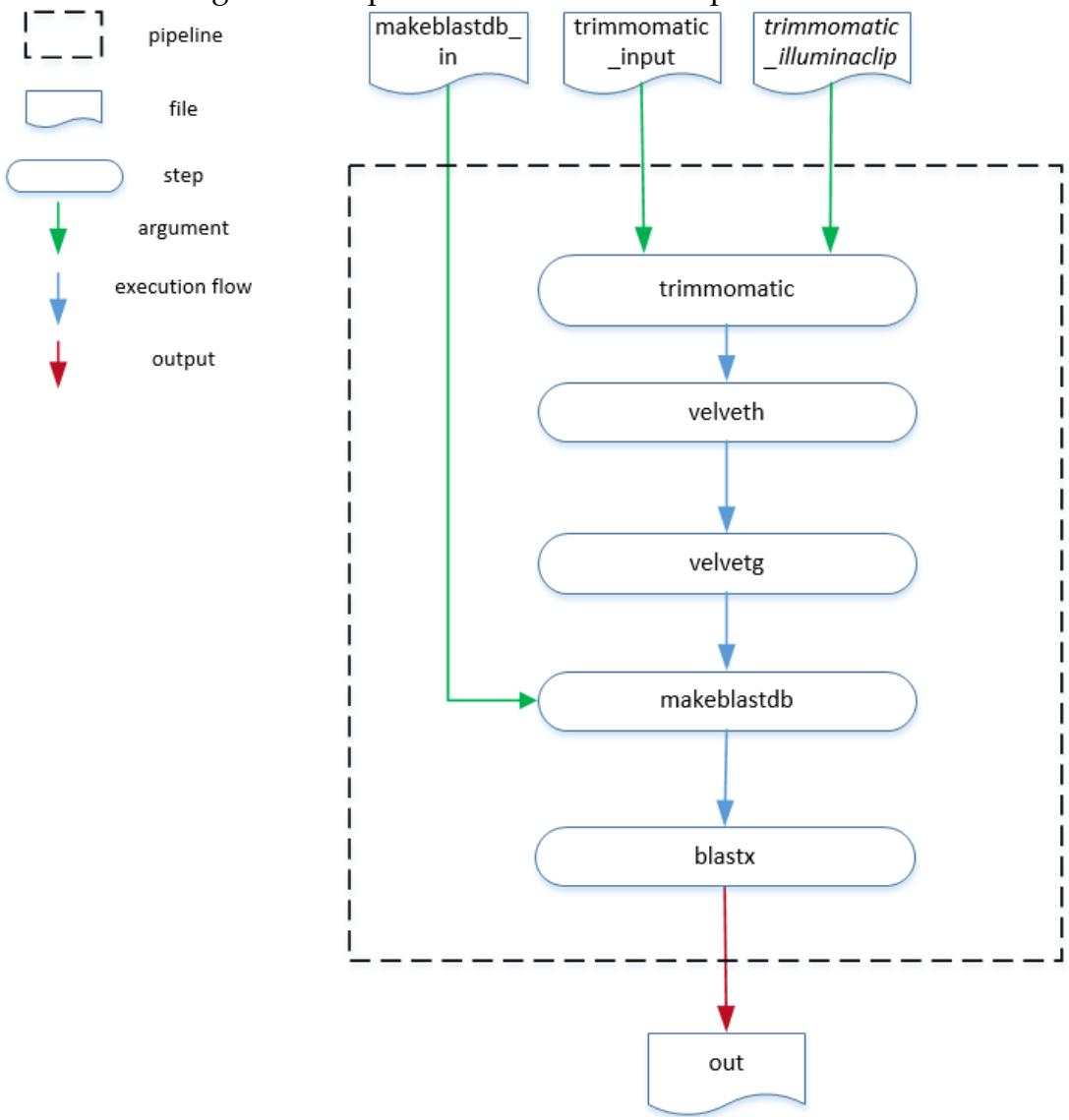
1. *Trimommatic* [6] – works based on files with *reads* containers or small sequences of DNA (FASTQ file type) and allows to remove adapters and positions with low quality. This tool is written in Java;

2. *Velvet* [12] – based on FASTQ files, generally previously preprocessed to remove low quality reads, gets the genome schema. This schema is composed by *contigs* (DNA long sequences from multiple reads). This tool is written in C++;

3. *Blast* [29] – based on *contigs* it determines genes sequences (string) and annotate them by comparing with multiple data bases or compare the genome with multiple databases. This tool is written in C++.

The *pipeline* represented on **Figure 2.2** results from the case study described above and consist in five tasks. *Variant 1* represents a possible sequential execution of *pipeline*.

Figure 2.2: Pipeline variant 1 visual representation.



Each task corresponds to a command, where commands are:

1. *trimmomatic* corresponds to the *Trimomatic* tool. This command recieves a FASTQ[1] file and removes adapters and positions with low quality;
2. *velveth* is one of the commands supplied by *Velvet* tool and, based on a FASTQ file, constructs a dataset consumed by *velvetg* command;
3. *velvetg* is one of the commands supplied by *Velvet* tool. Based on the *velveth* output creates the genome schema;

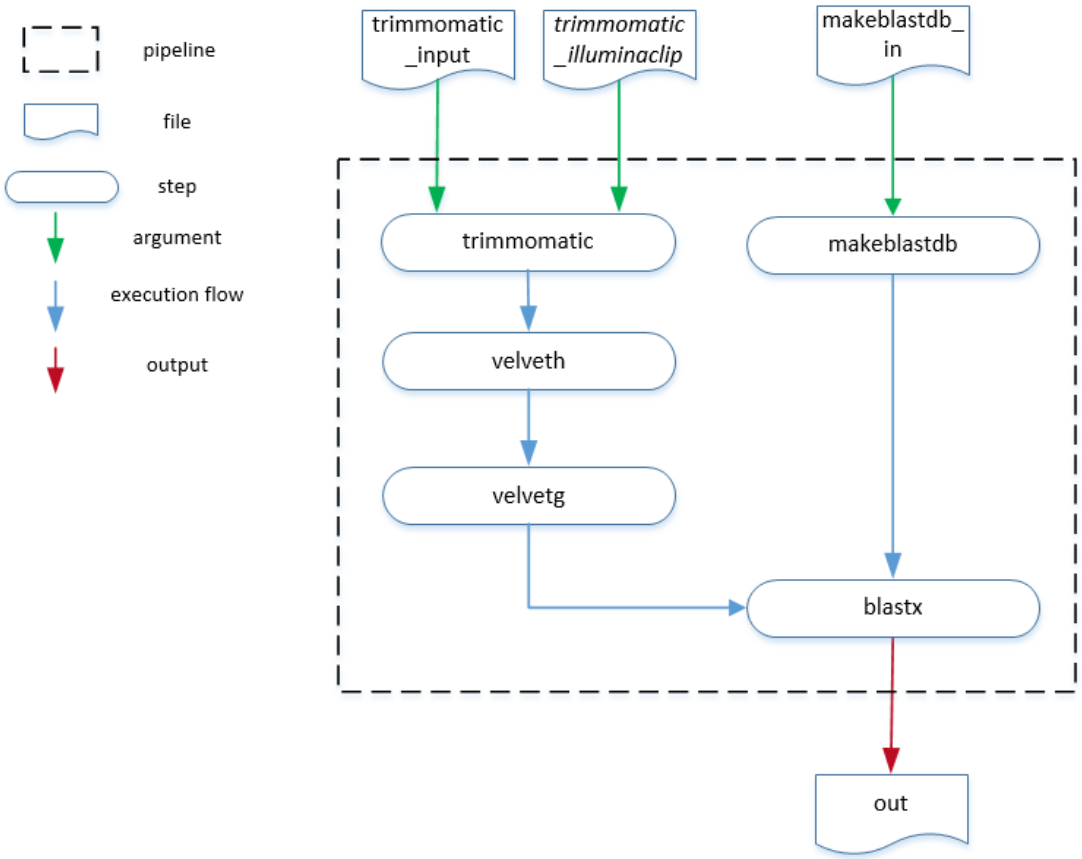
[1]File format used to store a sequence and an associate numeric quality score with each nucleotide in a sequence.

4. *makeblastdb* is one of the commands supplied by *Blast* tool. It's responsible for building a BLAST database.

5. *blastx* is one of the commands supplied by *Blast* tool. This command uses *makeblastdb* and *velvetg* outputs to translate a *nucleotide* query and searches it against protein subject sequences or a protein database.

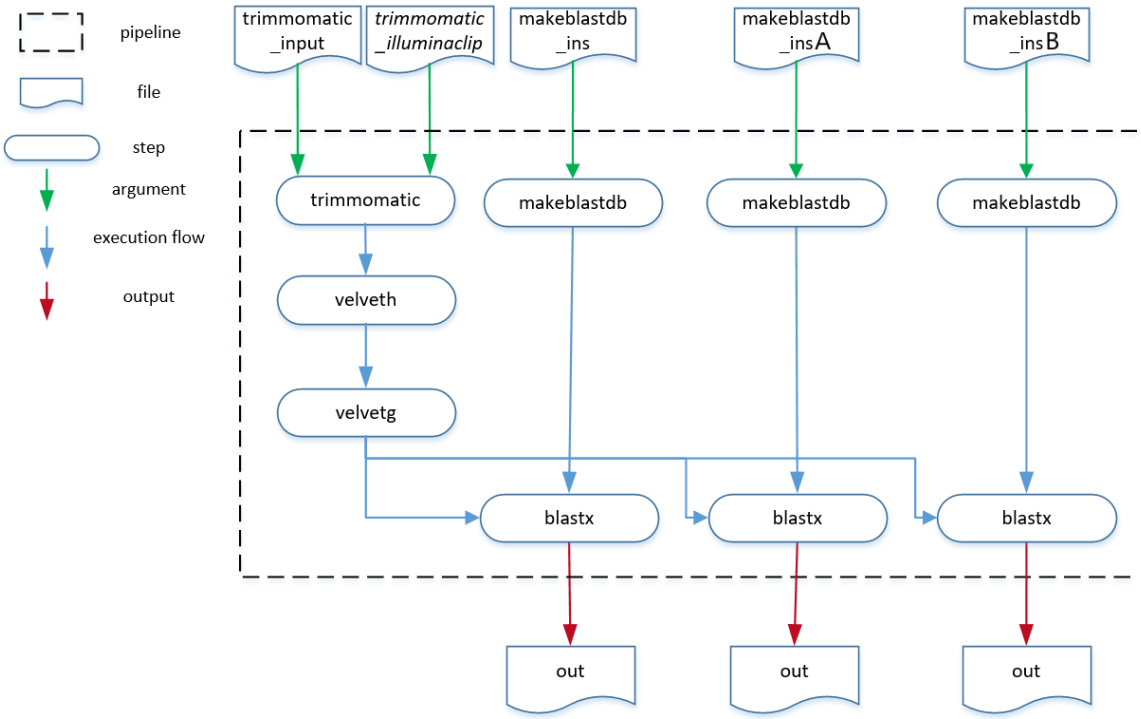
A dependency graph can be generated if the *pipeline* is analyzed from an execution point of view (see **Figure 2.3**). *Variant 2* shows that *pipeline* can be parallelized (which tasks can be executed in parallel) based on dependency graph analysis.

Figure 2.3: Pipeline variant 2, visual representation of execution dependency graph.



To enhance case study, two more databases references (`makeblastdb_inA` and `makeblastdb_inB`) were added to *variant 2*. So, **Figure 2.4**, as *variant 3*, shows the dependency graph of a *pipeline* that will be used in this project. This new variant will represent a *pipeline* that searches subject protein in three different protein databases.

Figure 2.4: Pipeline variant 3, visual representation of execution dependency graph for multiple database files.



Another case study is a nested *pipeline*, which means that one or more steps within a *pipeline* is a *pipeline*. Figure 2.5 shows the dependency graph of this case study as *variant 4*. This variant is based on *variant 2*.

Now that *pipeline* variants (case studies) have been presented, some issues may arise, namely:

- which requirements each tool has?
- how to install each tool?
- what are the parameters of each command?
- which resources each command requires to be executed?
- which execution syntax is used for each command?
- how to manage tools update?
- how to manage parallel execution?
- how to split and join data to apply *data-parallelism*?

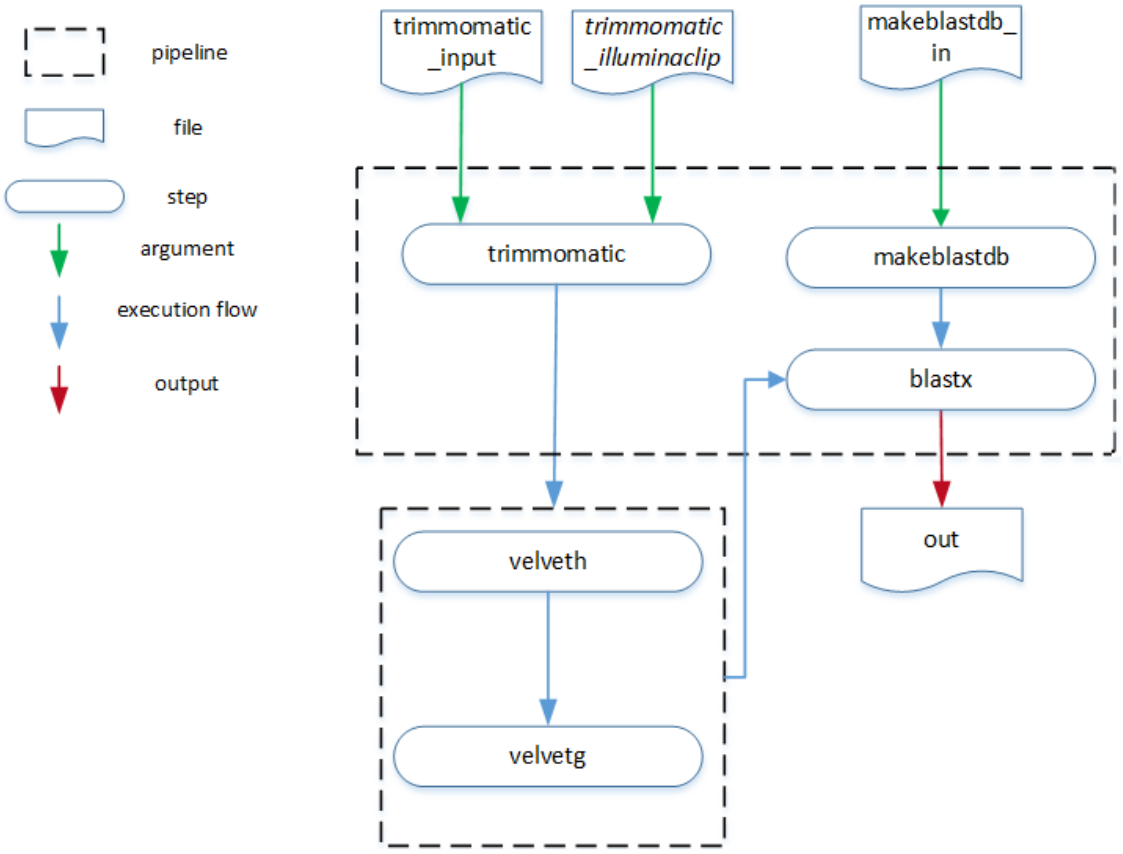


Figure 2.5: Pipeline variant 4, visual representation of execution dependency graph for nested pipeline.

Obviously, a user executing a *pipeline* must know some of these issues, but it isn't easy most of the times. Managing a parallel execution isn't always an easy work to do. To better understand the issues associated to execution of *pipelines*, Chapter 3 describes techniques and approaches that helps to solve these issues.

3

Scientific workflow systems

Scientific Workflow Systems (SWS) are a specialization of Workflow Management Systems (WMS - business workflows). SWSs help to solve complex scientific problems and accelerate scientific progress. These systems can be differentiated by the following characteristics: user usability experience, *pipeline* reproduction, *pipeline* sharing, *pipeline* specification and *pipeline* as a task, *pipeline* control.

There are different ways to create and execute a scientific *pipeline* like:

- use scripts to write *pipeline* description (e.g. write in Perl or bash shell script) and use a command line interface to execute it;
- use UI for *pipeline* creation and execution.

This project will take into account a set of SWSs on characteristics such as *pipelines* execution contexts and tool metadata. These SWSs are:

- *NGSPipesV1* [43] – allows to design and use *pipelines* without users need to configure, install and manage tools, servers and complex pipeline managements systems. It is oriented to users with no or little programming knowledge (eg: scientists) as well as users with programming capability;
- *NGS4Cloud* [2] - is an extension of *NGSPipesV1* to easily produce shareable and reusable *pipeline* descriptions, by executing *pipelines* on cloud infrastructures;

- *Galaxy* [35] - is an open, web-based platform for data intensive biomedical research. Whether on a local public server or on local instance, it is possible to perform, reproduce, and share complete analyses;
- *Ruffus* [17] – is a Python [39] library for writing computational *pipelines*. This solution doesn't support pre-processing (e.g. order inference), so user must code and configure. Also, this solution doesn't support containers technology.
- *Nextflow* [5] - is a fluent DSL modelled around the UNIX pipe concept, that simplifies writing parallel and scalable *pipelines* in a portable manner. Since it doesn't have a user interface this system is more oriented to users with programming knowledge;
- *Swift* [37] – is an open-source project that allows *pipeline* development through an implicitly parallel programming language. Supports distributed execution across clusters, clouds, grids and supercomputers;
- *Cwl-runner* [34] – is a Common Workflow Language [33] (CWL) reference implementation. Cwl-runner consists in a local CWL interpreter written in Python. CWL is a specification to describe command line tools and create *pipelines* by connecting tools through inputs and outputs. Since CWL is a specification, artefacts described using CWL are portable across a variety of platforms that support the CWL standard.

In general, as we will see, not all of these systems support tool metadata, parallel execution or execution in infrastructures. Some of these systems require that tools used within the *pipeline* must be previously installed. The next three sections present a comparison of SWSs based on execution infrastructures, execution and metadata. Comparisons were made based on *NGSPipesV1* system.

3.1 Execution infrastructures

This section discusses the main components of execution infrastructures and how they support distribution of work. These infrastructures are known as job[1] schedulers. Most of this section study is based on article [41][44]. Job schedulers match and execute multiple compute jobs from many users on multiple computational resources.

[1]Job is equivalent to a task

The main components of a job scheduler architecture are:

- *job lifecycle management* – receives jobs and places them in the job queue(s) to wait for execution. The resources requirements for the job (e.g. memory) are requested through the user interface. This component is also responsible for prioritizing and sorting candidate jobs for execution by using the queue management policies;
- *resource management* – keeps availability and resource state information from the compute nodes and makes it available to the scheduler. It also collects jobs progress to store in logs and to make it available to users;
- *scheduling* – allocate resources and assigns the job to the resource(s), when resources are available;
- *job execution* – launch the job on the resources. Once a job is completed it closes the job and sends the job statistics to the job lifecycle management.

Feature comparison is about how schedulers behaves against each other based on these features. This section presents a comparison among a group of schedulers that were study on the paper. In the past last decades, distributed computation has become an area with a big development and with it a great number of schedulers. Since there are a big number of schedulers a selection of them will be studied:

- *Grid Engine [1]* – full-featured and very flexible scheduler;
- *IBM Platform Load Sharing Facility (LSF) [21]* - full-featured and high performing scheduler that is very intuitive to configure and use;
- *Simple Linux Utility for Resource Management (Slurm) [42]* – extremely scalable, full-featured scheduler with a modern multithread core scheduler;
- *Apache Hadoop YARN [15]* – job scheduler that enables to scaling map-reduce [20] jobs environment to execute efficiently on several thousand servers. It was built to improve Apache Hadoop MapReduce scalability and processing speed. It is a monolithic scheduler with a simple API for batch map-reduce jobs, and it does not support micro-jobs or persistent jobs well;

- *Apache Mesos [14]* - is a two-level scheduler that enables the partitioning of a pool of compute nodes among many scheduling domains. Each scheduling domain has a pluggable scheduler called a Mesos framework, and each framework allocates the resources within its domain resources that have been allocated to it by Mesos. It has a rich API for communicating with the scheduler.

Next subsections will describe the behavior of scheduler on features related to: scheduler basics, jobs, scheduling performance and execution.

3.1.1 Scheduler basics

When choosing a framework to work with, some characteristics must be taken in consideration:

- *Actively developed* – this characteristic is important and can be a double-edge sword. A project that is too active, it can be difficult to keep up with the most recent version, but a project also needs to be actively enough to correct bugs and add new features;
- *Cost and licensing* – refers whether the scheduler is open source or has a license for which one must pay;
- *Operating system support* – captures the operating systems on which the scheduler runs and on which jobs can be executed. Theses schedulers run all on Linux but distribution support may not be entirely universal. Commercial schedulers may have a more limited list of supported Linux distributions;
- *Language constraints* – refers to the programming languages in which executed applications can be written;
- *Access control and security* – involves user authentication and isolation.

These features are compared in Table 3.1. This table shows that all schedulers are being actively developed. While LSF and Grid Engine have a commercial version, the rest are open-source projects and Grid Engine also has an open-source version too. All schedulers support Linux OS, *Slurm* and *Grid Engine* also supports Nix. Since *Mesos* biggest part is written in C++ should be portable to diverse (non-Linux) environments. Except for *YARN*, who strongly supports Java and Python,

schedulers support all languages (Matlab, Java, Python, R and Scala). About access control all schedulers have support.

Table 3.1: Meta-data features comparison. Taken from paper [41]

Metadata	**LSF**	**Slurm**	**Grid Engine**	**YARN**	**Mesos**
Actively developed	yes	yes	yes	yes	yes
Cost/ licensing	€€€	Open source	€€€, Open source	Open source	Open source
OS support	Linux	Linux	Linux, *nix	Linux, *nix	Linux
Language support	All	All	All	Java, Python	All
Access control/ security	yes	yes	yes	yes	yes

3.1.2 Jobs

This subsection describes the schedulers ways to classify and support jobs:

- *Parallel and job array support* – picks up which kind of parallelism scheduler handles: single-process jobs and/or parallel jobs. Single-process jobs are multiple independent process that runs using a single job identifier with different parameters for each process. Parallel jobs each process is launched simultaneously and communicate during the computation;
- *Parallel and array jobs* – indicates whether the job scheduler allows synchronous dependent parallel and/or asynchronously independent parallel (array) jobs;
- *Queue support* – refers whether scheduler supports jobs with different characteristics and resource requirements in separate data structures called queues. On one hand having multiples queues helps to manage jobs with different requirements but having too many can be confusing to user experience.
- *Timesharing* – represents the ability to allocate multiple jobs from one or more users to a single compute node;
- *Dependency and directed acyclic graphs (DAG)* – allows to define execution dependencies between jobs. Dependencies between jobs are intrinsically linked to *pipeline* context, so support for this feature is essential;

- *Prolog/epilog* – captures whether a scheduler allows to execute scripts before and/or after job execution. This feature could include setting up an execution environment or copying data;

- *Restarting* – restarts jobs after is aborted or fails (should be required by the submitter);

- *Preemption* – allows to hibernate low-priority executing jobs to guaranty execute high-priority jobs.

Table 3.2 is a comparison of job support features. *YARN* and *Mesos* do not have incorporate support for tightly coupled parallel jobs. *Mesos* could support parallel jobs if build a specific framework for other batch schedulers (e.g. *Slurm*, *Grid Engine* or *LSF*). All schedulers have queue support. While *YARN* queues are available in the capacity scheduler, in *Mesos* each installed framework can be seen as a queue for different applications and/ or application types. *Mesos* is a meta-scheduler by design and allocates shared resources to other scheduler (framework), remaining schedulers do not support multiple resource managers. Timesharing is a critical feature for schedulers, so a scheduler which can't handle scheduling multiple jobs to a compute node, doesn't have a full feature set. Job dependencies are also essential for job schedulers and almost all schedulers supports them except *Mesos*. *Mesos* can support job dependencies if a plugged-in scheduler framework, such as Chronos [26], Marathon [23] and others.

Table 3.2: Job support features comparison. Based on paper [41]

Job	**LSF**	**Slurm**	**Grid Engine**	**YARN**	**Mesos**
Parallel and array	Both	Both	Both	Array	Array
Queue	yes	yes	yes	yes	yes
Timesharing	yes	yes	yes	yes	yes
DAG	yes	yes	yes	yes	no*
Prolog/ epilog	yes	yes	yes	no	yes
Restarting	yes	yes	yes	yes	yes
Preemption	yes	yes	yes	no	yes

3.1.3 Scheduling performance

In this subsection is described how schedulers warranty jobs availability and scalability.

- *Type (Centralized vs. distributed scheduling)* – captures the architecture used in job scheduling. This feature is related to how the schedulers keep information about jobs that have been submitted and are waiting for execution in queue and jobs that have been dispatched and are currently executing;

- *Scheduler fault tolerance* – is related to the previous feature, refers whether the scheduler and its jobs continue to execute even when occurs a fault on the scheduler;

- *Scalability and throughput* – refer to how many nodes and jobs can be simultaneously schedule;

- *Latency* – scheduler latency involves several factors when using a scheduler: submission, queue management, resource identification, resource selection, resource allocation, job dispatch and job tear down. Latency is difficult to measure because each scheduler has different steps and configurations.

Comparison of scheduling performance features is presented on Table 3.3. As table shows all schedulers except Mesos has a centralized architecture job scheduling. Mesos, by its metascheduling nature, is a distributed scheduler so, each of the framework schedulers are distributed. All schedulers can handle from thousands to hundreds of thousands of simultaneously executing jobs slots. The number of executing slots depends on the design and implementation of the job, the resource data structure and the management algorithms.

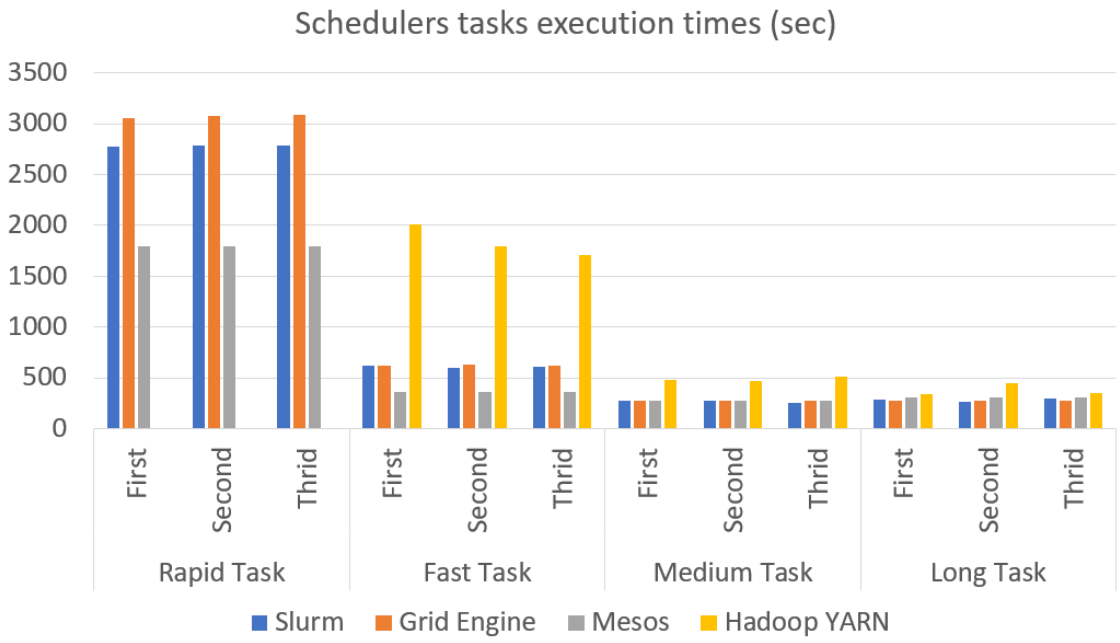
Table 3.3: Job scheduling features comparison. Taken from paper [41]

Job scheduling	**LSF**	**Slurm**	**Grid Engine**	**YARN**	**Mesos**
Type	Cent.	Cent.	Cent.	Cent.	Dist.
Scheduler fault tolerance	yes	yes	yes	yes	yes
Scalability and throughput	10K+	100K+	10K+	10K+	100K+

To represent latency and performance and calculate them, in paper [41] tests were made on a cluster with forty-five nodes: one to serve as scheduler node and forty-four as compute nodes. The total of cores was 1408. All the nodes were connected via a 10 GigE (Ethernet 10 Gigabit) interface. Just four of the schedulers were chosen: *Sun of Grid Engine, Slurm, Mesos* and *Apache Hadoop YARN*. Each one installed

on the scheduler node and compute nodes and were configured to achieve optimal performance on short-duration jobs. Each *pipeline* was run three times for each scheduler and results are presented in graphic within **Figure 3.1**. In *YARN*, Rapid Task trial set was ignored because scheduler latency was excessive. The jobs were all sleep jobs of 1 (Rapid Task), 5 (Fast Task), 30 (Medium Task) or 60 (Long Task) seconds, used as task time (t). All tests were submitted as job arrays to reduce the quantity of introduced scheduler latency. While *Mesos* presents one of the best time results, *YARN* is the worst.

Figure 3.1: Results of performance tests execution for each scheduler. Data taken from paper [41]



3.2 Execution

This section discusses how several aspects related to *pipelines* execution can be as much transparent as possible to the user. The objective is to understand how SWSs get an automated or *semi-automated* execution of *pipelines*. The main challenges regarding *pipeline* execution are:

- *Execution context*- where each task is executed (e.g. locally or in a server);
- *Parallelism*- how to infer which tasks can be parallelize and how to parallelise *pipeline* execution;

- *Tools management*- how to deal with tools used within *pipelines*.

Comparisons were made based on *NGSPipesV1* system.

3.2.1 Pipeline execution context

SWSs commonly support different *pipeline* execution contexts, whether they run on a server or locally (see Table 3.4). As it can be observed *YARN* infrastructures are not supported by any of the systems. *NGS4Cloud* and *Galaxy* are the only systems which don't support local execution. *NGSPipesV1* only supports local execution through a supplied virtual machine (VM).

Table 3.4: Execution contexts supported by each SWS

Systems	**Execution Context**						
	LSF	**Slurm**	**Grid Engine**	**YARN**	**Mesos**	**Galaxy Server**	**Local**
NGSPipesV1	no	no	no	no	no	no	yes
NGS4Cloud	no	no	no	no	yes	no	no
Galaxy	no	yes	no	no	no	yes	no
Ruffus	no	no	yes	no	no	no	yes
Nextflow	yes	yes	yes	no	no	no	yes
Swift	yes	yes	yes	no	no	no	yes
Cwl-runner	no	no	no	no	no	no	yes

3.2.2 Tools management and execution

The execution of a *pipeline* may involve several tools, each one corresponding to a *pipeline* task. The easiest user experience when using a SWS, is to avoid dealing with tools installation. The difference between operating systems, hardware, tools and their dependencies require effort to get a tool ready to use. Also, even when the user has knowledge about how to install, this process takes a considerable time.
Another important aspect is where tools are executed within *pipeline* execution context. Most of SWS supports to execute tools locally and within a container. Table 3.5 presents the support of each SWSs for tool management.

Table 3.5: Contexts supported by each SWS

Systems	**Context**	
	Local	**Container**
NGSPipesV1	no	Docker
NGS4Cloud	no	Docker
Galaxy	yes	Docker, Singularity
Ruffus	yes	-
Nextflow	yes	Docker
Swift	yes	-
Cwl-runner	yes	Docker

Local execution requires to install tools which also means having more "garbage" on user machine. Local execution has other problems like manage tools versions and operating system compatibility. While the other SWSs requires tools to be installed on local context, *NGSPipesV1* supports installing tools dynamically.

Containers technology main advantage is to provide hosting VMs sharing operating system and devices without hypervisor overhead. There are multiple container platforms, the most popular are: Docker [22], Singularity [24] and Shifter [30]. Another advantage of these technologies is the users being able to run tools in similar ecosystem that was used for they development, reducing problems like: system configurations or tools and libraries versions.

Table 3.6 compare *Docker* and *Singularity*, only these two systems were chosen because they are the most used. This comparison between container platforms is based on articles [7] [18] and these systems WEB pages [22] [24]. The features that will be used to compare are:

- *Users group* - describes in which type of users the platform is focused;
- *Access to host file system* - indicates if, within container, the file system of the host machine can be accessed;
- *Privilege model* - related to which kind of permission containers are executed;
- *Images Repository* - feature that allows sharing container images.

Table 3.6: Container systems supported by each SWS

Container systems	Users group	Host file system access	Privilege model	Images repository
Docker	developer	yes	root	yes
Singularity	scientific	yes	user	yes

The main differences between these systems are that *Singularity* focus on single applications or pipelines and *Docker* is more general: microservices, business applications, data science and others.
The host file system access in *Docker* is done by using volumes that user can mount. While *Singularity* by default mounts various directories (*$HOME*, */tmp*, */proc*, */sys*, */dev*) besides system administrator can enable other users to add another folders.
An important issue that involves *Docker* platform is related to privilege model because *docker daemon* requires root access, while an user to run *Singularity* doesn't needs special privileges.

Paper [7] presents a study related to containers performance, where *Docker* demonstrates better results in most of the cases.

3.2.3 Parallelism

On parallelism context there are two types: data parallelism and task parallelism. Data parallelism is whether a SWS is prepared to receive multiple files and each one must be executed within a task. Task Parallelism is when the *pipeline* has tasks that are independent, so they could be executed in parallel. Table 3.7 presents which kind of parallelization each SWS supports.

Table 3.7: Parallelization supported by each SWS

Systems	**Parallelism**	
	Data parallelism	**Task Parallelism**
NGSPipesV1	no	no
NGS4Cloud	no	yes
Galaxy	yes	yes
Ruffus	yes	yes
Nextflow	yes	yes
Swift	yes	yes
Cwl-runner	yes	yes

3.3 Tools metadata

Pipeline execution involves the execution of tools. SWSs use metadata, also known as tool description, of tools to represent all information associated with each tool. Having tool metadata improve system utilization by:

- resources and time used- for example lets imagine gave a wrong value to an argument to a tool. Generally, when a tool begins to execute validates given arguments values and which one is required. Then the result of this execution will be an error and as consequence unnecessary use of resources and time. Having tool metadata helps systems to prevent execution of tools with wrong argument values;
- how to build a tool execution command- with tool's metadata, systems can infer arguments order and the value representation on execution command generation. This characteristic permit to provide a simpler user experience.

A tool metadata must contain its name, commands and the parameters (inputs and outputs) of each description. Nowadays most of SWS use description files to store the metadata of available tools. Tool's metadata ensure that SWSs can validate the correct usage of parameters, resources each task needs and execution context.

Comparisons were made based on *NGSPipesV1* system.

The next three subsections describe tool metadata basics, parameters and execution features.

3.3.1 Metadata

This subsection explains how each SWS supports and manages the tools metadata, based on the follow topics:

- *Support* - refers if a SWSs supports tool metadata. Support can be *Yes* if the system is based on tool metadata and *No* if doesn't use;
- *Addition* - describes how new tool metadata became available for SWS;
- *Addition privilege* - describes which kind of user can add a tool in a SWS, in order to let tool available for pipeline creation;
- *Context* - captures whether a system represent a description of a whole tool or only a command;
- *Versioning* - describes how a SWS manages tool metadata versioning;
- *Portability* - represent with which different SWS a description can be used.

Table 3.8 describes how each SWS behaves for basics tool metadata characteristics and the next subsections presents an explanation for each one.

Next sections will describe each column of Table 3.8, in which *Nextflow*, *Ruffus* and *Swift* systems aren't mentioned because they have the *N/A* (Not Applicable) since have no tool metadata support.

3.3.1.1 Support

One of SWSs goals is to help scientists to focus on development of pipelines, reducing the need of programming knowledge. One of the best ways is offering an UI to create and execute pipelines. Using an UI requires to know the tool metadata in order to supply tool information for the scientists. Tool metadata generally, is stored in a descriptor file within a repository, but could be even in a database system or in memory.

Ruffus, *NextFlow* and *Swift* systems don't use any tool metadata, which requires users to be familiar with the tool's management/ requirements, things like tool arguments and tool execution syntax.

Table 3.8: Systems tool metadata comparison

Systems	Metadata					
	Support	**Addition**	**Addition privilege**	**Context**	**Versioning**	**Portability**
NGSPipesV1	Yes	Publish	Everyone	Tool	Weak	No
NGS4Cloud	Yes	Publish	Admin	Command	Weak	Yes
Galaxy	No	N/A	N/A	N/A	N/A	N/A
Ruffus	No	N/A	N/A	N/A	N/A	N/A
NextFlow	No	N/A	N/A	N/A	N/A	N/A
Swift	No	N/A	N/A	N/A	N/A	N/A
Cwl-runner	Yes	Publish	Everyone	Command	Weak	Yes

NGSPipesV1, *NGS4Cloud*, *Galaxy* and *Cwl-runner* use descriptor files to store the metadata of available tools (*Strong* in Table 3.8). These systems store descriptors files in directory or repository. In *NGSPipesV1* and *NGS4Cloud* systems the descriptors files are supported in JSON or XML languages, in *Galaxy* are written in XML and in Cwl-runner are written in JSON or YAML languages.

A short example of *Velvet* metadata for SWSs that support tool metadata is shown in **Listing 3.1** (for *NGSPipesV1*), **Listing 3.2** (for *NGS4Cloud*), **Listing 3.3** (for *Galaxy*) and **Listing 3.4** (for *Cwl-runner*).

Listing 3.1 shows *NGSPipes* tool descriptor, which corresponds to an entire tool with all included commands.

Listing 3.1: NGSPipesV1 *velvet* descriptor file. Complete example on Appendix A

```
{
  "name" :"Velvet",
  "author" :"Daniel Zerbino[1] ,Ewan Birney",
  "version" :"0.7.01",
  "description" :"Velvet is an algorithm .....",
  "documentation":["https://www.ebi.ac.uk/~zerbino/velvet"],
  "setup" :["make ..." ],
  "requiredMemory": 12288,
  "commands" :[{
    "name" :"velveth",
    "command" :"velveth",
    "description" :"construct the dataset .....",
    "priority" :2,
    "argumentsComposer" :"values_separated_by_space",
    "arguments" :[{
      "name" :"output_directory",
      "argumentType" :"directory",
      "isRequired" :"true",
      "description" :"Directory output files"
      },{
      "name" :"strand_specific",
      "argumentType" :"string",
      "isRequired" :"false",
      "description" :"Strand specific sequencing"
    },... ],
    "outputs" :[{
      "name" :"sequence",
      "outputType" :"directory_dependent",
      "argument_name" :"output_directory",
      "value" :"Sequences"
```

```
    }]},... ]
}
```

In **Listing 3.2** project can be observed that *NGS4Cloud* descriptor is very similar to *NGSPipesV1*. This solution adds some properties to handle the parallel execution (e.g. *tooltype* refers to the type of parallel execution supported by tool). *Velvet* isn't a paralel tool thats why `tooltype` value is `unit`. Property `tooltype` could have values: `unit`, `splitting`, `joinning`, and `listing`.

Listing 3.2: NGS4Cloud *velvet* descriptor file. Complete example on Appendix B

```
{
  "name" :"Velvet",
  "author" :"Daniel Zerbino[1] ,Ewan Birney",
  "version" :"0.7.01",
  "description" :"Velvet is an algorithm .....",
  "documentation":["https://www.ebi.ac.uk/~zerbino/velvet"],
  "setup" :["make ..." ],
  "toolType": "unit",
  "requiredMemory": 12288,
  "recommendedDiskSpace": 1,
  "recommendedCpus": 1,
  "commands" :[{
    "name" :"velveth",
    "command" :"velveth",
    "description" :"construct the dataset .....",
    "priority" :2,
    "argumentsComposer" :"values_separated_by_space",
    "arguments" :[{
      "name" :"output_directory",
      "argumentType" :"directory",
      "isRequired" :"true",
      "description" :"Directory output files"
    },... ],
    "outputs" :[{
        "name" :"sequence",
        "description" :"",
        "outputType" :"directory_dependent",
        "argument_name" :"output_directory",
        "value" :"Sequences"
      },...
    ],
    "inputs": [{
      "name": "",
      "description": "",
      "inputType": "file_dependent",
```

```
    "argument_name": "filename",
    "value": ""
    }]},...]
}
```

In *Galaxy* a descriptor file represents a command within a tool (e.g. command *velveth* of *Velvet*). In **Listing 3.3** is a resumed version of *velveth* command descriptor.

Listing 3.3: Galaxy *velveth* command descriptor. Complete example on Appendix C

```
<tool id="velveth" name="velveth" version="@WRAPPER_VERSION@.0">
  <description>Prepare a dataset .....</description>
  <version_command>velveth 2&gt;&amp;1 | grep "Version" | sed -e 's/
      Version//'</version_command>
  <macros>
    <import>macros.xml</import>
  </macros>
  <expand macro="requirements"/>
  <expand macro="stdio"/>
  <command interpreter="python">
    velveth_wrapper.py
      '$out_file1' '$out_file1.extra_files_path'
      $hash_length
      $strand_specific
      ...
  </command>
  <inputs>
    <param label="Hash Length" name="h_length" type="select" help="k-mer
        length...">
      <option value="19">19</option>
      <option value="21" selected="yes">21</option>
      <option value="27">27</option>
    </param>
    ...
  </inputs>
  <outputs>
    <data format="velvet" name="out_file1" />
  </outputs>
  <requirements>
    <requirement type="package">velvet</requirement>
  </requirements>
</tool>
```

Cwl-runner descriptor file, similar to *Galaxy*, represents a command within a tool (e.g. command *velveth* of *Velvet* tool). In **Listing 3.4** is shown a resumed version

of *velveth* command descriptor.

Listing 3.4: Cwl-runner *velveth* command descriptor. Complete example on Appendix D

```
#!/usr/bin/env cwl-runner
cwlVersion: cwl:v1.0
class: CommandLineTool
baseCommand: velveth
inputs:
  - id: output_directory
  type: string
  inputBinding:
    position: 1
...
outputs:
  - id: output
  type: Directory
  outputBinding:
    glob: $(inputs.output_directory)
...
```

While *NGSPipesV1*, *NGS4Cloud* and *Cwl-runner* descriptors are written from the perspective of using the tools/ commands, *Galaxy* has some primitives and supported values used for the user interface (e.g. on Listing 3.3 para `Hash Length` of type `select`). So, we can say *Galaxy* descriptors are more coupled to user interface.

3.3.1.2 Addition

Since a SWS uses tools, it is necessary a mechanism to add new tools that can surge. This process is different in each SWS. Adding a tool's metadata to a SWS can involve:

- Publish (in Table 3.8) - use of a component which contains the available tools metadata (e.g. *NGSPipesV1*, *NGS4Cloud*, *Galaxy* and *Cwl-runner*). Tools metadata is published on a repository or directory;
- Recompile - for tools metadata within system's code, metadata addition can imply recompile. Although this is a valid strategy is not practical to recompile the whole solution every time we want to add a new tool. For this reason we didn't find any SWS with this strategy.

Galaxy uses repositories to store tools metadata. *Galaxy* users can add new tools to these repositories or add tools from another repository (e.g. a *ToolShed* [36]). A *Galaxy* tool repository keeps available tools metadata in a root folder named *tools*. This folder is composed of subfolders, each one is named with the respective tool name. Each tool folder contains different files as descriptors files, tool dependencies, macros, etc. Each descriptor file is written in XML and corresponds to a command inside the current tool. For example, if current tool is *Velvet*, them the descriptors files inside *Velvet* folder will be *velvetg.xml* and *velvetg.xml*.

1. The steps to add a custom tool are:
   - add tool directory to *Galaxy* instance on `tools` directory;
   - register new tool into *Galaxy* instance on file *tool_conf.xml* (see Listing 3.5)

     Listing 3.5: Galaxy *velvet* custom tool addition

     ```
     <section name="velveth" id="velveth">
       <tool file="velvet/velveth.xml" />
     </section>
     <section name="velvetg" id="velvetg">
       <tool file="velvet/velvetg.xml" />
     </section>
     ```

2. The steps to add a tool that exists in another tool repository (e.g. a *ToolShed* [36]) are:
   - open Galaxy Admin Interface;
   - open the tool shed;
   - search for a tool;
   - select the wanted tool from the search results to install it;
   - confirm dependency installation.

NGSPipesV1 and *NGS4Cloud* tools metadata are stored into a repository. The root of repository has a *Tools.json* file with all available tools name and multiple directories, each one with the name of tool that it represents. Inside each tool directory there is a descriptor file, configurators list file, tool logo and a file for each configurator a tool has associated. Tool addition is done by adding tool metadata directory into repository and adding the tool name into *Tools.json* file (see Listing 3.6).

Listing 3.6: NGSPipesV1 *velvet* tool addition

```
{
  "toolsName": ["Blast", "Bowtie2", "Picard", "SAMTools", "Trimmomatic"
     , "Velvet" ]
}
```

3.3.1.3 Addition privilege

Defining tool metadata can be done by everyone, the problem is that to use it, it must be added to the system. Some SWSs have permission restrictions in who can add tools to the system. These restrictions could give more security, in order to avoid intrusive code, but also limits system usability. These restrictions are most used on systems that allow code within tool metadata files.

Cwl-runner allows any user to define a tool (*Everyone* in Table 3.8). This approach of not-restriction is conditioned to the tool addition process into the system that requires user have permission to install the tool into the target execution system.

NGSPipesV1 has no restrictions for tool definition, some restriction may occur if repository (e.g. Github) have edition permission associated. However, each user can define is own repository.

Galaxy system requires admin privileges in order to add a tool, as we saw in last subsection.

3.3.1.4 Context

Context characteristic describes, in each SWS that supports tool descriptor, what root information has every descriptor. In Table 3.1 exist two values:

- *Tool* meaning descriptor file contains all tool metadata;
- *Command* meaning descriptor file only contains one command metadata.

While *Tool* approach can be more verbose, in *Command* approach a metadata of a tool could be divided and confuse to understand which commands are within a tool.

3.3.1.5 Versioning

A new tool's metadata versioning can be related to an error correction (eg. argument type was *string* and passed to *flag*) or a new version of the tool (eg. adding a new command). This work considers versioning support as:

- *None* - a SWS that doesn't support tool metadata versioning;
- *Weak* (in Table 3.8) - a SWS that supports tool metadata versioning but depends on human supervision or has some restriction in order to apply a new version of tool metadata;
- *Strong* - a SWS that supports automatically tool metadata versioning.

NGSPipesV1 and *NGS4Cloud* systems support tool metadata but is human supervision dependent for tool metadata versioning. The most critical case is when argument type is updated:

Imagine *Velvet* tool, that was added to our system previously and now has a new version. This new version change argument `strand_specific` (in **Listing 3.1**) type from `string` to `flag`. At this point a compatible problem occurs because the tool metadata is updated, and thus existing *pipelines* enter in conflict. We can work around this problem if we create in the repository a new entry for the tool like Velvet_new, but we would be repeating information.

Galaxy system is also human supervision dependent. It also has a restriction for adding a new version of tool metadata. This restriction is that user must be *Galaxy* instance admin. This system allows the revision of metadata, using the `id` property of the tool metadata. Since they have a descriptor file for each command, the duplication of information is minimal.

Cwl-runner also depends of human supervision.

3.3.1.6 Portability

Tool description portability between SWSs is one of the requisites that some solutions try to solve. We could imagine two different users, *A* and *B*. Suppose *A* wants to use a tool that *B* has been used. Imagine *B* is using *Galaxy* and *A* is using one of the SWSs that has *None* tool metadata portability (see Table 3.8), then the

problem is they can't share the same tool metadata, so the solution will require *A* to create the tool descriptor.

As Table 3.8 shows that only *Galaxy* and *Cwl-runner* permit some portability. So, both systems have some tool metadata portability.

Galaxy and *Cwl-runner* are portable between them when using *argparse2tool* [40].

3.3.2 Inputs and outputs

Inputs and outputs within a command can be described by name, description, type and mandatory. In this subsection is described how each SWS supports inputs and outputs characteristics (structure, types, mandatory, order, inputs dependency) within a metadata description. Table 3.9 presents a resume of this characteristics for each SWSs. Since *Ruffus*, *NextFlow* and *Swift* don't support tool metadata description, they are irrelevant to discuss in this subsection. The most common metadata associated to inputs and outputs are:

- *Primitives* - refers how a tool description commands inputs and outputs are organized and its primitives. For example, systems usually separate inputs from outputs;
- *Types* - describes supported inputs types (e.g. *int*, *string*, *file*). This characteristic helps to validate tool usage;
- *Mandatory* - describes whether input can be annotated as mandatory. This characteristic helps to validate tool usage;
- *Order* - describes the order in which the input should be passed on the command execution line (e.g. a property position or descriptor order);
- *Builder* - describes if has properties to express how inputs are built to be included within command's execution line;
- *Dependency* - refers how is expressed dependency between inputs and outputs (e.g. an input `a` can be specified only if `b` is used). Three types of dependency is considered: `input -> input`, `output -> input` and `output -> output`. This characteristic helps to validate tool usage.

Table 3.9: Input/ output tool metadata comparison

Systems	**Inputs / Outputs**					
	Primitives	**Types**	**Mandatory**	**Order**	**Dependency**	**Builder**
NGSPipesV1	Arguments, Outputs	Basic	yes	Implicit	Not all	yes
NGS4Cloud	Arguments, Inputs, Outputs	Basic	yes	Implicit	Not all	yes
Galaxy	Inputs, Outputs	Complex	yes	Explicit	All	no
Cwl-runner	Inputs, Outputs	Complex	yes	Explicit, Implicit	Not all	yes

3.3.2.1 Primitives

As observed on Table 3.9 all SWSs have similar *Primitives*. The *Primitives* Arguments/ Inputs and Outputs make sense since is a concept within a tool that a commands have inputs and outputs.

A definition of inputs and outputs is presented on **Listing 3.1** (for *NGSPipesV1*), **Listing 3.2** (for *NGS4Cloud*), **Listing 3.3** (for *Galaxy*) and **Listing 3.4** (for *Cwl-runner*).

NGS4Cloud has a particular structure, it uses inputs to identify easily what are the arguments that are files in order to copy them for command execution directory.

3.3.2.2 Types

In Table 3.9 the type support for inputs and outputs within tool metadata is classified as:

- *Basic*- refers a set of types based on the common primitives types of most languages, namely: `text` or `string`, `integer` or `int`, `float`, `boolean`, `double`, `float` and `files`;
- *Complex*- includes *Basic* and some other types like: `null`, `directory`, `enum` and `array`.

Galaxy besides complex types also supports: `genomebuild`, `baseurl`, `ftpfile`, `data`, `data_collection`, `library_data` and `drill_down`. Types associated to user interface are `select`, `color`, `data_column`, `hidden`, `hidden_data`.

Cwl-runner besides complex types also supports: `CWLType`, `CommandInput-RecordSchema`, `CommandInputEnumSchema`, `CommandInputArraySchema` and `array` of previous complex types.

NGSPipesV1 system only supports basic.

3.3.2.3 Mandatory

Mandatory characteristic refers if SWSs tool metadata supports specifying whether inputs / arguments are mandatory or not. This primitive helps to validate if all required inputs were specified before running a command, this away we can manage the available resources more efficiently.

NGSPipesV1, *NGS4Cloud* and *Galaxy* systems support this characteristic, *Galaxy* supplies the attribute `optional`, *NGSPipesV1* and *NGS4Cloud* through `isRequired` property. *Cwl-runner* doesn't have a specific property, so all inputs are mandatory by default. In order to make an input optional a `?` character must follow the input type (see Listing 3.7).

Listing 3.7: Optional example for Cwl-runner

```
...
 - id: input_file
 type: File?
...
```

3.3.2.4 Order

When a command is going to be executed the order in which inputs and outputs are used is important. Table 3.9 values to classify how each SWS supports order specification are:

- *Explicit* - when it is used a property or attribute to specify order;
- *Implicit* - when order is implicitly inferred.

Cwl-runner has a optional primitive (`position`) that allows to specify the order of the inputs (see Listing 3.8). If this property isn't specified, the order is inferred from the order in which each input was written on the tool descriptor.

Listing 3.8: Cwl-runner *position* property example

```
...
inputs:
  - id: output_directory
  type: string
  inputBinding:
    position: 1
...
```

In *Galaxy* descriptor is mandatory to specify the complete execution command, so, inputs and outputs order is *Explicit*.

In *NGSPipesV1* and *NGS4Cloud* cases the order is inferred by the order on which each argument is written on the tools descriptor.

3.3.2.5 Dependency

This section describes three types of dependencies between inputs and outputs:

- input that depends on another input;
- output that depends on an input;
- output that depends on an output.

Knowledge of dependencies within a command can help to achieve better execution time and avoid use of resources. When a SWS is asked to execute a *pipeline* before executing it must exist inputs and outputs validations. Let's imagine some scenarios:

1. An input is mandatory when another is specified. In command *trimmomatic* `mode` input can have two possible values (*ES- single end* or *PE- paired end*). When `mode` value is *PE* then it takes two inputs files arguments (`paired input 1` and `paired input 2`) otherwise just `inputFile` (see Listing E.1);
2. A command output depends on an input value (see Listing 3.4). In command *velveth* `output_directory` input of type *string* represents the name of the directory where command outputs are stored and output `output`, of type *directory*, value will dependent of `output_directory`.
3. A command output depends on another output value (see Listing 3.4). As explained in previous scenario, command *velveth* stores outputs in a directory. A possible representation is an output `out_dir` of type *directory* and then the other outputs depending of it (e.g. `sequence->out_dir`).

NGSPipesV1 and *NGS4Cloud* only support the second scenario. Examples of the support of `output->input` dependency can be seen on Listing 3.1 (*line 27*) where *velveth* command has an output (`sequence`) that depends on the input `output_directory` and Listing 3.9 (*line 8*) where *trimmomatic* command has an output `outputFile` which depends on `outputFile` (`argument_name` property) input.

Listing 3.9: NGSPipesV1 *trimmomatic* output -> argument dependency example. Taken from Appendix E.1

```
1 {
```

```
  "name": "Trimmomatic",
  ...
  "commands": [ {
    "name": "trimmomatic",
    ...
    "outputs": [ {
      "name": "outputFile",
      "outputType": "file_dependent",
      "argument_name": "outputFile"
    }, ...]
  }]
}
```

Galaxy ensures all considered dependencies. For the first scenario, in Listing 3.10 (*line 4*), tool *trimmomatic* has a `conditional` attribute that depends on the selected value for input `readtype`, it can require one or more input files. Second and third scenarios are supported through attribute `filter`. As example, output `fastq_out` depends of the input `readtype` value `readtype.fastq_in.name` (see Listing 3.10 in *line 21*). It is important to note that Galaxy supports these dependencies introducing verbosity and making user, who is writing tool metadata, know the necessary code to make the filter.

Listing 3.10: Galaxy *trimmomatic* dependency reduced example. Taken from Listing F.1

```
<tool id="trimmomatic" name="Trimmomatic" version="0.36.5">
  ...
  <inputs>
  <conditional name="readtype">
    <param name="se_or_pe" type="select" label="SE or PE reads?">
      <option value="se" selected="true">SE</option>
      <option value="pair_of_files">PE</option>
    </param>
    <when value="se">
      <param name="fastq_in" type="data" format="fastqsanger,
   fastqsanger.gz" label="Input FASTQ file" />
    </when>
    <when value="pair_of_files">
      <param name="fastq_r1_in" type="data" format="fastqsanger,
   fastqsanger.gz"
      label="Input FASTQ file (R1/first of pair)" />
      <param name="fastq_r2_in" type="data" format="fastqsanger,
   fastqsanger.gz"
      label="Input FASTQ file (R2/second of pair)" />
    </when>
```

```
18   </conditional>
19   ...
20   <outputs>
21     <data name="fastq_out" label="${tool.name} on ${readtype.fastq_in.
     name}" format_source="fastq_in">
22       <filter>readtype['se_or_pe'] == 'se'</filter>
23     </data>
24     <data name="fastq_out_r1_pe" label="${tool.name} on ${readtype.
     fastq_r1_in.name} (R1 paired)" format_source="fastq_r1_in">
25       <filter>readtype['se_or_pe'] == "pair_of_files"</filter>
26     </data>
27     <data name="fastq_out_r2_pe" label="${tool.name} on ${readtype.
     fastq_r2_in.name} (R2 paired)" format_source="fastq_r2_in">
28       <filter>readtype['se_or_pe'] == "pair_of_files"</filter>
29     </data>
30   </outputs>
31 </tool>
```

Cwl-runner system only supports second scenario (`output -> input`). For example in Listing 3.4 (*line 12*), where *velveth* command has an output `output` depending on input `output_directory` value. Another example is in Listing 3.11 (*line 6*), where *trimmomatic* command has an output `reads1_trimmed` dependent on input `reads1` (`inputs.reads1.nameroot`).

Listing 3.11: Cwl-runner *trimmomatic* reduced dependency example. Taken from Appendix G.1

```
1  #!/usr/bin/env cwl-runner
2  cwlVersion: v1.0
3  class: CommandLineTool
4  ...
5  outputs:
6    reads1_trimmed:
7      type: File
8      format: edam:format_1930  # fastq
9      outputBinding:
10     glob: $(inputs.reads1.nameroot).trimmed.fastq
11 ...
```

3.3.2.6 Builder

This topic refers to which kind of metadata SWSs have to define how an input should be written, in order to appear on command execution line. Each tool has

a different specification about how the inputs must be written. Some examples can be:

- build input and its value separated by a character, for example *makeblastdb* execution command (`makeblastdb -out=/out/allrefs -dbtype=prot -in=/inputs/allrefs.fna.pro -title=allrefs`) where inputs names and their values are separated by `equal` character;
- build just input value;
- a mix of the two previous points. For example, *velveth* execution command (`velveth /out/velvetdir 21 -fastq /out/ERR40-6040.fastq`) where `-fastq` is the input name separated by `space` character from its value and the rest are the other inputs value.

NGSPipesV1 and *NGS4Cloud* use the `argumentComposer` property. This approach isn't good enough because `argumentComposer` value is limited to be one of the composers supplied or one developed and added programmatically to solution's library. When there is a new format to write inputs, isn't a good practice make the user to add it, these systems were classified as *Weak*. Listing 3.12 shows an example of `argumentComposer` property.

Listing 3.12: NGSPipesV1 *velveth* argument composer example. Taken from Appendix A.1

```
{
  "name": "Velvet",
  ...
  "commands": [{
    "name": "velveth",
    "argumentsComposer" : "values_separated_by_space",
    ...}, ...]
}
```

Galaxy system already has the command written on descriptor, with wildcards for inputs value, then this aspect is already treated and doesn't require a dedicated property.

Cwl-runner by default only writes the input value, but it provides optional properties like `prefix` or `valueFrom` (eg. Listing 3.13).

Listing 3.13: Cwl-runner *trimmomatic* builder metadata examples. Taken from Appendix G.1

```
#!/usr/bin/env cwl-runner
cwlVersion: v1.0
class: CommandLineTool
...
inputs:
  phred:
  type: trimmomatic-phred.yaml#phred?
  inputBinding:
    prefix: -phred
    separate: false
    position: 4
  doc: |
  "33" or "64" specifies the base quality encoding. Default: 64
  maxinfo:
  type: trimmomatic-max_info.yaml#maxinfo?
  inputBinding:
    position: 15
    valueFrom: |
    ${ if ( self ) {
      return "MAXINFO:" + self.targetLength + ":" + self.strictness;
    } else {
      return self;
    }}
...
```

3.3.3 Execution

Tool's metatada also can include some information related to execution configuration, like:

- *Resources* - refers which resources properties (eg. recommended cpu, disk or memory) are supported within a descriptor;
- *Context* - describes if execution context is associated to tool metadata.

Table 3.10 presents a resumed comparison of SWSs execution supported characteristics that will be discussed with more detail in next subsections.

Table 3.10: Execution tool metadata comparison

Systems	Execution	
	Resources	**Context**
NGSPipesV1	memory	yes
NGS4Cloud	all	yes
Galaxy	all	yes
Cwl-runner	all	yes

3.3.3.1 Resources

Describes if and which kind of metadata each SWS has support for required resources.

Unlike *NGSPipesV1* that only supports `requiredMemory`, *NGS4Cloud* supports various execution resources properties. In Listing 3.14 are shown the three properties supported by this system, where:

- `requiredMemory` represents the total amount of memory required by *Velvet* tool in MB (megabytes);
- `recommendedDiskSpace` represents a possible amount of disk space required by *Velvet* tool in GB (gigabytes);
- `recommendedCpus` represents the total amount of processors required by *Velvet* tool.

Listing 3.14: NGS4Cloud *velvet* execution resources reduced example

```
{
  "name" : "Velvet",
  ...
  "requiredMemory": 12288,
  "recommendedDiskSpace": 1,
  "recommendedCpus": 1,
...
}
```

Galaxy system supports resources through runtime properties as environment variables, some of those examples are:

- {$GALAXY_SLOTS: -4}- Number of cores/threads allocated by the job runner or resource manager for the given job (here 4 is the default number of threads to use);

- $GALAXY_MEMORY_MB- Total amount of memory in MB allocated by the administrator for the given job. If it's unset, tools should not attempt to limit memory usage;
- $GALAXY_MEMORY_MB_PER_SLOT- Amount of memory in MB to be used per core allocated by the administrator for the given job. If it's unset, tools should not attempt to limit memory slot usage.

An example of the usage of these properties is shown on Listing 3.15.

Listing 3.15: Galaxy *trimmomatic* execution resources reduced example. Taken from Appendix F.1

```
<tool id="trimmomatic" name="Trimmomatic" version="0.36.5">
...
<command detect_errors="aggressive"><![CDATA[
  @CONDA_TRIMMOMATIC_JAR_PATH@ &&
  @CONDA_TRIMMOMATIC_ADAPTERS_PATH@ &&
  #if $readtype.se_or_pe == "pair_of_files"
    ...
  #end if
  java \${_JAVA_OPTIONS:--Xmx8G} -jar \$TRIMMOMATIC_JAR_PATH/
   trimmomatic.jar
  #if $readtype.se_or_pe in ["pair_of_files","collection"]
    PE -threads \${GALAXY_SLOTS:-6} -phred33
    fastq_r1.'$r1_ext' fastq_r2.'$r2_ext'
    fastq_out_r1_paired.'$r1_ext' fastq_out_r1_unpaired.'$r1_ext'
    fastq_out_r2_paired.'$r2_ext' fastq_out_r2_unpaired.'$r2_ext'
  #else
    SE -threads \${GALAXY_SLOTS:-6} -phred33 fastq_in.'$fastq_in.
   extension' fastq_out.'$fastq_in.extension'
  #end if
...
</tool>
```

Cwl-runner supports a variety of resources configurations within a `ResourceRequirement` property:

- `coresMin`– minimum reserved number of CPU cores;
- `coresMax`– maximum reserved number of CPU cores;
- `ramMin`– minimum reserved RAM in MB;
- `ramMax`– maximum reserved RAM in MB;

- `tmpdirMin`– minimum reserved storage for temporary directory, in MB;
- `tmpdirMax`– maximum reserved storage for temporary directory, in MB;
- `outdirMin`– minimum reserved storage for output directory, in MB;
- `outdirMax`– maximum reserved storage for output directory, in MB.

Listing 3.16 presents an example where is specified the minimum of ram and the minimum of cores to use.

Listing 3.16: Cwl-runner *trimmomatic* execution resources reduced example. Taken from Appendix G.1

```
#!/usr/bin/env cwl-runner
...
  requirements:
    ResourceRequirement:
      ramMin: 10240
      coresMin: 8
...
```

Systems that allow *pipeline* parallel execution must support these characteristics, ensuring a better resources management.

3.3.3.2 Context

In Section 3.2.2, Table 3.5 presents which contexts are supported by each SWS. This section explains how each SWS associate's contexts to tool's metadata. Execution context contains the instructions needed to have tool available to execute.

NGSPipesV1 and *NGS4Cloud* systems objective is to leave the execution context of tools decoupled from the descriptor file. This strategy makes the descriptor file less verbose. These systems consider that, from tool's developer point of view, this information is not essential because same description could be used for different contexts. Listing 3.17 present an example of docker context file for *trimmomatic*.

Listing 3.17: NGSPipesV1 *trimmomatic* docker context example.

```
{
  "name" : "DockerConfig",
  "builder" : "Docker",
  "uri" : "ngspipes/trimmomatic0.33",
  "setup" : [
```

```
    "sudo apt-get install -y docker.io"
  ]
}
```

Both *Galaxy* and *Cwl-runner* descriptors are committed to the execution context. Listing 3.18 shows a *Galaxy* example and Listing 3.19 shows a *Cwl-runner* example. These system's default contexts are *Galaxy Server* for *Galaxy* and *localhost* for *Cwl-runner*.

Listing 3.18: Galaxy *trimmomatic* docker context example.

```
<tool id="trimmomatic (docker)" name="Trimmomatic" version="0.36.5">
  <requirements>
    <container type="docker">ngspipes/trimmomatic0.33</container>
  </requirements>
  ...
</tool>
```

Listing 3.19: Cwl-runner *Trimmomatic* docker context reduced example.

```
#!/usr/bin/env cwl-runner
...
requirements:
  - class: DockerRequirement
    dockerPull: ngspipes/trimmomatic0.33
...
```

3.4 Final remarks

The objective of this chapter was, not only to study SWS features, but also to understand the main limitations of *NGSPipesV1*. Globally these issues are not supporting parallelism and a lack of the tool's metadata annotation expressiveness.

In order to support parallelism this project will supply a new execution graph inference implementation and two ways to execute: on local machine and on a cluster (by using an infrastructure). Based on the study of infrastructures, *Mesos* was the chosen infrastructure for remote execution because:

- is an open source solution;
- presents a high scalability;
- was one of the best performances on tests;

- is used on platforms like: Twitter, Ebay and Netflix.

Tool's metadata annotation expressiveness will be refactored by adding primitives to cover more dependencies scenarios (Section 3.3.2.5), more input types (eg. `flag` and `array`) and resource information (Section 3.3.3.1). Without this support in tool's metadata, less validations can be made, and errors will only be raised during *pipeline* command execution. Considering a scenario where the command isn't the first one, the error will appear after executing one or more commands, causing unnecessary resources allocation and waste of time.

4

Architecture

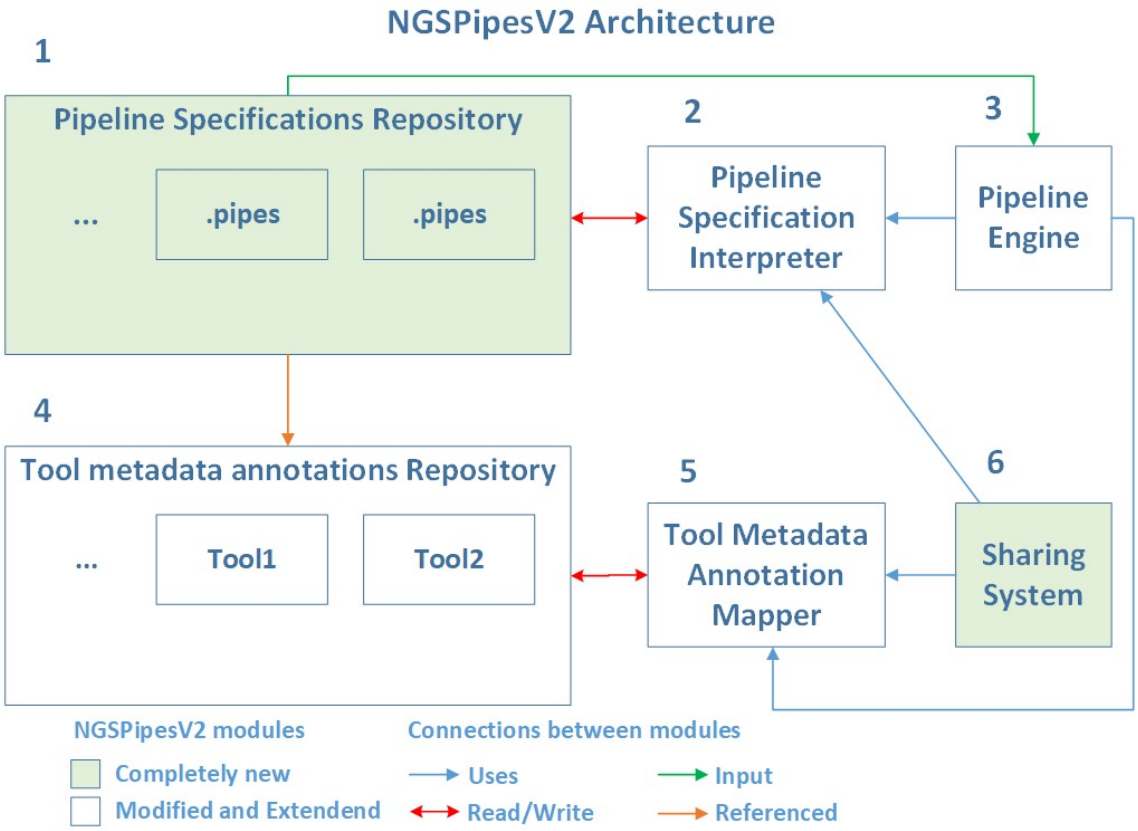
This chapter presents the evolution of *NGSPipes* architecures. *NGSPipesV1* was evolved to *NGSPipesV2* resulting on a new architecture due to the transformations and new characteristics added with the development of this new version.

Figure 4.1 shows a macro visualization of modules within the approach and how they communicate with each other.

The main modules of this architecture are:

1. *Pipeline Specifications Repository-* represents the repository containing *pipelines* specification files;
2. *Pipeline Specifications Interpreter-* converts *pipelines* written with *NGSPipesV2* Domain Specific Language (DSL) [3] to Java objects;
3. *Pipeline Engine-* executes *pipelines*;
4. *Tool metadata annotations repository-* represents the repository containing tools metadata files;
5. *Tool metadata annotations mapper-* maps tools metadata annotations into Java objects based on JSON or YAML format;
6. *Sharing System-* supplies a WEB application which permits *NGSPipesV2*'s users to share *pipelines* specification and tools metadata.

Figure 4.1: Macro architecture



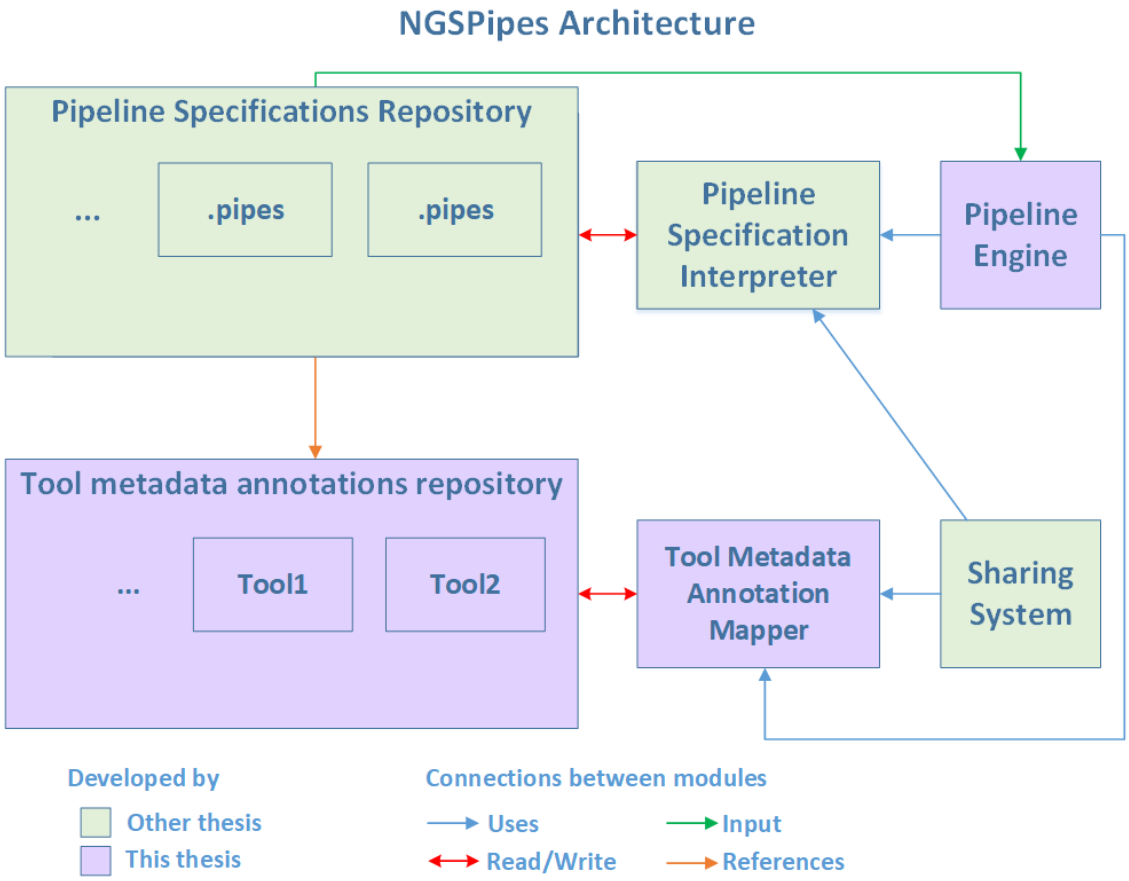
Along with this project it was being developed another project thesis, by Bruno Dantas "Bioinformatic pipeline specification language and sharing system" [10], which is responsible for the development of modules *Pipeline Specifications Interpreter* and *Pipeline Specifications Repository*. Figure 4.2 depicts all modules within *NGSPipesV2* solution, as well as the project that is responsible for each module.

Section 4.1 describes tool metadata annotations repository and tool metadata annotation mapper. Section 4.2 explains the pipeline engine.

4.1 Tool metadata annotation repository and mapper

This section describes the tool metadata annotation modules (repository and mapper). These modules are essential to accomplish the solution main goal, because they help to validate the correct usage of *pipeline* parts (steps/ commands, inputs and outputs). These modules provide the resource information each command

Figure 4.2: Architecture modules



requires execute and infer how execution commands are built. Subsection 4.1.1 describes *NGSPipesV1* limitations and *NGSPipesV2* is covered on Subsection 4.1.2.

4.1.1 NGSPipesV1 metadata

NGSPipesV1 has some tool metadata restrictions. To explain some of these limitations a possible concretization of command *trimmomatic* will be take in consideration and can be seen on Listing 4.1.

Listing 4.1: Example of a possible concretization command for *trimmomatic*.

```
1 java -jar trimmomatic-0.33.jar PE -phred33 in_forward.fq.gz in_reverse.
    fq.gz out_forward_paired.fq.gz out_forward_unpaired.fq.gz out_
    reverse_paired.fq.gz out_reverse_unpaired.fq.gz ILLUMINACLIP
    :TruSeq3-PE.fa:2:30:10 SLIDINGWINDOW:4:15 LEADING:3 TRAILING:3
    MINLEN:36
```

The main limitations of *NGSPipesV1* are:

1. **Types supported:** for example, value `-pdhred33` represents an input of type `flag`. In this version, the closest supported type is `string`. Because of that an error while executing the tool is more probable to occour (e.g. if given value isn't known by the tool). Another aspect that can be mentioned is that in this case, user will need to put value `pdhred33`. So, by supporting more types, the system can validate if the value given to an input is compatible with its type;

2. **How to build execution command:** this version uses a property named `argument_composer` to specify how command's arguments are written whithin execution command. This project supplies already some implementations (see Listing 4.2). The main limitation is that it's value can only be an already implemented `ArgumentsComposer`. For example, commands like *trimmomatic* (see Listing 4.1), which has more than one way to write the arguments: outputs arguments only are represented by their values (e.g. `out_forward_paired.fq.gz` and `out_forward_unpaired.fq.gz`), while `MINLEN` is writed by `{name}:{value}`. Commands like this require the implementation of `IArgumentComposer` interface and recompiling the solution.

3. **Composed argument:** *trimmomatic* has also a particular kind of argument (eg. `ILLUMINACLIP` formed as `ILLUMINACLIP:<fastaWith AdaptersEtc>:<seed mismatches>:<palindrome clip threshold>:<simple clip threshold>` [6]). In this version tool descriptor consider `ILLUMNACLIP` as a simple argument where user must complete the value `/inputs/adapters/TruSeq3-SE.fa:2:30:10`, this is a problem from the point of view of validating the correct usage of arguments.

4. **Dependency:** another aspect that helps to validate correct usage of arguments is to support dependencies like `arguments->argument` and `outputs->arguments`. This version supports `outputs->argument` dependency. For example, in Listing 3.1 *line 28* and *29* output `sequence` depends on argument `output_directory`. An example that justifies the support for dependency `arguments->argument` could be explained through *trimmomatic* in Listing 4.1. When input `mode` has `PE` value then the command requires to define input files `pairedInput1` and `pairedInput2` (eg. `in_forward.fq.gz in_reverse.fq.gz`).

Listing 4.2: Argument composers supplied by *NGSPipesV1*.

```
1. dummy -> []
2. valuesSeparatedBySpace -> [value value value]
3. nameValuesSeparatedByEqual -> [name=value name=value name=value]
4. nameValuesSeparatedByColon -> [name:value name:value name:value]
5. nameValuesSeparatedByHyphen -> [name-value name-value name-value]
6. nameValuesSeparatedBySpace -> [name value name value name value]
7. valuesSeparatedByColon -> [value:value:value]
8. valuesSeparatedByVerticalBar -> [value|value|value]
9. valuesSeparatedByHyphen -> [value-value-value]
10. valuesSeparatedBySlash -> [value/value/value]
11. valuesSeparatedByComma -> [value,value,value]
12. trimmomatic -> [TRIMMOMATIC STYLE ArgCategory:arg:arg:arg]
13. velvetG -> [VELVETG STYLE all arguments has format [name value]
    except output_directory that has format [value]]
```

4.1.2 NGSPipesV2 metadata

Based on the study made for Section 3.3.1 and on previous version limitations (Section 4.1.1), it was decided to change and add some properties to *NGSPipesV2* tool metadata. Next subsections describe each modification.

4.1.2.1 Renamed properties

Renamed properties to improve descriptor semantic were:

- `arguments` property name, which became `parameters` because when declaring a functionality or method is always used the parameter term, argument is used for the concretization of a parameter;
- `requiredMemory` property was renamed by `recommended_mem`;
- `argument_type` property name, which became `type`, since this property only appears within the argument (now parameter) scope;
- `isRequired` property name, which became `required`;
- `output_type` property name, which became `type`, since this property only appears within the output scope.

4.1.2.2 Moved properties

Property `setup` on tool descriptor contained the instructions to be executed in order to install the tool. This property is directly related to the semantic of execution contexts and became a representation of local execution context because:

- *descriptor shouldn't be compromised with operating system* - a tool could have different ways of installation depending on the operating system in which will be installed;
- *not used*- execution contexts based on containers technology don't need to know how to install a tool.

4.1.2.3 Merged properties

This subsection describes the properties that were merged because two were used to refer the same aspect. There were two properties that represents this case, `argument_name` and `value`, both used to describe the dependency between outputs and parameter. Property `value` on *NGSPipesV1* has two possible values:

- an empty string, for example Listing 4.3 describes output (`sequence`) which depends on `output_directory` parameter value. In this case the resulting value for output will be a concatenation of parameter value with output value;
- a non-empty string, for example in Appendix E, Listing E.1 *line 144* describes an output (`outputFile`) which depends on `outputFile` parameter value. In this case the resulting value for output will only be the parameter value.

Listing 4.3: Example for case empty string

```
1 ...
2 "outputs": [ {
3 "name": "sequence",
4 "outputType" : "directory_dependent",
5 "argument_name": "output_directory"
6 "value": "Sequences"
7 } ... ]
8 ...
```

The *NGSPipesV2* approach has a property `value` to represent the previous two. When an output depends on a parameter, then the value of this output must contain the character `$` followed by the parameter name. Listing 4.4 *line 5* depicts an example for empty string and for non-empty string see Listing 4.4 *line 9*.

Listing 4.4: Example for merged property

```
1  ...
2  "outputs": [ {
3  "name": "outputFile",
4  "type" : "file",
5  "value": "$output"
6  }, {
7  "name": "sequence",
8  "type" : "file",
9  "value": "$output_directory/Sequences"
10 } ... ]
11 ...
```

4.1.2.4 Added properties

To improve tool's annotation expressiveness and support new metadata, some properties were added:

- within the same tool, different commands could require different execution resources, so to the `command` property it was added the following properties:
  - `recommended_mem`, which is the total amount of memory that command recommends using, in MB;
  - `recommended_cpu`, which is the number of processors that command recommends using;
  - `recommended_disk`, which is the total amount of disk that command recommends using, in GB;
- complex types are now supported (e.g. type `enum`, `flag`). When a *parameter* type is `enum` a new property (`values`) appears containing the `enum` possible values (see Listing 5.1, *line 17*);
- `depends` property introduces dependency between inputs (see Listing 5.1 *line 23*). To support a more complex dependency it was added the property

`dependent_values` which allows to specify what values activates the dependency;

- to improve flexibility when building tool execution's command, it was removed `argument_composer` and added two properties `prefix` and `suffix` (see Listing 5.1 *line 35*);
- property `sub_parameters` was added to contain child parameters of a parameter of type `composed` (see Listing 5.1 *line 37*);
- property `separator` was added to specify how each sub-parameter value is separated within composed parameter (see Listing 5.1 *line 36*);

4.1.3 Tool repository

Tools repository is a module that contains all the information related to the available tools which can be used when defining a pipeline. *NGSPipesV2* solution provides a repository prototype that contains some tools to test our system, which can be found at `https://github.com/ngspipes2/tools_support`. User made repositories can be used, as it will be explained in this section. This component has to supply the following information:

- a list of tools names;
- a list of tool descriptors (see Subsection 5.3);
- a logo URL for each tool (optional);
- a list of execution contexts of a given tool.

Users can create their own repositories and use them with this project. In order to use their repositories, a repository must follow the structure explained above and information should be stored on type of repositories supported namely, local (on system files), *Github* or *Server* (supplying information via HTTP request) repositories.

4.2 Pipeline engine

This section describes the approach for *pipeline* execution. The idea is to provide a solution that allows to schedule *pipelines* with a semi-automated execution. This behaviour allows users to decide where they want to execute a *pipeline*:

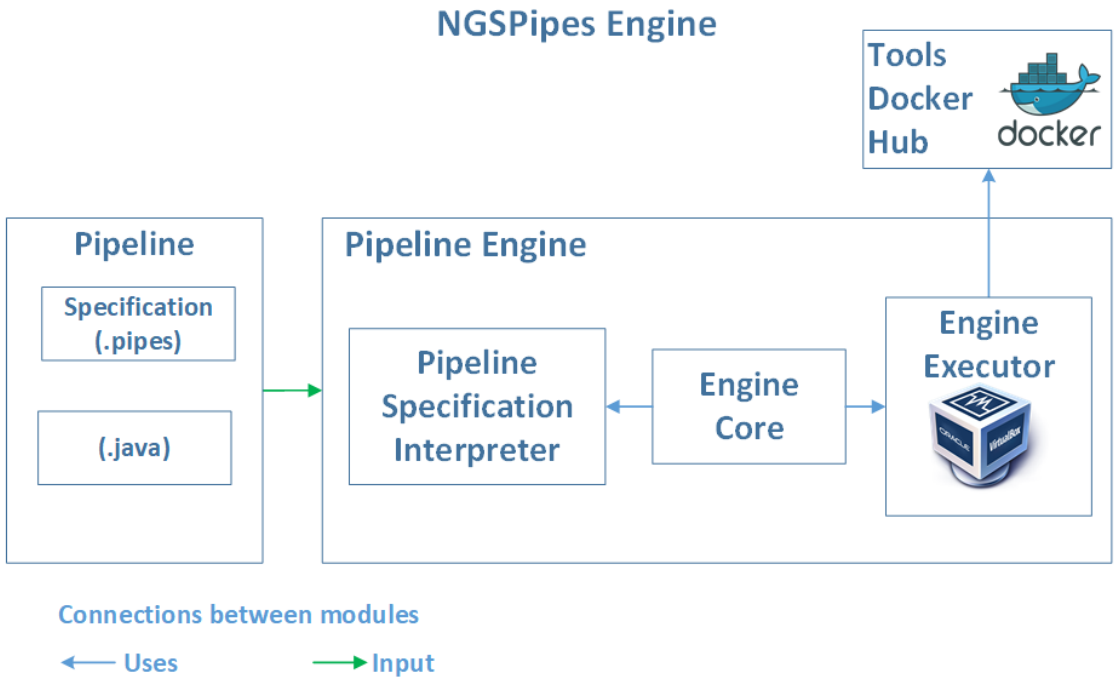
- *locally* - *pipelines* that don't require a lot of resources could be executed on the user local machine;
- *cluster* - for *pipelines* requiring higher resources utilization.

The next two sections will describe *NGSPipesV1 Engine* and *NGSPipesV2 Engine* main differences.

4.2.1 NGSPipesV1 engine

NGSPipesV1 Engine only supported a local execution through a VM. Engine previous version architecture is presented on Figure 4.3.

Figure 4.3: NGSPipesV1 engine architecture



Engine receives a file, which can be either *.pipes* or *.java*. *Pipeline* processing in this engine involves the following sequence of steps:

- if file is *.pipes*, it is submitted to an interpretation process and converted to a Java object. If file is *.java*, this step is not necessary and the file is passed directly to next step;
- a Java object is embedded in an application and a compiled Java file is created;

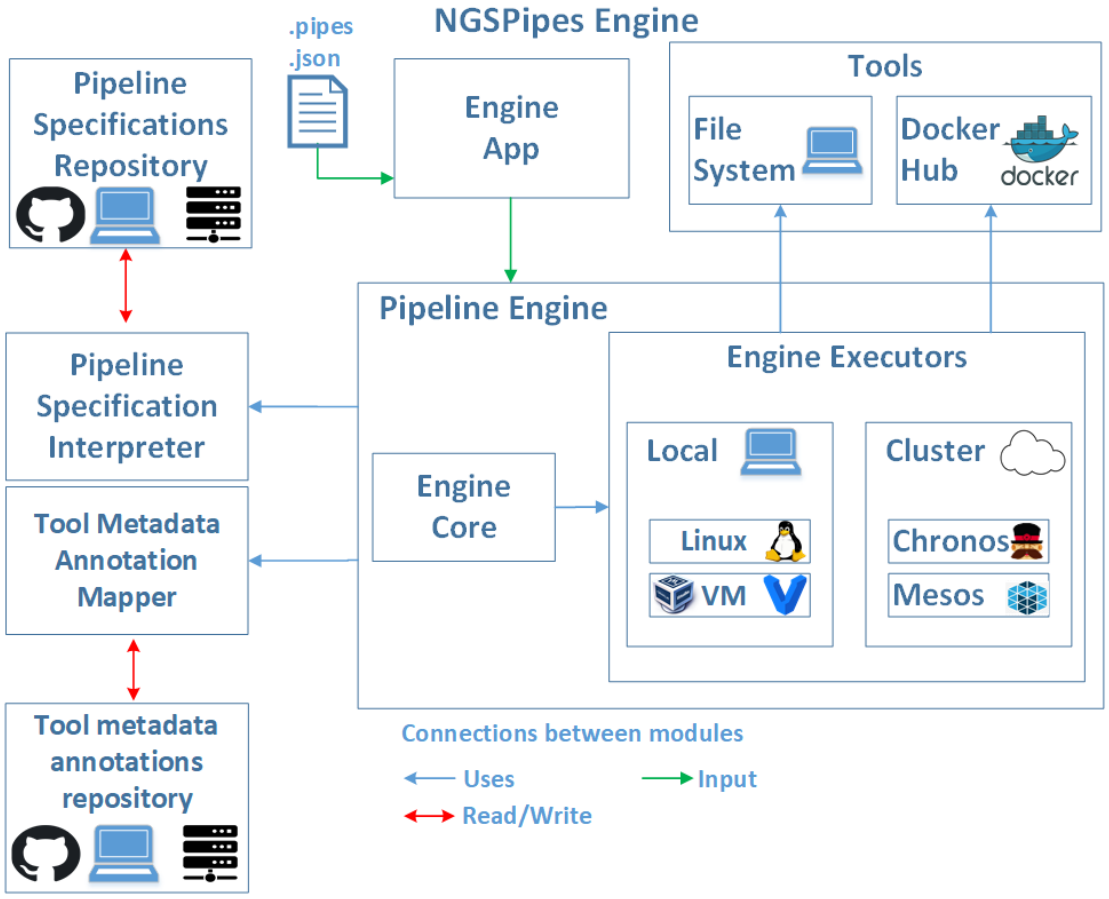
- a clone of the VM is created for the *pipeline* execution;
- engine collects resources information and configure the VM (set the base memory, add share folders);
- VM is started and through an initiation script Java compiled file, which is within a shared folder, is executed using *Docker* containers.

The main limitation of this version is that executes *pipelines* only sequentially and only executes on local machine.

4.2.2 NGSPipesV2 engine

NGSPipesV2 engine supports the parallel execution of *pipelines*. Figure 4.4 shows the resulting architecture, of *NGSPipesV2* engine.

Figure 4.4: NGSPipesV2 engine architecture



It all starts when an user submits a *pipeline* to the engine. The engine executes a series of steps where each one runs depending if its predecessors executed successfully. The series of steps are:

1. *Engine App*
    - the application receives a file that could be `.pipes` (*NGSPipes pipeline* specification) or .json (*pipeline* intermediate representation);
    - convert *pipeline* specification to `Pipeline` Java object and pass it to `Engine Core`. Case `.pipes` file *pipeline* specification is interpreted, using module *Pipeline Specifications Interpreter*;
2. *Engine Core*
    - receives the object and validates the *pipeline* by verifying it isn't cyclic and that its repositories and outputs aren't duplicated;
    - creates an intermediate representation object to uncouple *pipeline* execution from its specification and tool metadata. During the object creation the correct usage of each tool is validated (e.g. parameters type, if parameter is required and its dependencies);
    - creates an execution graph based on dependencies between *pipeline* steps;
    - sends the intermediate object to execute within the executor;
3. *Engine Executor*
    - copies *pipeline* initial inputs to the working directory;
    - starts executing the *pipeline*, based on execution graph;
    - if the Engine Executor is a Vagrant Executor, it creates the *vagrantFile* and initiates *Vagrant* [19] VM with the resources required by the *pipeline* (eg. memory RAM). The others Engine Executors omits this step because the execution context already exists.

4.2.2.1 Intermediate representation

An intermediate representation is used to support interoperability of *NGSPipesV2 Engine* with other SWSs. This representation guaranties a *pipeline* specification closer to its execution, meaning that it contains all necessary information to execute a *pipeline*. Listing 4.5 shows an excerpt of a possible specification of *pipeline*

variant 1 (see Chapter 2), where information like step command and outputs can't be found because they are within respective tool descriptor. Listing 4.6 presents the corresponding intermediate representation, in JSON, for the previous specification of *pipeline* excerpt.

Listing 4.5: *Pipeline variant 1* trimmomatic step specification.

```
Properties: {
  author: "NGSPipes Team"
  description: "Study case 1"
  version:  "1.0"
  documentation: ["http://ngspipes.readthedocs.io/en/..."]
}
Repositories: [
  ToolRepository repo: {
    location: "...\\tools_support"
  }
]
Outputs: {
  output1: trimmomatic[outputFile]
}
Steps: [
  Step trimmomatic: {
    exec: repo[Trimmomatic][trimmomatic]
    execution_context: "DockerConfig"
    inputs: {
      mode: "SE"
      inputFile1: "...\\ERR4060.fastq"
      output: "ERR4060.filtered.fastq"
      fastaWithAdaptersEtc: "...\\TruSeq3-SE.fa"
      seedMismatches: 2
      ...} }, ... ]
```

Listing 4.6: *Pipeline variant 1* trimmomatic step intermediate representation.

```
{
  "name" : "_1554652157043",
  "outputs" : [ {
    "name" : "output1",
    "type" : "file",
    "value" : "ERR4060.filtered.fastq",
    "originJob" : "trimmomatic" }],
  "jobs" : [ {
    "parents": [],
    "id" : "trimmomatic",
    "inputs" : [ {
```

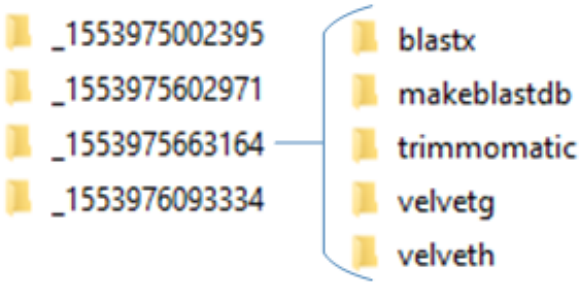
```
12          "name" : "mode",
13          "type" : "enum",
14          "originStep" : "trimmomatic",
15          "value" : "SE",
16        }, {
17          "name" : "inputFile1",
18          "type" : "file",
19          "originStep" : "trimmomatic",
20          "value" : "...\\ERR4060.fastq",
21        }, {
22        "name" : "ILLUMINA_CLIP",
23        "type" : "composed",
24        "originStep" : "trimmomatic",
25        "prefix" : "ILLUMINACLIP:",
26        "separator" : ":",
27        "subInputs" : [ {
28            "name" : "fastaWithAdaptersEtc",
29            "type" : "file",
30            "originStep" : "trimmomatic",
31            "value" : "...\\TruSeq3-SE.fa", }, ...]
32      }],
33      "outputs" : [ {
34        "name" : "outputFile",
35        "type" : "file",
36        "value" : "ERR4060.filtered.fastq"
37      }],
38      "executionContext" : {
39        "name" : "DockerConfig",
40        "context" : "Docker",
41        "config" : {
42          "uri" : "ngspipes/trimmomatic0.33",
43          "tag" : "latest"
44      }},
45      "command" : "java -jar /trimmomatic-0.33.jar"},
46      "environment" : {
47        "workDirectory" : "path/for/working/directory",
48        "outputsDirectory" : "path/for/outputs/directory",
49        "memory" : 1024,
50        "cpu" : 2
51      }, ...] }
```

As Listing 4.6 describes, all the necessary information to execute the *pipeline* was collected and added to this intermediate representation. This way the engine executors can execute *pipelines* provided by different SWSs or even a simple person, since those *pipelines* are written using this representation.

4.2.2.2 Execution data structure

An important issue during *pipeline* execution is how files are handled and organized. This solution creates, within the working directory, a directory for each *pipeline*. Within each *pipeline* directory is also created a folder for each job. Each job folder will contain its inputs and outputs. The inputs with origin on other jobs output is also copied to folder of the dependent job. This strategy guarantees that each job executes on an isolated context. Figure 4.5 presents the data structure for *pipeline variant 1* (see Chapter 2). At the left each folder represents a *pipeline* folder, its name is composed by *pipeline* name (on the example is empty), the character _ and a creation date string value.

Figure 4.5: Pipeline data structuration.



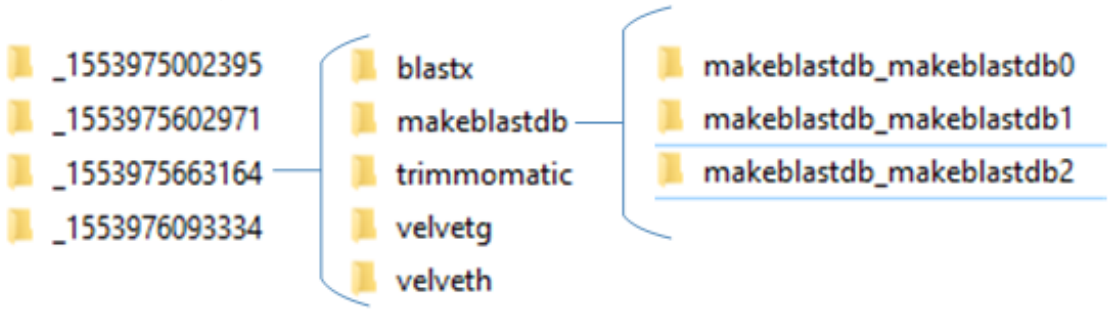
There is a special case of data manipulation related to data parallelism jobs, since these jobs will create multiple sub jobs. A possible specification for *pipeline variant 3* (see Chapter 2) data parallelism is depicted on Listing 4.7.

Listing 4.7: *Pipeline variant 3* specification for data parallelism.

```
1  Step makeblastdb: {
2    exec: repo[Blast][makeblastdb]
3    execution_context: "DockerConfig"
4    inputs: {
5      dbtype: "prot"
6      out: ["allrefs", "allrefsB", "allrefsC"]
7      title: ["allrefs", "allrefsB", "allrefsC"]
8      in: ["...\\allrefs.fna.pro", "...\\allrefs.fnaB.pro","...\\allrefs.
       fnaC.pro"]
9    }
10   spread: {
11     inputs_to_spread: [in, out, title]
12     strategy: one_to_one(in, one_to_one(out, title))
13   }
14 }
```

Engine maps `makeblastdb` step into three jobs and Figure 4.6 presents the correspondent data structure. At the left each folder represents a *pipeline* folder, its name is composed by *pipeline* name, the character _ and a creation date string value.

Figure 4.6: Pipeline data structuration for variant 3.



4.3 Final Remarks

On this chapter it was presented and explained changes made to *NGSPipes V1* architecture. The changes were made based on the limitations mentioned in this chapter and the purpose of this solution: support parallelism on *pipelines* execution and improve tool's metadata annotation expressiveness.
An intermediate representation of *pipeline* was added as a way for *NGSPipes V2 Engine* have interoperability with others SWSs.

5

Implementation

This chapter describes the solution proposed to extend *NGSPipesV1* by adding the following characteristics:

1. task parallelism to engine module;
2. multi-core parallelism to engine module;
3. data parallelism to engine module;
4. automation of execution's command construction to tool metadata modules;
5. composed parameters to tool metadata modules;
6. dependency between parameters and outputs to tool metadata modules;

This chapter is divided in three sections: *(i)* architecture; *(ii)* tool metadata annotation repository and mapper; *(iii)* pipeline engine.

The implementation code can be found at NGSPipesV2 repository [28].

5.1 Technologies

This section briefly presents the technologies used in each component: languages and tools/applications that were involve in this project development.

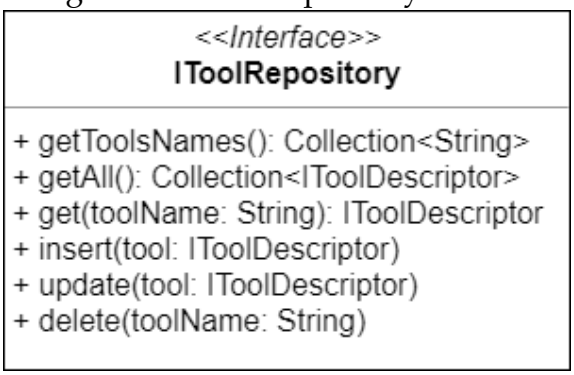
1. **Tool metadata annotation mapper**
   (a) **Java** - language used to convert tool metadata annotation content to Java objects.
2. **Tool metadata annotations repository**
   (a) *JSON* - supported format by *Tool metadata annotation mapper*;
   (b) *YAML* - supported format by *Tool metadata annotation mapper*.
3. **Tools execution**
   (a) **Docker** - supported by *Engine executors* to executed tools;
   (b) **File System** - supported by *Engine executors* to executed tools.
4. ***Pipeline* engine**
   (a) **Engine core**
      i. **Java** - language used to supply module implementation.
   (b) **Engine executors**
      i. **Java** - language used to supply all module implementation;
      ii. **Local executor implementation**
         - **VirtualBox** - application used to run, through a VM, engine on local users computer;
         - **Vagrant** - application used to manage (e.g. create, run and delete) local approach VMs;
      iii. **Cluster executor implementation**
         - **Mesos** - infrastructure used to manage executed *pipeline* jobs on the cluster;
         - **Chronos** - application used to schedule *pipeline* jobs on *Mesos* cluster.

5.2 Tool metadata annotation mapper

Tool metadata annotation mapper module is a Java library to manipulate tool metadata annotations within a tool repository. There are repository implementations to manage metadata stored on: Local (file system), *GitHub* repositories and

Server (e.g. API). Each repository implementation supports `IToolRepository` contract (see Figure 5.1).

Figure 5.1: IToolRepository contract



5.3 Tool metadata annotation

Listing 5.1 shows *NGSPipesV2* tool description for *Trimmomatic* where `description` property has been removed from `command`, `parameters` and `outputs` properties to make the example shorter. Next subsection will cover in detail all changed and added properties.

Tool metadata annotations repository module, which was explained in section 4.1.3, represent tools metadata storage.

Listing 5.1: NGSPipesV2 *trimmomatic* descriptor file. Complete example on Appendix H.

```
{
  "name": "Trimmomatic",
  "author": "UNKNOWN",
  "version": "0.0.33",
  "description": "It works with FASTQ ...",
  "documentation": ["http://www.usadellab.org/cms/?page=trimmomatic"],
  "commands": [ {
    "name": "trimmomatic",
    "command": "java -jar trimmomatic-0.33.jar",
    "recommended_mem": 1024,
    "recommended_cpu" : 1,
    "recommended_disk" : 1024,
    "parameters": [ {
      "name": "mode",
      "type": "enum",
      "required": true,
```

```
17        "values" : ["SE", "PE"]
18      }, {
19        "name": "inputFile1",
20        "type" : "file",
21        "required": true,
22        "depends" : "$mode",
23        "dependent_values" : ["SE"]
24      }, {
25        "name": "basein",
26        "type": "flag",
27        "required": true,
28        "prefix" : "-basein",
29        "depends" : "$mode",
30        "dependent_values" : ["PE", "org.usadellab.trimmomatic.
    TrimmomaticPE"]
31      }, {
32        "name" : "ILLUMINA_CLIP",
33        "type" : "composed",
34        "required" : false,
35        "prefix" : "ILLUMINA_CLIP:",
36        "separator" : ":",
37        "sub_parameters": [
38        {
39          "name": "seedMismatches",
40          "type": "int",
41          "required": true,
42        } ...
43    ] } ],
44    "outputs": [
45      {
46        "name": "outputFile",
47        "type" : "file",
48        "value": "$output"
49      } ... ]
50  } } ] }
```

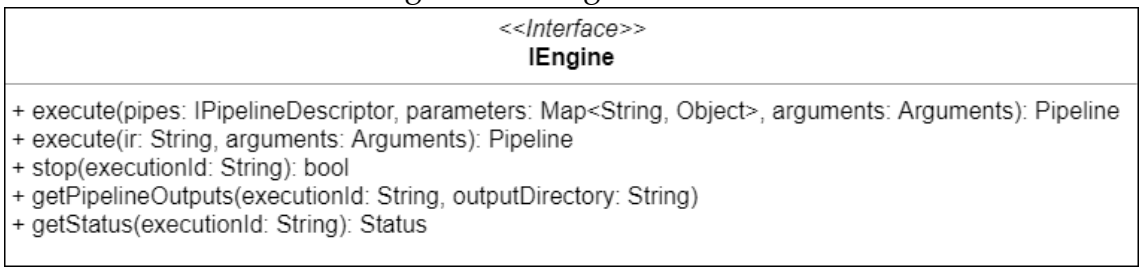
This new version of tool metadata annotation results from the extension that was made, by moving, merging or adding properties, see Section 4.1.2.

5.4 Pipeline engine

This section describes the engine implementation. Figure 5.2 shows interface `IEngine` used to schedule *pipelines* execution. This interface has two methods

to schedule an execution. One method receives a *pipeline* specification as object (`Pipeline`). The other one receives a `String` which represents the *pipeline* as an intermediate representation (see Section 4.2.2.1). This last method approach is a strategy to uncouple engine from the rest of *NGSPipes* modules and a way to gain interoperability.

Figure 5.2: IEngine contract



Using always a cluster to execute a *pipeline* could be a waste of time and resources. For example, if the *pipeline* to execute requires low resources usage and local machine have them available, it would be much better to execute locally because the time for uploading inputs and downloading outputs would have less latency. Considering the previous described *pipeline* as a sequence of tasks to be executed on cluster, the scenario could get even worse.

5.4.1 Execution graph inference

Pipeline execution graph is used by *Engine Executor* module to execute jobs based on graph composition. This graph is created on *Core* module to generalize the pipeline interpretation and avoid each executor to have this responsibility. The graph generation algorithm involves two dependency types between jobs: direct and conditioned, both specified within *pipeline* description. Conditioned dependency is when a job depends on another that produces an unknown quantity of outputs, so the dependent job needs to wait the parent execution and then execute for each output (e.g. when a parent job is a tool that splits input). A direct dependency is when a job depends on another and can resolve directly its dependencies.

A user can choose between the following executions:

- **sequential**- algorithm do a topological sort based on jobs directly dependent;

- **parallel**- algorithm do a topological sort too, but considers when a conditioned dependency appears and leaves the parent job as an inconclusive leaf. Inconclusive jobs must be detected by executor and when they run, executor must trigger the process of the graph generation from that job and execute the new available jobs.

Figure 5.3 describes an example of a *pipeline* that includes inconclusive jobs. The first resulting execution graph will exclude `C`, `G` and they child jobs (Figure 5.4) since they are inconclusive jobs. When an inconclusive parent job finish, the executor will create a new graph in which the inconclusive job (e.g. `C` or `G`) will be the root. If the executor detects that the children of this independent job has other parents that haven't finished yet, the execution of this new graph will be postponed.

Figure 5.3: Representation of a *pipeline* that includes inconclusive jobs.

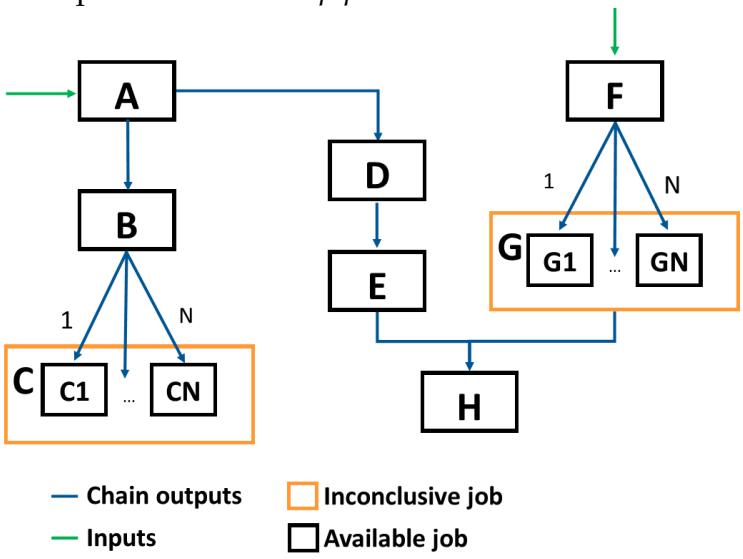
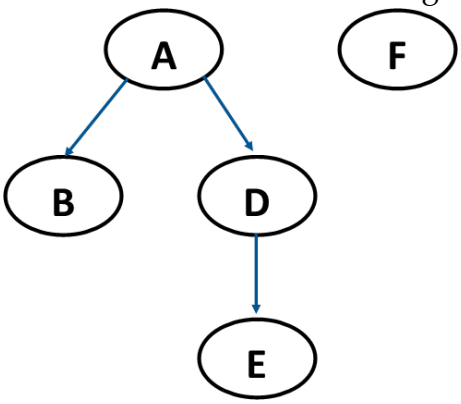


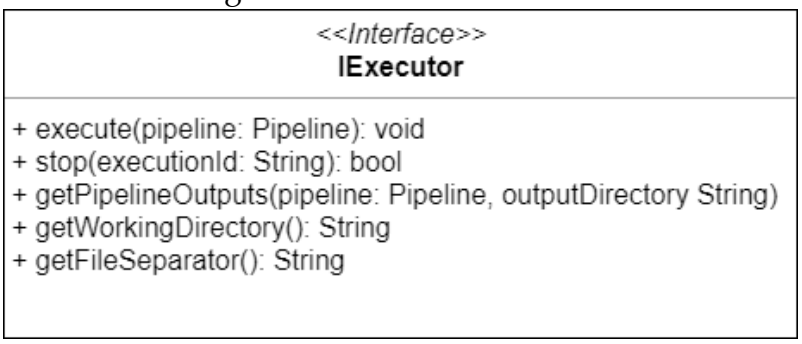
Figure 5.4: Representation of the first resulting execution graph.



5.4.2 Executors

Executors module are the main piece of engine, they are responsible to manage a *pipeline* execution. Figure 5.5 shows `IExecutor` interface.

Figure 5.5: IExecutor contract



The next sections will present the executors implementation that are supplied within this project.

5.4.2.1 Local executor

Local execution depends on the local operating system. Managing execution using Docker have some implications depending on the operating system:

- *Windows* - once *Docker* was not supported in all versions of this operating system, this solution provides a *NGSPipesV2* VM created from a `Vagrantfile`. In this case the executor manages the VM creation, start, power off and deletion;
- *Linux and Mac* - on Linux distributions, *Docker* is supported so everything executes directly on the system. This brings a space related issue, because each *pipeline* requires `n` *Docker* images. In these cases, user is responsible to manage and decide when to remove a *Docker* image.

5.4.2.2 Cluster executor

In Section 3.1 the main infrastructures were compared. As said before (Section 3.4), this solution supports execution on a *Mesos* cluster.

Mesos has some limitations concerning aspects required by the engine, namely, the access in each step to files produced by other steps and the management of

dependencies between steps in the *pipeline*. These limitations were solved as follows:

1. **Access to steps within the same *pipeline* files-** the approach is to use Network File System (NFS) [31] to allow each node access to all *pipeline* files as a local file system;

2. **Support for job dependency and DAGs-** because this infrastructure is a meta-scheduler that supports the use of another schedulers on top (known as frameworks), the solution is to use a framework (*Chronos*) built for *Mesos*.

There are a variety of frameworks schedulers that were built on *Mesos* and they have different focuses to solve a diverse type of problems such as: DevOps tooling, Long Services Running, Big Data Processing, Batch Scheduling, Data Storage and Machine Learning. This project solution is based on batch scheduling. There are some frameworks for batch scheduling built on *Mesos*. In this project it was chosen *Chronos* [26] since it is the one with better documentation among the others based on batch scheduling. *Chronos* is a framework for scheduling batch jobs. To schedule jobs, they offer a user interface and a RESTApi. Jobs can be scheduled like a command line job or docker job and support job dependency.

In order to use this infrastructure, the first step was to install and configure the cluster. This process was done in a VM on the local machine and on a server. Several steps were made to mount and setup *Mesos* infrastructure. These steps are presented on a script in Appendix I and they can be summarized as:

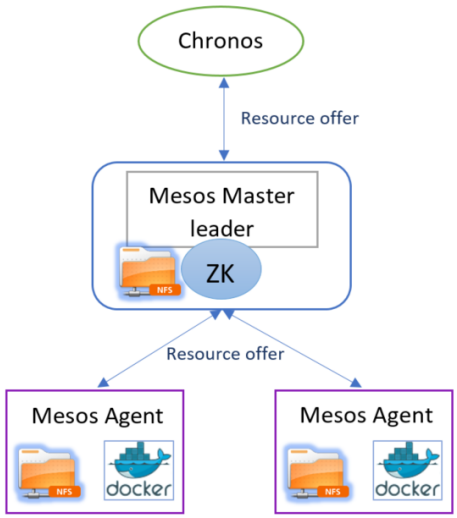
- install and configure *Zookeeper* [13]. *Zookeeper* is used to manage *Mesos* infrastructure distribution;
- install and configure *Mesos*;
- install and configure *Docker* on each *Mesos Agent*. Even when *Mesos* supports to execute tasks with *Docker* containers, *Docker* must be installed apart;
- install and configure *Chronos*;
- install and configure *NFS*.

Mesos framework works based on *Mesos Master* quorum (*Mesos Masters* and *Zookeeper*), *Agents* and frameworks. Each piece specified above has responsibilities on cluster and communicate with other pieces, so within the cluster their responsibilities are:

- *Zookeeper* is used to choose a master to be the leader, while the other masters will be on standby. The standby masters will be prepared to be chosen as leader if the actual leader get offline;
- *Master* manages all the *Agents* daemons running on the cluster machines. The leading master also decides what resources will be offered to each plugged framework (scheduler);
- Each scheduler can accept or reject the resources offered, depending if it has work to run at that time;
- *Chronos* scheduler uses *Zookeeper* to ensure fault tolerance;
- *Master* has an *NFS* server and each *Agent* has a *NFS* client to guaranty *pipelines* files access independent from which node is executing each job;
- *Docker* is used by an *Agent* when a *Docker* job is scheduled on *Chronos*;

The configured infrastructure has one master and two agent nodes. The *Chronos* scheduler framework uses resources from these nodes. Figure 5.6 depicts the cluster architecture.

Figure 5.6: Cluster executor architecture.



5.5 Results

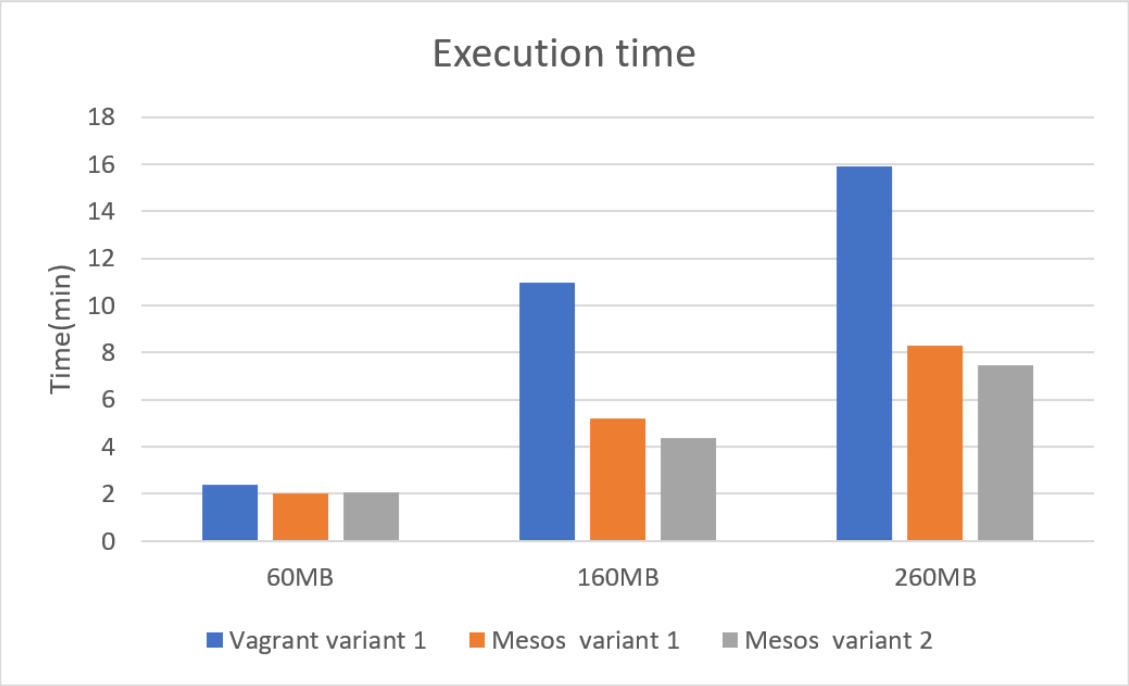
The executors supplied by this solution were tested. The tests were made for two of the *pipeline* variants (1 and 2, see Section 2.2), where variant 1 is a sequential

pipeline and variant 2 is a parallel version of the same *pipeline*. For each variant on `trimmomatic` command input (`trimmomatic_input`) three different file sizes were used (60 MB, 160 MB and 260 MB). Tests were made using Java Microbenchmark Harness [8] (JMH) benchmark library. Each iteration corresponds to ten executions of the target variant. The resources used by executors were:

- **Mesos** tested in a cluster with 4 CPUs and 6.6GB of RAM;
- **Vagrant** (local executor for Windows) tested in VMs created by the executor with a maximum of 4 CPUs and 8GB of RAM.

Figure 5.7 shows the results for inpust files with 60 MB, 160 MB and 260 MB. The presented results on each bar of the graphs correspond to an average made from ten executions. Graph shows the average of spent time in minutes per executor. In general, the best times are when executing the *pipelines* with *Mesos* executor, this fact can be related to the time *Vagrant* takes to start and shutdown the VMs.

Figure 5.7: Results of tests in minutes.



6

Conclusions and Future Work

The aim of this thesis was to improve *NGSPipesV1* project, extending it to support tool metadata annotation and *pipelines* parallel execution. We started by studying the state of the art among different SWSs to understand which characteristic they support and the underlying execution infrastructures. Then based on a characteristic comparison, depicted on Chapter 3, some issues and aspects to enhance were detected. The principal issues found were: not supporting parallel execution and a limited tool metadata annotation from dependencies and command execution generation point of view.

NGSPipesV2 metadata enables the *NGSPipesV2 Engine* to:

- validate the correct use of each command before being executed and thus save time and not take up resources unnecessarily;
- guarantee an automatic and accurate execution.

The designed execution engine is able to infer dependencies between steps in the *pipeline* and, based on the generated dependency graph, executes jobs taking advantage of data and task parallelism. Engine also supports nested *pipelines*.
The obtained results with the experience of having supported *Mesos* infrastructure were positive and presents improvements on the execution time in comparison to local execution with *Vagrant*.

This work is a significant evolution in several aspects in relation to the *NGSPipesV1* project. Still there are some aspects that can be improved, such as:

- improve tool metadata creation, by automating this process, in order to agile the work of the developers;
- supply a server with fault tolerance to host the execution engine;
- add execution engine functionalities, like `pause` and `resume`.

As result of this thesis a paper will be writted and presented to publish.

Bibliography

[1] Son of grid engine, January 2019. URL https://arc.liv.ac.uk/trac/SGE. (p. 15)

[2] Joao Forja Alexandre Almeida. Ngs4cloud: Cloud-based ngs data processing. December 2017. (p. 13)

[3] Joost Visser Arie van Deursen, Paul Klint. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35:26–36, 2000. (p. 49)

[4] J Arnemann. *Next-Generation-Sequencing (NGS)*, pages 1746–1747. 04 2019. ISBN 978-981-13-0988-5. doi: 10.1007/978-3-662-48986-4_3542. (p. 1)

[5] Nextflow authors. Nextflow's documentation!, December 2017. URL https://www.nextflow.io/docs/latest/index.html. (p. 14)

[6] Anthony Bolger. Usadellab: Trimmomatic: A flexible read trimming tool for illumina ngs data, December 2017. URL http://www.usadellab.org/cms/?page=trimmomatic. (pp. 7 and 52)

[7] John Sanabria Carlos Arango, Rémy Dernat. Performance evaluation of container-based virtualization for high performance computing environments. *CoRR*, abs/1709.10140, 2017. (pp. 22 and 23)

[8] Oracle Corporation. Openjdk: jmh, January 2019. URL https://openjdk.java.net/projects/code-tools/jmh/. (p. 74)

[9] Bruno Dantas and Calmenelias Fleitas. Infraestrutura de suporte à execução de fluxos de trabalho para a bioinformática, 2015. URL https://drive.google.com/file/d/1iae7ANoSSbTAwAcLpczlT6h0q78Ffq-5/view. (p. 3)

[10] Bruno Miguel das Neves Dantas. Bioinformatic pipeline specification language and sharing system. December 2018. (pp. 3 and 50)

[11] devteam. velvet: 08256557922f, January 2018. URL https://toolshed.g2.bx.psu.edu/repos/devteam/velvet/file/tip. (p. v)

[12] E. Birney D.R. Zerbino. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, (18):821–829, December 2017. (p. 7)

[13] The Apache Software Foundation. Apache zookeeper- home, June 2018. URL https://zookeeper.apache.org/. (p. 72)

[14] The Apache Software Foundation. Apache mesos, January 2019. URL http://mesos.apache.org/. (p. 16)

[15] The Apache Software Foundation. Apache hadoop 3.2.0 - apache hadoop yarn, January 2019. URL https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html. (p. 15)

[16] Martin Fowler. *Domain-Specific Languages*. 01 2010. ISBN 0321712943. (p. 3)

[17] Leo Goodstadt. Ruffus — ruffus 2.6.3 documentation, December 2018. URL http://www.ruffus.org.uk/. (p. 14)

[18] Michael W. Bauer Gregory M. Kurtzer, Vanessa Sochat. Singularity: Scientific containers for mobility of compute. *PLoS One*, 12(5):e0177459, 2017. (p. 22)

[19] HashiCorp. Vagrant by hashicorp, December 2018. URL https://www.vagrantup.com/. (p. 59)

[20] IBM. What is mapreduce? | ibm analytics, November 2018. URL https://www.ibm.com/analytics/hadoop/mapreduce. (p. 15)

[21] IBM. Platform lsf overview, January 2019. URL https://www.ibm.com/support/knowledgecenter/en/SSETD4_9.1.3/lsf_foundations/chap_lsf_overview_foundations.html. (p. 15)

[22] Docker Inc. Docker - build, ship, and run any app, anywhere, June 2018. URL https://www.docker.com/. (p. 22)

[23] Mesosphere Inc. Marathon: A container orchestration platform for mesos and dc/os, January 2019. URL http://mesosphere.github.io/marathon/. (p. 18)

[24] Gregory M. Kurtzer, Michael Bauer, Yannick Cote, and Dave Godlove. Singularity | singularity, December 2018. URL https://singularity.lbl.gov/. (p. 22)

[25] Common Workflow Language. workflows/tools at master · common-workflow-language/workflows, January 2018. URL https://github.com/common-workflow-language/workflows/tree/master/tools. (p. xxxi)

[26] Florian Leibert, Andy Kramolisch, Harry Shoff, and Elizabeth Lingg. Chronos: Fault tolerant job scheduler for mesos, May 2018. URL https://mesos.github.io/chronos/. (pp. 18 and 72)

[27] Jeremy Leipzig. A review of bioinformatic pipeline frameworks. *Briefings in Bioinformatics*, 18:bbw020, 03 2016. doi: 10.1093/bib/bbw020. (p. 5)

[28] NGSPipes V2 libraries. Ngspipes v2 libraries, February 2019. URL https://github.com/ngspipes2. (p. 65)

[29] NCBI. Blast: Basic local alignment search tool, December 2017. URL https://blast.ncbi.nlm.nih.gov/Blast.cgi. (p. 7)

[30] NERSC. Overview - nersc documentation, December 2018. URL https://docs.nersc.gov/development/shifter/overview/. (p. 22)

[31] Haynes & Noveck. Rfc 7530 - network file system (nfs) version 4 protocol, June 2018. URL https://tools.ietf.org/html/rfc7530. (p. 72)

[32] pjbriggs. trimmomatic: dfa082f84068, January 2018. URL https://toolshed.g2.bx.psu.edu/repos/pjbriggs/trimmomatic/file/tip. (p. xv)

[33] CWL Project. Common workflow language, December 2017. URL http://www.commonwl.org/. (p. 14)

[34] CWL Project. Common workflow language reference implementation, December 2017. URL https://github.com/common-workflow-language/cwltool. (p. 14)

[35] Galaxy Project. Galaxy community hub, December 2017. URL https://galaxyproject.org/. (p. 14)

[36] Galaxy Project. Toolshed - galaxy wiki, January 2018. URL https://wiki.galaxyproject.org/ToolShed. (p. 31)

[37] Swift project. The swift parallel scripting language, December 2017. URL http://swift-lang.org/main/. (p. 14)

[38] The Linux Information Project. All about pipes, by the linux information project (linfo), June 2019. URL http://www.linfo.org/pipe.html. (p. 2)

[39] Python.org. Wellcome to python.org, April 2019. URL https://www.python.org/. (p. 14)

[40] Helena Rasche. erasche/argparse2tool: transparently build cwl and galaxy xml tool definitions for any script that uses argparse, March 2018. URL https://github.com/erasche/argparse2tool#cwl-specific-functionality. (p. 34)

[41] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, and Jeremy Kepner. Scalable system scheduling for hpc and big data. *Journal of Parallel and Distributed Computing*, 111:76–92, January 2018. (pp. 14, 17, 18, 19, and 20)

[42] ShedMD. Slurm workload manager documentation, January 2019. URL https://slurm.schedmd.com/documentation.html. (p. 15)

[43] NGSPipes Team. Welcome to NGSPipes's documentation!, December 2017. URL http://ngspipes.readthedocs.io/en/latest/. (pp. 3 and 13)

[44] Dan Tsafrir Yoav Etsion. A short survey of commercial cluster batch schedulers. 2005. (p. 14)

NGSPipes velvet descriptor

Listing A.1 presents an example of *Velvet* tool descriptor.

Listing A.1: NGSPipes *Velvet* tool descriptor.

```
{
  "name" : "Velvet",
  "author" : "Daniel Zerbino[1] , Ewan Birney",
  "version" : "0.7.01",
  "description" : "Velvet is an algorithm .....",
  "documentation":["https://www.ebi.ac.uk/~zerbino/velvet"],
  "setup" : [ "make" ],
  "requiredMemory": 12288,
  "commands" :[
    {
      "name" : "velveth",
      "command" : "velveth",
      "description" : "construct the dataset .....",
      "priority" : 2,
      "argumentsComposer" : "values_separated_by_space",
      "arguments" : [
        {
            "name" : "output_directory",
            "argumentType" : "directory",
            "isRequired" : "true",
            "description" : "Directory output files"
          }, {
            "name" : "strand_specific",
```

```
            "argumentType" : "string",
            "isRequired" : "false",
            "description" : ""
          }, {
            "name" : "hash_length",
            "argumentType" : "int",
            "isRequired" : "true",
            "description" : ""
          }, {
            "name" : "filename",
            "argumentType" : "file",
            "isRequired" : "false",
            "description" : "Filename for analyse"
        }],
        "outputs" : [
          {
            "name" : "sequence",
            "description" : "",
            "outputType" : "directory_dependent",
            "argument_name" : "output_directory",
            "value" : "Sequences"
        }],
        "inputs": [
          {
          "name": "",
          "description": "",
          "inputType": "file_dependent",
          "argument_name": "filename",
          "value": ""
          }
        ]
    }, {
      "name" : "velvetg",
      "command" : "velvetg",
      "description" : "Build the Brujin graph .....",
      "priority" : 1,
      "argumentsComposer" : "name_values_separated_by_space",
      "arguments" : [.....]
    }]
}
```

B

NGS4Cloud velvet tool descriptor

Listing B.1 presents an example of *Velvet* tool descriptor.

Listing B.1: NGS4Cloud *Velvet* tool descriptor.

```
{
  "name" : "Velvet",
  "author" : "Daniel Zerbino[1] , Ewan Birney",
  "version" : "0.7.01",
  "description" : "Velvet is an algorithm .....",
  "documentation":["https://www.ebi.ac.uk/~zerbino/velvet"],
  "setup" : [ "make" ],
  "toolType": "unit",
  "requiredMemory": 12288,
  "recommendedDiskSpace": 1,
  "recommendedCpus": 1,
  "commands" :[
    {
    "name" : "velveth",
    "command" : "velveth",
    "description" : "construct the dataset .....",
    "priority" : 2,
    "argumentsComposer" : "values_separated_by_space",
    "arguments" : [
      {
        "name" : "output_directory",
        "argumentType" : "directory",
        "isRequired" : "true",
```

```
        "description" : "Directory output files"
      }, {
        "name" : "strand_specific",
        "argumentType" : "string",
        "isRequired" : "false",
        "description" : ""
      }, {
        "name" : "hash_length",
        "argumentType" : "int",
        "isRequired" : "true",
        "description" : ""
      }, {
        "name" : "filename",
        "argumentType" : "file",
        "isRequired" : "false",
        "description" : "Filename for analyse"
      }],
    "outputs" : [
      {
        "name" : "sequence",
        "description" : "",
        "outputType" : "directory_dependent",
        "argument_name" : "output_directory",
        "value" : "Sequences"
      }],
    "inputs": [
      {
        "name": "",
        "description": "",
        "inputType": "file_dependent",
        "argument_name": "filename",
        "value": ""
      }
    ]
  }, {
    "name" : "velvetg",
    "command" : "velvetg",
    "description" : "Build the Brujin graph .....",
    "priority" : 1,
    "argumentsComposer" : "name_values_separated_by_space",
    "arguments" : [.....]
  }]
}
```

C

Galaxy velvet command descriptor

Listing C.1 presents an example of *velveth* command descriptor from *Velvet* tool.

Listing C.1: Galaxy *velveth* command descriptor. Taken from [11]

```
<tool id="velveth" name="velveth" version="@WRAPPER_VERSION@.0">
  <description>Prepare a dataset .....</description>
  <version_command>velveth 2&gt;&amp;1 | grep "Version" | sed -e 's/
   Version//'</version_command>
  <macros>
    <import>macros.xml</import>
  </macros>
  <expand macro="requirements"/>
  <expand macro="stdio"/>
  <command interpreter="python">
    velveth_wrapper.py
    '$out_file1' '$out_file1.extra_files_path'
    $hash_length
    $strand_specific
    #for $i in $inputs
    ${i.file_format}
    ${i.read_type}
    ${i.input}
    #end for
  </command>
  <inputs>
    <param label="Hash Length" name="hash_length" type="select" help="k
   -mer length in base pairs .....">
```

```
22        <option value="19">19</option>
23        <option value="21" selected="yes">21</option>
24        <option value="27">27</option>
25      </param>
26      <param name="strand_specific" type="boolean" checked="false"
      truevalue="-strand_specific" falsevalue="" label="Use strand ..."
      help="If you are using ..."/>
27      <repeat name="inputs" title="Input Files">
28        <param label="file format" name="file_format" type="select">
29        <option value="-fasta" selected="yes">fasta</option>
30        <option value="-fastq">fastq</option>
31        </param>
32        <param name="input" type="data" format="fasta,fastq,eland,gerald"
       label="Dataset"/>
33      </repeat>
34    </inputs>
35    <outputs>
36      <data format="velvet" name="out_file1" />
37    </outputs>
38    <requirements>
39      <requirement type="package">velvet</requirement>
40    </requirements>
41    <expand macro="citation"/>
42  </tool>
```

D

Cwl-Runner velvet command descriptor

Listing D.1 presents an example of *velveth* command descriptor from *Velvet* tool.

Listing D.1: Cwl-runner *velveth* command descriptor.

```
#!/usr/bin/env cwl-runner

cwlVersion: cwl:v1.0
class: CommandLineTool

baseCommand: velveth

inputs:
  - id: output_directory
  type: string
  inputBinding:
    position: 1
  - id: hash_length
  type: int
  inputBinding:
    position: 2
  - id: file_format
  type: string
  inputBinding:
    position: 3
  - id: file
```

```
22    type: File
23    inputBinding:
24      position: 4
25
26  outputs:
27    - id: output
28    type: Directory
29    outputBinding:
30      glob: $(inputs.output_directory)
31    - id: log
32    type: File
33    outputBinding:
34      glob: $(inputs.output_directory + "/Log")
35    - id: roadmaps
36    type: File
37    outputBinding:
38      glob: $(inputs.output_directory + "/Roadmaps")
39    - id: sequences
40    type: File
41    outputBinding:
42      glob: $(inputs.output_directory + "/Sequences")
```

NGSPipes trimmomatic tool descriptor

Listing E.1 presents an example of a possible *Trimmomatic* tool descriptor. To see complete example go to https://github.com/ngspipes/tools/blob/master/Trimmomatic/Descriptor.json.

Listing E.1: NGSPipes *Trimmomatic* tool descriptor example.

```
{
  "name": "Trimmomatic",
  "author": "UNKNOWN",
  "version": "0.0.33",
  "description": "It works with FASTQ ...",
  "documentation": [
    "http://www.usadellab.org/cms/?page=trimmomatic",
    ...
  ],
  "setup": [
    "apt-get install -y default-jre"
  ],
  "toolType": "unit",
  "requiredMemory": 1024,
  "recommendedDiskSpace": 1,
  "recommendedCpus": 1,
  "commands": [
    {
```

```
19      "name": "trimmomatic",
20      "command": "/trimmomatic-run.sh",
21      "description": "It works with FASTQ...",
22      "priority": 1,
23      "argumentsComposer": "trimmomatic",
24      "arguments": [
25        {
26          "name": "mode",
27          "argumentType": "string",
28          "isRequired": true,
29          "description": "For single-ended data,..."
30        }, {
31          "name": "threads",
32          "argumentType": "int",
33          "isRequired": false,
34          "description": "indicates the number of threads ..."
35        }, {
36          "name": "quality",
37          "argumentType": "string",
38          "isRequired": true,
39          "description": "phred + 33 or phred + 64 quality scores"
40        }, {
41          "name": "trimlog",
42          "argumentType": "string",
43          "isRequired": false,
44          "description": "specifies the path to the log file..."
45        }, {
46          "name": "inputFile",
47          "argumentType": "file",
48          "isRequired": false,
49          "description": "Specifies the path to the fastq input file."
50        }, {
51          "name": "paired input 1",
52          "argumentType": "file",
53          "isRequired": false,
54          "description": "Specifies the path to the input file..."
55        }, {
56          "name": "paired input 2",
57          "argumentType": "file",
58          "isRequired": false,
59          "description": "Specifies the path to the input file..."
60        }, {
61          "name": "outputFile",
62          "argumentType": "file",
63          "isRequired": false,
```

```
64            "description": "Specifies the name of output file."
65          }, {
66            "name": "paired output 1",
67            "argumentType": "string",
68            "isRequired": false,
69            "description": "Specifies the name of paired output file 1."
70          }, {
71            "name": "unpaired output 1",
72            "argumentType": "string",
73            "isRequired": false,
74            "description": "Specifies the name of unpaired output..."
75          }, {
76            "name": "paired output 2",
77            "argumentType": "string",
78            "isRequired": false,
79            "description": "Specifies the name of paired output..."
80          }, {
81            "name": "unpaired output 2",
82            "argumentType": "string",
83            "isRequired": false,
84            "description": "Specifies the name of unpaired output..."
85          }, {
86            "name": "fastaWithAdaptersEtc",
87            "argumentType": "file",
88            "isRequired": false,
89            "description": "Specifies the path to a fasta file..."
90          }, {
91            "name": "seed mismatches",
92            "argumentType": "int",
93            "isRequired": false,
94            "description": "Specifies the maximum mismatch..."
95          }, {
96            "name": "palindrome clip threshold",
97            "argumentType": "int",
98            "isRequired": false,
99            "description": "Specifies how accurate the match ..."
100         }, {
101           "name": "simple clip threshold",
102           "argumentType": "int",
103           "isRequired": false,
104           "description": "Specifies how accurate the match..."
105         }, {
106           "name": "windowSize",
107           "argumentType": "int",
108           "isRequired": false,
```

```
109           "description": "Specifies the number of bases to..."
110         }, {
111           "name": "requiredQuality",
112           "argumentType": "int",
113           "isRequired": false,
114           "description": "Specifies the average quality..."
115         }, {
116           "name": "leading quality",
117           "argumentType": "int",
118           "isRequired": false,
119           "description": "Specifies the minimum quality..."
120         }, {
121           "name": "trailing quality",
122           "argumentType": "int",
123           "isRequired": false,
124           "description": "Specifies the minimum quality..."
125         }, {
126           "name": "crop length",
127           "argumentType": "int",
128           "isRequired": false,
129           "description": "The number of bases to keep..."
130         }, {
131           "name": "headcrop length",
132           "argumentType": "int",
133           "isRequired": false,
134           "description": "The number of bases to remove..."
135         }, {
136           "name": "minlen length",
137           "argumentType": "int",
138           "isRequired": false,
139           "description": "Specifies the minimum length..."
140         }
141       ],
142       "outputs": [
143         {
144           "name": "outputFile",
145           "description": "",
146           "outputType": "file_dependent",
147           "value": "",
148           "argument_name": "outputFile"
149         }, {
150           "name": "paired output 1",
151           "description": "",
152           "outputType": "file_dependent",
153           "value": "",
```

```
            "argument_name": "paired output 1"
          }, {
            "name": "unpaired output 1",
            "description": "",
            "outputType": "file_dependent",
            "value": "",
            "argument_name": "unpaired output 1"
          }, {
            "name": "paired output 2",
            "description": "",
            "outputType": "file_dependent",
            "value": "",
            "argument_name": "paired output 2"
          }, {
            "name": "unpaired output 2",
            "description": "",
            "outputType": "file_dependent",
            "value": "",
            "argument_name": "unpaired output 2"
          }
        ]
      }
  ]
}
```

F

Galaxy trimmomatic command descriptor

Listing F.1 presents an example of a possible *Trimmomatic* command descriptor for Trimmomatic.

Listing F.1: Galaxy *Trimmomatic* command descriptor example. Taken from [32]

```
<tool id="trimmomatic" name="Trimmomatic" version="0.36.5">
<description>flexible read trimming tool for Illumina NGS data</
    description>
<macros>
<import>trimmomatic_macros.xml</import>
</macros>
<requirements>
<requirement type="package" version="0.36">trimmomatic</requirement>
</requirements>
<command detect_errors="aggressive"><![CDATA[
@CONDA_TRIMMOMATIC_JAR_PATH@ &&
@CONDA_TRIMMOMATIC_ADAPTERS_PATH@ &&
#if $readtype.single_or_paired == "pair_of_files"
#set r1_ext = $readtype.fastq_r1_in.extension
#set r2_ext = $readtype.fastq_r2_in.extension
ln -s '$readtype.fastq_r1_in' fastq_r1.'$r1_ext' &&
ln -s '$readtype.fastq_r2_in' fastq_r2.'$r2_ext' &&
#elif $readtype.single_or_paired == "collection"
#set r1_ext = $readtype.fastq_pair.forward.extension
```

```
#set r2_ext = $readtype.fastq_pair.reverse.extension
ln -s '$readtype.fastq_pair.forward' fastq_r1.'$r1_ext' &&
ln -s '$readtype.fastq_pair.reverse' fastq_r2.'$r2_ext' &&
#else
ln -s '$fastq_in' fastq_in.'$fastq_in.extension' &&
#end if
java \${_JAVA_OPTIONS:--Xmx8G} -jar \$TRIMMOMATIC_JAR_PATH/trimmomatic.jar
#if $readtype.single_or_paired in ["pair_of_files","collection"]
PE -threads \${GALAXY_SLOTS:-6} -phred33
fastq_r1.'$r1_ext' fastq_r2.'$r2_ext'
fastq_out_r1_paired.'$r1_ext' fastq_out_r1_unpaired.'$r1_ext'
fastq_out_r2_paired.'$r2_ext' fastq_out_r2_unpaired.'$r2_ext'
#else
SE -threads \${GALAXY_SLOTS:-6} -phred33 fastq_in.'$fastq_in.extension'
 fastq_out.'$fastq_in.extension'
#end if
## ILLUMINACLIP option
#if $illuminaclip.do_illuminaclip
#if $illuminaclip.adapter_type.standard_or_custom == "custom"
#if $readtype.single_or_paired in ["pair_of_files","collection"]
ILLUMINACLIP:$adapter_file_from_text:$illuminaclip.seed_mismatches:$illuminaclip.palindrome_clip_threshold:$illuminaclip.simple_clip_threshold:$illuminaclip.min_adapter_len:$illuminaclip.keep_both_reads
#else
ILLUMINACLIP:$adapter_file_from_text:$illuminaclip.seed_mismatches:$illuminaclip.palindrome_clip_threshold:$illuminaclip.simple_clip_threshold
#end if
#else
#if $readtype.single_or_paired in ["pair_of_files","collection"]
ILLUMINACLIP:\$TRIMMOMATIC_ADAPTERS_PATH/$illuminaclip.adapter_type.adapter_fasta:$illuminaclip.seed_mismatches:$illuminaclip.palindrome_clip_threshold:$illuminaclip.simple_clip_threshold:$illuminaclip.min_adapter_len:$illuminaclip.keep_both_reads
#else
ILLUMINACLIP:\$TRIMMOMATIC_ADAPTERS_PATH/$illuminaclip.adapter_type.adapter_fasta:$illuminaclip.seed_mismatches:$illuminaclip.palindrome_clip_threshold:$illuminaclip.simple_clip_threshold
#end if
#end if
#end if
## Other operations
#for $op in $operations
```

```
52 ## SLIDINGWINDOW
53 #if str( $op.operation.name ) == "SLIDINGWINDOW"
54 SLIDINGWINDOW:$op.operation.window_size:$op.operation.required_quality
55 #end if
56 ## MINLEN:36
57 #if str( $op.operation.name ) == "MINLEN"
58 MINLEN:$op.operation.minlen
59 #end if
60 #if str( $op.operation.name ) == "LEADING"
61 LEADING:$op.operation.leading
62 #end if
63 #if str( $op.operation.name ) == "TRAILING"
64 TRAILING:$op.operation.trailing
65 #end if
66 #if str( $op.operation.name ) == "CROP"
67 CROP:$op.operation.crop
68 #end if
69 #if str( $op.operation.name ) == "HEADCROP"
70 HEADCROP:$op.operation.headcrop
71 #end if
72 #if str( $op.operation.name ) == "AVGQUAL"
73 AVGQUAL:$op.operation.avgqual
74 #end if
75 #if str( $op.operation.name ) == "MAXINFO"
76 MAXINFO:$op.operation.target_length:$op.operation.strictness
77 #end if
78 #end for
79 2>&1 | tee trimmomatic.log &&
80 if [ -z "\$(tail -1 trimmomatic.log | grep "Completed successfully")"
       ]; then echo "Trimmomatic did not finish successfully" >&2 ; exit 1
        ; fi
81 &&
82 #if $readtype.single_or_paired  == "pair_of_files"
83 mv fastq_out_r1_paired.'$r1_ext' '${fastq_out_r1_paired}' &&
84 mv fastq_out_r1_unpaired.'$r1_ext' '${fastq_out_r1_unpaired}' &&
85 mv fastq_out_r2_paired.'$r2_ext' '${fastq_out_r2_paired}' &&
86 mv fastq_out_r2_unpaired.'$r2_ext' '${fastq_out_r2_unpaired}'
87 #elif $readtype.single_or_paired  == "collection"
88 mv fastq_out_r1_paired.'$r1_ext' '${fastq_out_paired.forward}' &&
89 mv fastq_out_r1_unpaired.'$r1_ext' '${fastq_out_unpaired.forward}' &&
90 mv fastq_out_r2_paired.'$r2_ext' '${fastq_out_paired.reverse}' &&
91 mv fastq_out_r2_unpaired.'$r2_ext' '${fastq_out_unpaired.reverse}'
92 #else
93 mv fastq_out.'$fastq_in.extension' '${fastq_out}'
94 #end if
```

```
]]></command>
<configfiles>
<configfile name="adapter_file_from_text">#set from_text_area = ''
#if str( $illuminaclip.do_illuminaclip ) == "yes" and str( $
    illuminaclip.adapter_type.standard_or_custom ) == "custom":
#set from_text_area = $illuminaclip.adapter_type.adapter_text
#end if
${from_text_area}</configfile>
</configfiles>

<inputs>
<conditional name="readtype">
<param name="single_or_paired" type="select" label="Single-end or
    paired-end reads?">
<option value="se" selected="true">Single-end</option>
<option value="pair_of_files">Paired-end (two separate input files)</
    option>
<option value="collection">Paired-end (as collection)</option>
</param>
<when value="se">
<param name="fastq_in" type="data" format="fastqsanger,fastqsanger.gz"
    label="Input FASTQ file" />
</when>
<when value="pair_of_files">
<param name="fastq_r1_in" type="data" format="fastqsanger,fastqsanger.
    gz"
label="Input FASTQ file (R1/first of pair)" />
<param name="fastq_r2_in" type="data" format="fastqsanger,fastqsanger.
    gz"
label="Input FASTQ file (R2/second of pair)" />
</when>
<when value="collection">
<param name="fastq_pair" format="fastqsanger,fastqsanger.gz" type="data
    _collection" collection_type="paired" label="Select FASTQ dataset
    collection with R1/R2 pair" />
</when>
</conditional>
<conditional name="illuminaclip">
<param name="do_illuminaclip" type="boolean" label="Perform initial
    ILLUMINACLIP step?" help="Cut adapter and other illumina-specific
    sequences from the read" truevalue="yes" falsevalue="no" checked="
    False" />
<when value="yes">
<conditional name="adapter_type">
```

```
128 <param name="standard_or_custom" type="select" label="Select standard
    adapter sequences or provide custom?">
129 <option value="standard" selected="true">Standard</option>
130 <option value="custom">Custom</option>
131 </param>
132 <when value="standard">
133 <param name="adapter_fasta" type="select" label="Adapter sequences to
    use">
134 <option value="TruSeq2-SE.fa">TruSeq2 (single-ended, for Illumina GAII)
    </option>
135 <option value="TruSeq3-SE.fa">TruSeq3 (single-ended, for MiSeq and
    HiSeq)</option>
136 <option value="TruSeq2-PE.fa">TruSeq2 (paired-ended, for Illumina GAII)
    </option>
137 <option value="TruSeq3-PE.fa">TruSeq3 (paired-ended, for MiSeq and
    HiSeq)</option>
138 <option value="TruSeq3-PE-2.fa">TruSeq3 (additional seqs) (paired-ended
    , for MiSeq and HiSeq)</option>
139 <option value="NexteraPE-PE.fa">Nextera (paired-ended)</option>
140 </param>
141 </when>
142 <when value="custom">
143 <param name="adapter_text" type="text" area="True" size="10x30" value="
    "
144 label="Custom adapter sequences in fasta format" help="Write sequences
    in the fasta format.">
145 <sanitizer>
146 <valid initial="string.printable"></valid>
147 <mapping initial="none"/>
148 </sanitizer>
149 </param>
150 </when>
151 </conditional>
152 <param name="seed_mismatches" type="integer" label="Maximum mismatch
    count which will still allow a full match to be performed" value="2
    " />
153 <param name="palindrome_clip_threshold" type="integer" label="How
    accurate the match between the two 'adapter ligated' reads must be
    for PE palindrome read alignment" value="30" />
154 <param name="simple_clip_threshold" type="integer" label="How accurate
    the match between any adapter etc. sequence must be against a read"
     value="10" />
155 <param name="min_adapter_len" type="integer" label="Minimum length of
    adapter that needs to be detected (PE specific/palindrome mode)"
    value="8" />
```

```
156 <param name="keep_both_reads" type="boolean" label="Always keep both
       reads (PE specific/palindrome mode)?" truevalue="true" falsevalue="
       false" checked="true"
157 help="See help below"/>
158 </when>
159 <when value="no" /> <!-- empty clause to satisfy planemo lint -->
160 </conditional>
161 <repeat name="operations" title="Trimmomatic Operation" min="1">
162 <conditional name="operation">
163 <param name="name" type="select" label="Select Trimmomatic operation to
        perform">
164 <option selected="true" value="SLIDINGWINDOW">Sliding window trimming (
       SLIDINGWINDOW)</option>
165 <option value="MINLEN">Drop reads below a specified length (MINLEN)</
       option>
166 <option value="LEADING">Cut bases off the start of a read, if below a
       threshold quality (LEADING)</option>
167 <option value="TRAILING">Cut bases off the end of a read, if below a
       threshold quality (TRAILING)</option>
168 <option value="CROP">Cut the read to a specified length (CROP)</option>
169 <option value="HEADCROP">Cut the specified number of bases from the
       start of the read (HEADCROP)</option>
170 <option value="AVGQUAL">Drop reads with average quality lower than a
       specified level (AVGQUAL)</option>
171 <option value="MAXINFO">Trim reads adaptively, balancing read length
       and error rate to maximise the value of each read (MAXINFO)</option
       >
172 </param>
173 <when value="SLIDINGWINDOW">
174 <param name="window_size" type="integer" label="Number of bases to
       average across" value="4" />
175 <param name="required_quality" type="integer" label="Average quality
       required" value="20" />
176 </when>
177 <when value="MINLEN">
178 <param name="minlen" type="integer" label="Minimum length of reads to
       be kept" value="20" />
179 </when>
180 <when value="LEADING">
181 <param name="leading" type="integer" label="Minimum quality required to
        keep a base" value="3" help="Bases at the start of the read with
       quality below the threshold will be removed" />
182 </when>
183 <when value="TRAILING">
```

```
184 <param name="trailing" type="integer" label="Minimum quality required
       to keep a base" value="3" help="Bases at the end of the read with
       quality below the threshold will be removed" />
185 </when>
186 <when value="CROP">
187 <param name="crop" type="integer" label="Number of bases to keep from
       the start of the read" value="" />
188 </when>
189 <when value="HEADCROP">
190 <param name="headcrop" type="integer" label="Number of bases to remove
       from the start of the read" value="" />
191 </when>
192 <when value="AVGQUAL">
193 <param name="avgqual" type="integer" label="Minimum average quality
       required to keep a read" value="" />
194 </when>
195 <when value="MAXINFO">
196 <param name="target_length" type="integer" label="Target read length"
       value="" help="The read length which is likely to allow the
       location of the read within the target sequence to be determined."
       />
197 <param name="strictness" type="float" label="Strictness" value="" help=
       "Set between zero and one - specifies the balance between
       preserving read length versus removal of incorrect bases; low
       values (&lt;0.2) favours longer reads, high values (&gt;0.8)
       favours read correctness." />
198 </when>
199 </conditional>
200 </repeat>
201 </inputs>
202 <outputs>
203 <data name="fastq_out_r1_paired" label="${tool.name} on ${readtype.
       fastq_r1_in.name} (R1 paired)" format_source="fastq_r1_in">
204 <filter>readtype['single_or_paired'] == "pair_of_files"</filter>
205 </data>
206 <data name="fastq_out_r2_paired" label="${tool.name} on ${readtype.
       fastq_r2_in.name} (R2 paired)" format_source="fastq_r2_in">
207 <filter>readtype['single_or_paired'] == "pair_of_files"</filter>
208 </data>
209 <data name="fastq_out_r1_unpaired" label="${tool.name} on ${readtype.
       fastq_r1_in.name} (R1 unpaired)" format_source="fastq_r1_in">
210 <filter>readtype['single_or_paired'] == "pair_of_files"</filter>
211 </data>
212 <data name="fastq_out_r2_unpaired" label="${tool.name} on ${readtype.
       fastq_r2_in.name} (R2 unpaired)" format_source="fastq_r2_in">
```

```
213 <filter>readtype['single_or_paired'] == "pair_of_files"</filter>
214 </data>
215 <data name="fastq_out" label="${tool.name} on ${readtype.fastq_in.name}
       " format_source="fastq_in">
216 <filter>readtype['single_or_paired'] == 'se'</filter>
217 </data>
218 <collection name="fastq_out_paired" type="paired" label="${tool.name}
       on ${on_string}: paired">
219 <filter>readtype['single_or_paired'] == "collection"</filter>
220 <data name="forward" label="${tool.name} on ${readtype.fastq_pair.
       forward.name} (R1 paired)" format_source="fastq_pair['forward']"/>
221 <data name="reverse" label="${tool.name} on ${readtype.fastq_pair.
       reverse.name} (R2 paired)" format_source="fastq_pair['reverse']"/>
222 </collection>
223 <collection name="fastq_out_unpaired" type="paired" label="${tool.name}
        on ${on_string}: unpaired">
224 <filter>readtype['single_or_paired'] == "collection"</filter>
225 <data name="forward" label="${tool.name} on ${readtype.fastq_pair.
       forward.name} (R1 unpaired)" format_source="fastq_pair['forward']"/
       >
226 <data name="reverse" label="${tool.name} on ${readtype.fastq_pair.
       reverse.name} (R2 unpaired)" format_source="fastq_pair['reverse']"/
       >
227 </collection>
228
229 </outputs>
230 <tests>
231 <test>
232 <!-- Single-end example -->
233 <param name="single_or_paired" value="se" />
234 <param name="fastq_in" value="Illumina_SG_R1.fastq" ftype="fastqsanger"
        />
235 <param name="operations_0|operation|name" value="SLIDINGWINDOW" />
236 <output name="fastq_out" file="trimmomatic_se_out1.fastq" />
237 </test>
238 <test>
239 <!-- Single-end example - gzipped -->
240 <param name="single_or_paired" value="se" />
241 <param name="fastq_in" value="Illumina_SG_R1.fastq.gz" ftype="
       fastqsanger.gz" />
242 <param name="operations_0|operation|name" value="SLIDINGWINDOW" />
243 <output name="fastq_out" file="trimmomatic_se_out1.fastq.gz" />
244 </test>
245 <test>
246 <!-- Paired-end example - gzipped -->
```

```
247 <param name="single_or_paired" value="pair_of_files" />
248 <param name="fastq_r1_in" value="Illumina_SG_R1.fastq.gz" ftype="
       fastqsanger.gz" />
249 <param name="fastq_r2_in" value="Illumina_SG_R2.fastq.gz" ftype="
       fastqsanger.gz" />
250 <param name="operations_0|operation|name" value="SLIDINGWINDOW" />
251 <output name="fastq_out_r1_paired" file="trimmomatic_pe_r1_paired_out1.
       fastq.gz" />
252 <output name="fastq_out_r1_unpaired" file="trimmomatic_pe_r1_unpaired_
       out1.fastq.gz" />
253 <output name="fastq_out_r2_paired" file="trimmomatic_pe_r2_paired_out1.
       fastq.gz" />
254 <output name="fastq_out_r2_unpaired" file="trimmomatic_pe_r2_unpaired_
       out1.fastq.gz" />
255 </test>
256 <test>
257 <!-- Paired-end example -->
258 <param name="single_or_paired" value="pair_of_files" />
259 <param name="fastq_r1_in" value="Illumina_SG_R1.fastq" ftype="
       fastqsanger" />
260 <param name="fastq_r2_in" value="Illumina_SG_R2.fastq" ftype="
       fastqsanger" />
261 <param name="operations_0|operation|name" value="SLIDINGWINDOW" />
262 <output name="fastq_out_r1_paired" file="trimmomatic_pe_r1_paired_out1.
       fastq" />
263 <output name="fastq_out_r1_unpaired" file="trimmomatic_pe_r1_unpaired_
       out1.fastq" />
264 <output name="fastq_out_r2_paired" file="trimmomatic_pe_r2_paired_out1.
       fastq" />
265 <output name="fastq_out_r2_unpaired" file="trimmomatic_pe_r2_unpaired_
       out1.fastq" />
266 </test>
267 <test>
268 <!-- Single-end example (cropping) -->
269 <param name="single_or_paired" value="se" />
270 <param name="fastq_in" value="Illumina_SG_R1.fastq" ftype="fastqsanger"
        />
271 <param name="operations_0|operation|name" value="CROP" />
272 <param name="operations_0|operation|crop" value="10" />
273 <output name="fastq_out" file="trimmomatic_se_out2.fastq" />
274 </test>
275 <test>
276 <!-- Paired-end with dataset collection -->
277 <param name="single_or_paired" value="collection" />
278 <param name="fastq_pair">
```

```
279 <collection type="paired">
280 <element name="forward" value="Illumina_SG_R1.fastq" ftype="fastqsanger
    " />
281 <element name="reverse" value="Illumina_SG_R2.fastq" ftype="fastqsanger
    "/>
282 </collection>
283 </param>
284 <param name="operations_0|operation|name" value="SLIDINGWINDOW" />
285 <output_collection name="fastq_out_paired" type="paired">
286 <element name="forward" file="trimmomatic_pe_r1_paired_out1.fastq" />
287 <element name="reverse" file="trimmomatic_pe_r2_paired_out1.fastq" />
288 </output_collection>
289 <output_collection name="fastq_out_unpaired" type="paired">
290 <element name="forward" file="trimmomatic_pe_r1_unpaired_out1.fastq" />
291 <element name="reverse" file="trimmomatic_pe_r2_unpaired_out1.fastq" />
292 </output_collection>
293 </test>
294 <test>
295 <!-- Paired-end with dataset collection - gzipped -->
296 <param name="single_or_paired" value="collection" />
297 <param name="fastq_pair">
298 <collection type="paired">
299 <element name="forward" value="Illumina_SG_R1.fastq.gz" ftype="
    fastqsanger.gz" />
300 <element name="reverse" value="Illumina_SG_R2.fastq.gz" ftype="
    fastqsanger.gz"/>
301 </collection>
302 </param>
303 <param name="operations_0|operation|name" value="SLIDINGWINDOW" />
304 <output_collection name="fastq_out_paired" type="paired">
305 <element name="forward" file="trimmomatic_pe_r1_paired_out1.fastq.gz" /
    >
306 <element name="reverse" file="trimmomatic_pe_r2_paired_out1.fastq.gz" /
    >
307 </output_collection>
308 <output_collection name="fastq_out_unpaired" type="paired">
309 <element name="forward" file="trimmomatic_pe_r1_unpaired_out1.fastq.gz"
     />
310 <element name="reverse" file="trimmomatic_pe_r2_unpaired_out1.fastq.gz"
     />
311 </output_collection>
312 </test>
313 <test>
314 <!-- Single-end using AVGQUAL -->
315 <param name="single_or_paired" value="se" />
```

```
316 <param name="fastq_in" value="Illumina_SG_R1.fastq" ftype="fastqsanger"
        />
317 <param name="operations_0|operation|name" value="AVGQUAL" />
318 <param name="operations_0|operation|avgqual" value="30" />
319 <output name="fastq_out" file="trimmomatic_avgqual.fastq" />
320 </test>
321 <test>
322 <!-- Single-end using MAXINFO -->
323 <param name="single_or_paired" value="se" />
324 <param name="fastq_in" value="Illumina_SG_R1.fastq" ftype="fastqsanger"
        />
325 <param name="operations_0|operation|name" value="MAXINFO" />
326 <param name="operations_0|operation|target_length" value="75" />
327 <param name="operations_0|operation|strictness" value="0.8" />
328 <output name="fastq_out" file="trimmomatic_maxinfo.fastq" />
329 </test>
330 <test>
331 <!-- Paired-end ILLUMINACLIP - this does not check valid clipping -->
332 <param name="single_or_paired" value="pair_of_files" />
333 <param name="fastq_r1_in" value="Illumina_SG_R1.fastq" ftype="
       fastqsanger" />
334 <param name="fastq_r2_in" value="Illumina_SG_R2.fastq" ftype="
       fastqsanger" />
335 <param name="do_illuminaclip" value="true"/>
336 <param name="adapter_fasta" value="TruSeq2-PE.fa"/>
337 <param name="operations_0|operation|name" value="SLIDINGWINDOW" />
338 <output name="fastq_out_r1_paired" file="trimmomatic_pe_r1_paired_out1_
       clip.fastq" />
339 <output name="fastq_out_r1_unpaired" file="trimmomatic_pe_r1_unpaired_
       out1.fastq" />
340 <output name="fastq_out_r2_paired" file="trimmomatic_pe_r2_paired_out1.
       fastq" />
341 <output name="fastq_out_r2_unpaired" file="trimmomatic_pe_r2_unpaired_
       out1_clip.fastq" />
342 </test>
343 <test>
344 <!-- Paired-end ILLUMINACLIP providing 'custom' adapters - this does
       not check valid clipping -->
345 <param name="single_or_paired" value="pair_of_files" />
346 <param name="fastq_r1_in" value="Illumina_SG_R1.fastq" ftype="
       fastqsanger" />
347 <param name="fastq_r2_in" value="Illumina_SG_R2.fastq" ftype="
       fastqsanger" />
348 <param name="do_illuminaclip" value="true"/>
349 <param name="standard_or_custom" value="custom"/>
```

```
<param name="adapter_text"
value=">PrefixPE/1&#10;
    AATGATACGGCGACCACCGAGATCTACACTCTTTCCCTACACGACGCTCTTCCGATCT&#10;>
    PrefixPE/2&#10;
    CAAGCAGAAGACGGCATACGAGATCGGTCTCGGCATTCCTGCTGAACCGCTCTTCCGATCT&#10;>
    PCR_Primer1&#10;
    AATGATACGGCGACCACCGAGATCTACACTCTTTCCCTACACGACGCTCTTCCGATCT&#10;>PCR
    _Primer1_rc&#10;
    AGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGTAGATCTCGGTGGTCGCCGTATCATT&#10;>PCR
    _Primer2&#10;
    CAAGCAGAAGACGGCATACGAGATCGGTCTCGGCATTCCTGCTGAACCGCTCTTCCGATCT&#10;>
    PCR_Primer2_rc&#10;
    AGATCGGAAGAGCGGTTCAGCAGGAATGCCGAGACCGATCTCGTATGCCGTCTTCTGCTTG&#10;>
    FlowCell1&#10;TTTTTTTTTTAATGATACGGCGACCACCGAGATCTACAC&#10;>
    FlowCell2&#10;TTTTTTTTTTCAAGCAGAAGACGGCATACGA&#10;"/>
<param name="adapter_fasta" value="TruSeq2-PE.fa"/>
<param name="operations_0|operation|name" value="SLIDINGWINDOW" />
<output name="fastq_out_r1_paired" file="trimmomatic_pe_r1_paired_out1_
    clip.fastq" />
<output name="fastq_out_r1_unpaired" file="trimmomatic_pe_r1_unpaired_
    out1.fastq" />
<output name="fastq_out_r2_paired" file="trimmomatic_pe_r2_paired_out1.
    fastq" />
<output name="fastq_out_r2_unpaired" file="trimmomatic_pe_r2_unpaired_
    out1_clip.fastq" />
</test>
</tests>
<help><![CDATA[
.. class:: infomark

**What it does**

Trimmomatic performs a variety of useful trimming tasks for illumina
    paired-end and
single ended data.

This tool allows the following trimming steps to be performed:

* **ILLUMINACLIP:** Cut adapter and other illumina-specific sequences
    from the read

* If **Always keep both reads (PE specific/palindrome mode)** is True,
    the reverse read will also be retained in palindrome mode.
After read-though has been detected by palindrome mode, and the adapter
     sequence removed,
```

```
374 the reverse read contains the same sequence information as the forward
       read, albeit in reverse complement.
375 For this reason, the default behaviour is to entirely drop the reverse
       read.
376 Retaining the reverse read may be useful e.g. if the downstream tools
       cannot handle a combination of paired and unpaired reads.
377 * **SLIDINGWINDOW:** Perform a sliding window trimming, cutting once
       the average
378 quality within the window falls below a threshold
379 * **MINLEN:** Drop the read if it is below a specified length
380 * **LEADING:** Cut bases off the start of a read, if below a threshold
       quality
381 * **TRAILING:** Cut bases off the end of a read, if below a threshold
       quality
382 * **CROP:** Cut the read to a specified length
383 * **HEADCROP:** Cut the specified number of bases from the start of the
        read
384 * **AVGQUAL:** Drop the read if the average quality is below a
       specified value
385 * **MAXINFO:** Trim reads adaptively, balancing read length and error
       rate to
386 maximise the value of each read
387
388 If ILLUMINACLIP is requested then it is always performed first;
       subsequent options
389 can be mixed and matched and will be performed in the order that they
       have been
390 specified.
391
392 .. class:: warningmark
393
394 Note that trimming operation order is important.
395
396 -------------
397
398 .. class:: infomark
399
400 **Inputs**
401
402 For single-end data this Trimmomatic tool accepts a single FASTQ file;
       for
403 paired-end data it will accept either two FASTQ files (R1 and R2), or a
404 dataset collection containing the R1/R2 FASTQ pair.
405
406 .. class:: infomark
```

```
407
408 **Outputs**
409
410 For paired-end data a particular strength of Trimmomatic is that it
       retains the
411 pairing of reads (from R1 and R2) in the filtered output files:
412
413 * Two FASTQ files (R1-paired and R2-paired) contain one read from each
       pair where
414 both have survived filtering.
415 * Additionally two FASTQ files (R1-unpaired and R2-unpaired) contain
       reads where
416 one of the pair failed the filtering steps.
417
418 .. class:: warningmark
419
420 If the input consists of a dataset collection with the R1/R2 FASTQ pair
        then
421 the outputs will also inclue two dataset collections: one for the '
       paired'
422 outputs and one for the 'unpaired' (as described above)
423
424 Retaining the same order and number of reads in the filtered output
       fastq files is
425 essential for many downstream analysis tools.
426
427 For single-end data the output is a single FASTQ file containing just
       the filtered
428 reads.
429
430 -------------
431
432 .. class:: infomark
433
434 **Credits**
435
436 This Galaxy tool has been developed within the Bioinformatics Core
       Facility at the
437 University of Manchester, with contributions from Peter van Heusden,
       Marius
438 van den Beek, Jelle Scholtalbers and Charles Girardot.
439
440 It runs the Trimmomatic program which has been developed
441 within Bjorn Usadel's group at RWTH Aachen university.
442
```

```
443 Trimmomatic website (including documentation):
444
445 * http://www.usadellab.org/cms/index.php?page=trimmomatic
446
447 The reference for Trimmomatic is:
448
449 * Bolger, A.M., Lohse, M., &amp;amp; Usadel, B. (2014). Trimmomatic: A
       flexible trimmer
450 for Illumina Sequence Data. Bioinformatics, btu170.
451
452 Please kindly acknowledge both this Galaxy tool and the Trimmomatic
       program if you
453 use it.
454 ]]></help>
455 <citations>
456 <!--
457 See https://wiki.galaxyproject.org/Admin/Tools/ToolConfigSyntax#A.3
       Ccitations.3E_tag_set
458 Can be either DOI or Bibtex
459 Use http://www.bioinformatics.org/texmed/ to convert PubMed to Bibtex
460 -->
461 <citation type="doi">10.1093/bioinformatics/btu170</citation>
462 </citations>
463 </tool>
```

G

Cwl-Runner trimmomatic command descriptor

Listing G.1 presents an example of a possible *Trimmomatic* command descriptor for Cwl-Runner.

Listing G.1: Cwl-runner *Trimmomatic* command descriptor example. Taken from [25]

```
#!/usr/bin/env cwl-runner

cwlVersion: v1.0
class: CommandLineTool

hints:
  SoftwareRequirement:
    packages:
      trimmomatic:
        specs: [ "https://identifiers.org/rrid/RRID:SCR_011848" ]
        version: [ "0.32", "0.35", "0.36" ]

requirements:
  ResourceRequirement:
    ramMin: 10240
    coresMin: 8
  SchemaDefRequirement:
    types:
```

```
    - $import: trimmomatic-end_mode.yaml
    - $import: trimmomatic-sliding_window.yaml
    - $import: trimmomatic-phred.yaml
    - $import: trimmomatic-illumina_clipping.yaml
    - $import: trimmomatic-max_info.yaml
  InlineJavascriptRequirement: {}
  ShellCommandRequirement: {}

# hints:
#   - $import: trimmomatic-docker.yml

inputs:
  phred:
    type: trimmomatic-phred.yaml#phred?
    inputBinding:
      prefix: -phred
      separate: false
      position: 4
    doc: |
    "33" or "64" specifies the base quality encoding. Default: 64

  tophred64:
    type: boolean?
    inputBinding:
      position: 12
      prefix: TOPHRED64
      separate: false
    doc: This (re)encodes the quality part of the FASTQ file to base
     64.

  headcrop:
    type: int?
    inputBinding:
      position: 13
      prefix: 'HEADCROP:'
      separate: false
    doc: |
    Removes the specified number of bases, regardless of quality, from
     the
    beginning of the read.
    The numbser specified is the number of bases to keep, from the
     start of
    the read.

  tophred33:
```

```
    type: boolean?
    inputBinding:
      position: 12
      prefix: TOPHRED33
      separate: false
    doc: This (re)encodes the quality part of the FASTQ file to base
   33.

  minlen:
    type: int?
    inputBinding:
      position: 100
      prefix: 'MINLEN:'
      separate: false
    doc: |
    This module removes reads that fall below the specified minimal
   length.
    If required, it should normally be after all other processing steps
   .
    Reads removed by this step will be counted and included in the "
   dropped
    reads" count presented in the trimmomatic summary.

  java_opts:
    type: string?
    inputBinding:
      position: 1
      shellQuote: false
    doc: |
    JVM arguments should be a quoted, space separated list
    (e.g. "-Xms128m -Xmx512m")

  leading:
    type: int?
    inputBinding:
      position: 14
      prefix: 'LEADING:'
      separate: false
    doc: |
    Remove low quality bases from the beginning. As long as a base has
   a
    value below this threshold the base is removed and the next base
   will be
    investigated.

```

```
100   slidingwindow:
101     type: trimmomatic-sliding_window.yaml#slidingWindow?
102     inputBinding:
103       position: 15
104       valueFrom: |
105         ${ if ( self ) {
106             return "SLIDINGWINDOW:" + self.windowSize + ":"
107             + self.requiredQuality;
108           } else {
109             return self;
110           }
111         }
112     doc: |
113     Perform a sliding window trimming, cutting once the average quality
114     within the window falls below a threshold. By considering multiple
115     bases, a single poor quality base will not cause the removal of
      high
116     quality data later in the read.
117     <windowSize> specifies the number of bases to average across
118     <requiredQuality> specifies the average quality required
119
120   illuminaClip:
121     type: trimmomatic-illumina_clipping.yaml#illuminaClipping?
122     inputBinding:
123       valueFrom: |
124         ${ if ( self ) {
125             return "ILLUMINACLIP:" + inputs.illuminaClip.adapters.path
      + ":"
126             + self.seedMismatches + ":" + self.palindromeClipThreshold
      + ":"
127             + self.simpleClipThreshold + ":" + self.minAdapterLength +
      ":"
128             + self.keepBothReads;
129           } else {
130             return self;
131           }
132         }
133       position: 11
134     doc: Cut adapter and other illumina-specific sequences from the
      read.
135
136   crop:
137     type: int?
138     inputBinding:
139       position: 13
```

```
140       prefix: 'CROP:'
141       separate: false
142     doc: |
143     Removes bases regardless of quality from the end of the read, so
        that the
144     read has maximally the specified length after this step has been
145     performed. Steps performed after CROP might of course further
        shorten the
146     read. The value is the number of bases to keep, from the start of
        the read.
147
148   reads2:
149     type: File?
150     format: edam:format_1930  # fastq
151     inputBinding:
152       position: 6
153     doc: FASTQ file of R2 reads in Paired End mode
154
155   reads1:
156     type: File
157     format: edam:format_1930  # fastq
158     inputBinding:
159       position: 5
160     doc: FASTQ file of reads (R1 reads in Paired End mode)
161
162   avgqual:
163     type: int?
164     inputBinding:
165       position: 101
166       prefix: 'AVGQUAL:'
167       separate: false
168     doc: |
169     Drop the read if the average quality is below the specified level
170
171   trailing:
172     type: int?
173     inputBinding:
174       position: 14
175       prefix: 'TRAILING:'
176       separate: false
177     doc: |
178     Remove low quality bases from the end. As long as a base has a
        value
179     below this threshold the base is removed and the next base (which
        as
```

```
180     trimmomatic is starting from the 3 prime end would be base
    preceding
181     the just removed base) will be investigated. This approach can be
    used
182     removing the special Illumina "low quality segment" regions (which
    are
183     marked with quality score of 2), but we recommend Sliding Window or
184     MaxInfo instead
185
186   maxinfo:
187     type: trimmomatic-max_info.yaml#maxinfo?
188     inputBinding:
189       position: 15
190       valueFrom: |
191         ${ if ( self ) {
192           return "MAXINFO:" + self.targetLength + ":" + self.strictness
    ;
193         } else {
194           return self;
195         }
196         }
197     doc: |
198     Performs an adaptive quality trim, balancing the benefits of
    retaining
199     longer reads against the costs of retaining bases with errors.
200     <targetLength>: This specifies the read length which is likely to
    allow
201     the location of the read within the target sequence to be
    determined.
202     <strictness>: This value, which should be set between 0 and 1,
    specifies
203     the balance between preserving as much read length as possible vs.
204     removal of incorrect bases. A low value of this parameter (<0.2)
    favours
205     longer reads, while a high value (>0.8) favours read correctness.
206
207   end_mode:
208     type: trimmomatic-end_mode.yaml#end_mode
209     inputBinding:
210       position: 3
211     doc: |
212       Single End (SE) or Paired End (PE) mode
213
214 outputs:
215   reads1_trimmed:
```

```
    type: File
    format: edam:format_1930  # fastq
    outputBinding:
      glob: $(inputs.reads1.nameroot).trimmed.fastq

  output_log:
    type: File
    outputBinding:
      glob: trim.log
      label: Trimmomatic log
    doc: |
      log of all read trimmings, indicating the following details:
      the read name
      the surviving sequence length
      the location of the first surviving base, aka. the amount trimmed
     from the start
      the location of the last surviving base in the original read
      the amount trimmed from the end

  reads1_trimmed_unpaired:
    type: File?
    format: edam:format_1930  # fastq
    outputBinding:
      glob: $(inputs.reads1.nameroot).unpaired.trimmed.fastq

  reads2_trimmed_paired:
    type: File?
    format: edam:format_1930  # fastq
    outputBinding:
      glob: |
        ${ if (inputs.reads2 ) {
          return inputs.reads2.nameroot + '.trimmed.fastq';
        } else {
          return null;
        }
        }

  reads2_trimmed_unpaired:
    type: File?
    format: edam:format_1930  # fastq
    outputBinding:
      glob: |
        ${ if (inputs.reads2 ) {
          return inputs.reads2.nameroot + '.unpaired.trimmed.fastq';
        } else {
```

```
260           return null;
261         }
262         }
263
264 baseCommand: [ java, org.usadellab.trimmomatic.Trimmomatic ]
265
266 arguments:
267   - valueFrom: trim.log
268     prefix: -trimlog
269     position: 4
270   - valueFrom: $(runtime.cores)
271     position: 4
272     prefix: -threads
273   - valueFrom: $(inputs.reads1.nameroot).trimmed.fastq
274     position: 7
275   - valueFrom: |
276     ${
277     if (inputs.end_mode == "PE" && inputs.reads2) {
278       return inputs.reads1.nameroot + '.trimmed.unpaired.fastq';
279     } else {
280       return null;
281     }
282     }
283     position: 8
284   - valueFrom: |
285     ${
286     if (inputs.end_mode == "PE" && inputs.reads2) {
287       return inputs.reads2.nameroot + '.trimmed.fastq';
288     } else {
289       return null;
290     }
291     }
292     position: 9
293   - valueFrom: |
294     ${
295     if (inputs.end_mode == "PE" && inputs.reads2) {
296       return inputs.reads2.nameroot + '.trimmed.unpaired.fastq';
297     } else {
298       return null;
299     }
300     }
301     position: 10
302     doc: |
303       Trimmomatic is a fast, multithreaded command line tool that can
    be used to trim and crop
```

```
304     Illumina (FASTQ) data as well as to remove adapters. These
    adapters can pose a real problem
305     depending on the library preparation and downstream application.
306     There are two major modes of the program: Paired end mode and
    Single end mode. The
307     paired end mode will maintain correspondence of read pairs and
    also use the additional
308     information contained in paired reads to better find adapter or
    PCR primer fragments
309     introduced by the library preparation process.
310     Trimmomatic works with FASTQ files (using phred + 33 or phred +
    64 quality scores,
311     depending on the Illumina pipeline used).
312     $namespaces:
313     edam: http://edamontology.org/
314     s: http://schema.org/
315     $schemas:
316     - http://edamontology.org/EDAM_1.16.owl
317     - https://schema.org/docs/schema_org_rdfa.html
318     s:license: "https://www.apache.org/licenses/LICENSE-2.0"
319     s:copyrightHolder: "EMBL - European Bioinformatics Institute"
```

NGSPipes trimmomatic command descriptor

Listing H.1 presents an example of a possible *Trimmomatic* command descriptor for Trimmomatic.

Listing H.1: NGSPipes *Trimmomatic* tool descriptor example.

```
{
  "name": "Trimmomatic",
  "author": "UNKNOWN",
  "version": "0.0.33",
  "description": "It works with FASTQ (using phred + 33 or phred + 64
   quality scores, depending on the Illumina pipeline used), either
   uncompressed or gzipp'ed FASTQ. Use of gzip format is determined
   based on the .gz extension. For single-ended data, one input and
   one output file are specified, plus the processing steps. For
   paired-end data, two input files are specified, and 4 output files,
    2 for the paired output where both reads survived the processing,
   and 2 for corresponding unpaired output where a read survived, but
   the partner read did not.",
  "documentation": [
    "http://www.usadellab.org/cms/?page=trimmomatic",
    "http://bioinformatics.oxfordjournals.org/content/early/2014/04/01/
   bioinformatics.btu170"
  ],
  "commands": [
```

```
11    {
12      "name": "trimmomatic",
13      "command": "java -jar /trimmomatic-0.33.jar",
14      "description": "It works with FASTQ (using phred + 33 or phred +
      64 quality scores, depending on the Illumina pipeline used), either
       uncompressed or gzipp'ed FASTQ. Use of gzip format is determined
      based on the .gz extension. For single-ended data, one input and
      one output file are specified, plus the processing steps. For
      paired-end data, two input files are specified, and 4 output files,
       2 for the paired output where both reads survived the processing,
      and 2 for corresponding unpaired output where a read survived, but
      the partner read did not.",
15      "recommended_mem": 1024,
16      "recommended_cpu" : 1,
17      "recommended_disk" : 1024,
18      "parameters": [
19        {
20          "name": "mode",
21          "type": "enum",
22          "required": true,
23          "description": "For single-ended data, one input and one
      output file are specified, plus the processing steps. For paired-
      end data, two input files are specified, and 4 output files, 2 for
      the 'paired' output where both reads survived the processing, and 2
       for corresponding 'unpaired' output where a read survived, but the
       partner read did not.",
24          "values" : ["SE", "PE", "org.usadellab.trimmomatic.
      TrimmomaticSE", "org.usadellab.trimmomatic.TrimmomaticPE"]
25        }, {
26          "name": "threads",
27          "type": "int",
28          "required": false,
29          "description": "indicates the number of threads to use, which
       improves performance on multi-core computers. If not specified, it
       will be chosen automatically",
30          "prefix" : "-threads"
31        }, {
32          "name": "quality",
33          "type": "string",
34          "required": true,
35          "description": "phred + 33 or phred + 64 quality scores"
36        }, {
37          "name": "trimlog",
38          "type": "string",
39          "required": false,
```

```
40          "description": "specifies the path to the log file. Log file
    contains info as : \n-the read name\n-the surviving sequence length
    \n-the location of the first surviving base, aka. the amount
    trimmed from the start\n-the location of the last surviving base in
     the original read-\nthe amount trimmed from the end",
41          "prefix" : "-trimlog"
42        }, {
43          "name": "inputFile1",
44          "type" : "file",
45          "required": true,
46          "description": "Specifies the path to the fastq input file.",
47          "depends" : "$mode",
48          "dependent_values" : ["SE", "org.usadellab.trimmomatic.
    TrimmomaticSE"]
49        }, {
50          "name": "basein",
51          "type": "flag",
52          "required": true,
53          "description": "Specifies the path to the fastq input file.",
54          "prefix" : "-basein",
55          "depends" : "$mode",
56          "dependent_values" : ["PE", "org.usadellab.trimmomatic.
    TrimmomaticPE"]
57        }, {
58          "name": "pairedInput1",
59          "type" : "file",
60          "required": true,
61          "description": "Specifies the path to the input file 1 of
    paired mode.",
62          "depends" : "$basein"
63        }, {
64          "name": "pairedInput2",
65          "type" : "file",
66          "required": false,
67          "description": "Specifies the path to the input file 2 of
    paired mode.",
68          "depends" : "$pairedInput1"
69        }, {
70          "name": "baseout",
71          "type": "flag",
72          "required": true,
73          "description": "Specifies the path to the fastq input file.",
74          "prefix" : "-baseout",
75          "depends" : "$mode",
```

```
76          "dependent_values" : ["PE", "org.usadellab.trimmomatic.
    TrimmomaticPE"]
77        }, {
78          "name": "output",
79          "type": "string",
80          "required": true,
81          "description": "Specifies the name of output file.",
82          "depends" : "$mode",
83          "dependent_values" : ["SE", "org.usadellab.trimmomatic.
    TrimmomaticSE"]
84        }, {
85          "name": "pairedOutput1",
86          "type": "string",
87          "required": true,
88          "description": "Specifies the name of paired output file 1.",
89          "depends" : "$baseout"
90        }, {
91          "name": "unpairedOutput1",
92          "type": "string",
93          "required": false,
94          "description": "Specifies the name of unpaired output file 1
    .",
95          "depends" : "$baseout"
96        }, {
97          "name": "pairedOutput2",
98          "type": "string",
99          "required": false,
100         "description": "Specifies the name of paired output file 2.",
101         "depends" : "$baseout"
102       }, {
103         "name": "unpairedOutput2",
104         "type": "string",
105         "required": false,
106         "description": "Specifies the name of unpaired output file 2
    .",
107         "depends" : "$baseout"
108       }, {
109         "name" : "ILLUMINA_CLIP",
110         "type" : "composed",
111         "required" : false,
112         "description" : "This step is used to find and remove
    Illumina adapters.",
113         "prefix" : "ILLUMINA_CLIP:",
114         "separator" : ":",
115         "sub_parameters": [
```

```
116             {
117               "name": "fastaWithAdaptersEtc",
118               "type" : "file",
119               "required": true,
120               "description": "Specifies the path to a fasta file
    containing all the adapters, PCR sequences etc. The naming of the
    various sequences within this file determines how they are used."
121             }, {
122               "name": "seedMismatches",
123               "type": "int",
124               "required": true,
125               "description": "Specifies the maximum mismatch count
    which will still allow a full match to be performed."
126             }, {
127               "name": "palindromeClipThreshold",
128               "type": "int",
129               "required": true,
130               "description": "Specifies how accurate the match between
    the two 'adapter ligated' reads must be for PE palindrome read
    alignment."
131             },
132             {
133               "name": "simpleClipThreshold",
134               "type": "int",
135               "required": true,
136               "description": "Specifies how accurate the match between
    any adapter etc. sequence must be against a read."
137             },{
138               "name": "minAdapterLength",
139               "type": "int",
140               "required": false,
141               "description": "In addition to the alignment score,
    palindrome mode can verify that a minimum length of adapter has
    been detected. If unspecified, this defaults to 8 bases, for
    historical reasons. However, since palindrome mode has a very low
    false positive rate, this can be safely reduced, even down to 1, to
     allow shorter adapter fragments to be removed."
142             },{
143               "name": "keepBothReads",
144               "type": "boolean",
145               "required": false,
```

```
146             "description": "After read-though has been detected by
    palindrome mode, and the adapter sequence removed, the reverse read
     contains the same sequence information as the forward read, albeit
     in reverse complement. For this reason, the default behaviour is
    to entirely drop the reverse read. By specifying "true" for this
    parameter, the reverse read will also be retained, which may be
    useful e.g. if the downstream tools cannot handle a combination of
    paired and unpaired reads."
147           }
148         ]
149       }, {
150         "name" : "SLIDINGWINDOW",
151         "type" : "composed",
152         "required" : false,
153         "description" : "Perform a sliding window trimming, cutting
    once the average quality within the window falls below a threshold.
     By considering multiple bases, a single poor quality base will not
     cause the removal of high quality data later in the read. ",
154         "prefix" : "SLIDINGWINDOW:",
155         "separator" : ":",
156         "sub_parameters": [
157           {
158             "name": "windowSize",
159             "type": "int",
160             "required": true,
161             "description": "Specifies the number of bases to average
    across."
162           }, {
163             "name": "requiredQuality",
164             "type": "int",
165             "required": true,
166             "description": "Specifies the average quality required."
167           }
168         ]
169       }, {
170         "name" : "MAXINFO",
171         "type" : "composed",
172         "required" : false,
173         "description" : "Performs an adaptive quality trim, balancing
     the benefits of retaining longer reads against the costs of
    retaining bases with errors.",
174         "prefix" : "MAXINFO:",
175         "separator" : ":",
176         "sub_parameters": [
177           {
```

```
178             "name": "targetLength",
179             "type": "int",
180             "required": true,
181             "description": "This specifies the read length which is
    likely to allow the location of the read within the target sequence
     to be determined."
182           }, {
183             "name": "strictness",
184             "type": "float",
185             "required": true,
186             "description": "This value, which should be set between 0
     and 1, specifies the balance between preserving as much read
    length as possible vs. removal of incorrect bases. A low value of
    this parameter (<0.2) favours longer reads, while a high value (>0.
    8) favours read correctness."
187           }
188         ]
189       }, {
190         "name": "leadingQuality",
191         "type": "int",
192         "required": false,
193         "description": "Specifies the minimum quality required to
    keep a base.",
194         "prefix": "LEADING:"
195       }, {
196         "name": "trailingQuality",
197         "type": "int",
198         "required": false,
199         "description": "Specifies the minimum quality required to
    keep a base.",
200         "prefix": "TRAILING:"
201       }, {
202         "name": "cropLength",
203         "type": "int",
204         "required": false,
205         "description": "The number of bases to keep, from the start
    of the read.",
206         "prefix": "CROP:"
207       }, {
208         "name": "headcropLength",
209         "type": "int",
210         "required": false,
211         "description": "The number of bases to remove from the start
    of the read.",
212         "prefix": "HEADCROP:"
```

```
213         }, {
214           "name": "minlenLength",
215           "type": "int",
216           "required": false,
217           "description": "Specifies the minimum length of reads to be
    kept.",
218           "prefix": "MINLEN:"
219         }, {
220           "name": "topHred33",
221           "type": "flag",
222           "required": false,
223           "description": "Specifies the minimum length of reads to be
    kept.",
224           "prefix": "TOPHRED33"
225         }, {
226           "name": "topHred64",
227           "type": "flag",
228           "required": false,
229           "description": "Specifies the minimum length of reads to be
    kept.",
230           "prefix": "TOPHRED64"
231         }
232       ],
233       "outputs": [
234         {
235           "name": "outputFile",
236           "description": "",
237           "type" : "file",
238           "value": "$output"
239         },
240         {
241           "name": "pairedOutputFile1",
242           "description": "",
243           "type" : "file",
244           "value": "$pairedOutput1"
245         },
246         {
247           "name": "UnpairedOutputFile1",
248           "description": "",
249           "type" : "file",
250           "value": "$unpairedOutput1"
251         },
252         {
253           "name": "pairedOutputFile2",
254           "description": "",
```

```
            "type" : "file",
            "value": "$pairedOutput2"
          },
          {
            "name": "UnpairedOutputFile2",
            "description": "",
            "type" : "file",
            "value": "$unpairedOutput2"
          }
        ]
      }
    ]
}
```

I

Mesos cluster installation

Listing I.1 presents an script with the steps followed to install *Mesos* on minimal *Centos*.

Listing I.1: *Mesos* installation steps

```
#!/bin/bash
#Add mesosphere repository
  sudo rpm -Uvh http://repos.mesosphere.com/el/7/noarch/RPMS/mesosphere
    -el-repo-7-1.noarch.rpm

#Install Mesos
  sudo yum -y install mesos

#Install zookeeper
  sudo rpm -Uvh http://archive.cloudera.com/cdh5/one-click-install/
    redhat/7/x86_64/cloudera-cdh-5-0.x86_64.rpm
  sudo yum -y install zookeeper zookeeper-server java-1.8.0-openjdk

#Initialize zookeeper
  sudo -u zookeeper zookeeper-server-initialize --myid=1
  sudo service zookeeper-server start

#Test zookeeper
  /usr/lib/zookeeper/bin/zkCli.sh
  create /test 1
  get /test
  set /test 2
```

```
21   get /test
22   delete /test
23   quit
24   sudo service zookeeper-server stop
25   sudo service zookeeper-server start
26
27 # Mesos
28 #sudo vi /etc/mesos/zk # put zk://0.0.0.0:2181/mesos
29 #echo 10.0.2.8 | sudo tee /etc/mesos-master/ip
30 #sudo cp /etc/mesos-master/ip /etc/mesos-master/hostname
31
32   sudo service mesos-master start
33   sudo service mesos-master status
34   sudo netstat -nlp | grep mesos
35
36 # To install
37   sudo yum install -y firewalld
38   sudo systemctl start firewalld.service
39
40 # Enabling Mesos and Zookeeper port
41   sudo firewall-cmd --zone=public --add-port=5050/tcp --permanent
42   sudo firewall-cmd --zone=public --add-port=2181/tcp --permanent
43
44 # restart firewall
45   sudo firewall-cmd --reload
46
47 # Chronos
48   sudo yum -y install chronos
49   sudo service chronos start
50   sudo firewall-cmd --zone=public --add-port=4400/tcp --permanent
51   sudo firewall-cmd --reload
52
53 # Slave with docker
54 #Add mesosphere repository- Slave in another node
55   sudo rpm -Uvh http://repos.mesosphere.com/el/7/noarch/RPMS/mesosphere
       -el-repo-7-1.noarch.rpm
56
57 #Install Mesos- Slave in another node
58   sudo yum -y install mesos
59
60 #Docker
61   sudo yum -y install docker
62   sudo service docker start
63
64 #CONFIGURATION
```

```
65    sudo vi /etc/sysconfig/docker
66  # OPTIONS="--selinux-enabled --dns 8.8.8.8 --dns 8.8.4.4"
67
68  #CONFIGURING MESOS SLAVE FOR DOCKER
69    echo "docker,mesos" | sudo tee /etc/mesos-slave/containerizers
70    echo "5mins" | sudo tee /etc/mesos-slave/executor_registration_
        timeout
71
72  # Slave on same host of master
73  #sudo cp /etc/mesos-master/hostname /etc/mesos-slave/hostname
74  #sudo cp /etc/mesos-master/hostname /etc/mesos-slave/ip
75
76  #Slave in another node
77    sudo vi /etc/mesos/zk
78    #zk://192.168.1.127:2181/mesos
79
80  # To install
81    sudo yum install -y firewalld
82    sudo systemctl start firewalld.service
83
84    sudo firewall-cmd --zone=public --add-port=5051/tcp --permanent
85    sudo firewall-cmd --reload
86
87    sudo service mesos-slave start
88
89  # NFS -> share files between master and slaves
90    # Waranty SSH and NFS services are allowed to connect
91    sudo firewall-cmd --permanent --zone=public --add-service=ssh
92    sudo firewall-cmd --permanent --zone=public --add-service=nfs
93    sudo firewall-cmd --reload
94
95    # Install nfs
96    sudo yum -y install nfs-utils
97
98    # MESOS MASTER
99      # Enable and start nfs server
100     sudo systemctl enable nfs-server.service
101     sudo systemctl start nfs-server.service
102
103     # Share folder
104     mkdir /home/centos/pipes
105     # put nfsnobody as owner of the folder
106     sudo chown nfsnobody:nfsnobody /home/centos/pipes/*
107     sudo chmod -R 755 /home/centos/pipes
108     nano /etc/exports
```

```
109     #/home/centos/pipes/        slave_ip(rw,sync,no_subtree_check)
110     exportfs -a
111
112   # MESOS SLAVE
113     mount master_ip:/home/centos/pipes /home/centos/pipes
```