INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Bioinformatic pipeline specification language and sharing system

BRUNO MIGUEL DAS NEVES DANTAS

(Licenciado)

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

Orientadores : Doutora Cátia Raquel Jesus Vaz
Doutor José Manuel de Campos Lages Garcia Simão

Júri:

Presidente: Doutor Carlos Jorge de Sousa Gonçalves

Vogais: Doutor Nuno Miguel Soares Datia
Doutora Cátia Raquel Jesus Vaz

AGO, 2019

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Bioinformatic pipeline specification language and sharing system

BRUNO MIGUEL DAS NEVES DANTAS

(Licenciado)

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

Orientadores : Doutora Cátia Raquel Jesus Vaz
Doutor José Manuel de Campos Lages Garcia Simão

Júri:

Presidente: Doutor Carlos Jorge de Sousa Gonçalves

Vogais: Doutor Nuno Miguel Soares Datia
Doutora Cátia Raquel Jesus Vaz

AGO, 2019

Aos meus pais e irmã

Acknowledgments

Aos meus orientadores, por todo o apoio e disponibilidade que sempre tiveram ao longo da realização desta dissertação. A todos os meus amigos que sempre me motivaram para finalizar esta etapa. Em particular, um grande agradecimento aos meus pais e irmã, sem eles nada disto seria possível.

Abstract

The project *Infrastructure to support the execution of workflows for bioinformatics*, from now on referred to as *NGSPipes*, is a platform for creation and execution of *pipelines* of NGS(Next Generations Sequencing) data. *NGSPipes* project[24] was developed within final thesis of Computer Science degree at Instituto Superior de Engenharia de Lisboa. The main goal of *NGSPipes* is to help scientific community to develop and execute biological *pipelines* without the need for programming knowledge.

The project *Bioinformatic pipeline specification language and sharing system*, is an extension of *NGSPipes* project. The main goal of this project is to extend *NGSPipes* by adding essential features for the development and sharing of *pipelines*. To achieve this extension, we compared *NGSPipes* against other systems and defined a set of essential features to develop *pipelines*. After this comparison we defined a new language specification keeping the advantages of previous *NGSPipes* version and adding parallelism and argument definition primitives. To enable cooperation between community members, it was developed a platform to allow users to share tools and *pipelines*.

Keywords: NGSPipes; Workflow; Sharing; Domain Specific Language; Scientific Workflow System.

Resumo

O projeto *Infraestrutura de suporte à execução de fluxos de trabalho para a bioinformatica*, daqui em diante designado *NGSPipes*, é uma ferramenta para criação e execução de fluxos de execução de processamento e análise em serie de dados NGS(Next Generations Sequencing). O projeto *NGSPipes*[24] foi desenvolvido no âmbito da tese final de curso da Licenciatura em Engenharia Informática e de Computadores no Instituto Superior de Engenharia de Lisboa. O projeto *NGSPipes* tem como principal objetivo, auxiliar a comunidade científica na criação e execução de *pipelines* de cariz biológica sem a necessidade de conhecimentos de programação.

O projeto *Linguagens e modelos de partilha para fluxos de trabalho de ferramentas bioinformaticas*, é uma extensão do projeto *NGSPipes*. Este projeto tem como principal objetivo estender o *NGSPipes* adicionando funcionalidades essenciais para o desenvolvimento e partilha de *pipelines*. Para realizar esta extensão, o sistema *NGSPipes* foi comparado com outros sistemas de modo a definir um conjunto de características essenciais ao desenvolvimentos de *pipelines*. Após esta comparação, definimos uma nova linguagem de especificação mantendo as vantagens da versão anterior do *NGSPipes* e adicionando primitivas de paralelismo e definição de argumentos. Para permitir a cooperação entre membros da comunidade, foi desenvolvida uma plataforma para que os utilizadores possam partilhar as suas ferramentas e *pipelines*.

Palavras-chave: NGSPipes; Fluxo de trabalho; Partilha; Linguagem Especifica de Domínio; Sistema de Fluxo de Trabalho Científico.

Contents

List of Figures

List of Tables

Listing

Glossary

adapter Short DNA molecule that can be linked to the ends of other DNA molecules. 7

argument value supplied on invocation of a tool to configure its execution. 2

DNA deoxyribonucleic acid, is the hereditary material in humans. 1

epidemic surveillance practice in which the spread of disease is predicted, observed and minimized. This practice also aims to increase knowledge about which factors contribute to such circumstances. 5

genome genetic material of an organism. 7

NGS Next Generation Sequencing is a technique of determining the nucleic acid sequence. 1

nucleotide organic molecules that serve as the monomer units for DNA and RNA. 9

pathogen infectious agent, germ. 7

pipeline consists of an orchestrated set of steps. Each step invokes a command line tool. 1

reads small sequences of DNA. 1

tool command line software. 1

workflow consists of an orchestrated set of steps. Each step invokes a command line tool. 1

1

Introduction

In recent years, *workflows* gained more attention in scientific community[36][25] to support scientists work. Generally scientific *workflows* are often adapted to a particular application domain. In this context, *workflows* implement scientific simulations, experiments and computations typically dealing with huge amounts of data. Scientists model, execute, monitor and analyse *workflows*. In this document, scientific *workflows* will be approached in the field of bioinformatics.

Bioinformatics corresponds to the appliance of computer science techniques in biology and medicine areas, in order to help on data analysis. Experiments carried out today, both at the research and industry level, use NGS (Next Generation Sequencing) techniques (e.g. Reproductive Health Research[22]) that produce small pieces of sequencing of the original DNA (Deoxyribonucleic acid) sample known as `reads`. These `reads` are further processed and refined by a series of interrelated computational analysis and visualization tasks, applied to large amounts of biological data. These analysis, referred to as *pipeline* or *workflow*, start with bulky raw text sequences and end with detailed, structural, functional and evolutionary results. These *pipelines* involve the use of various *software* (tools) and resources, with the result produced by one tool being transmitted as input to other tool.

The various industrial and scientific actors in these experiments want to be able

to reuse the same data-processing logic (*pipeline*), with different *inputs* and different concrete tools, with the only restriction of the data format being the same as well as the signatures of the algorithms. This way it would be possible to reproduce the execution of these *pipelines*, in an automatic manner.

1.1 Scientific Workflow System

Bioinformatic analysis consist of applying series of transformations to huge data files, commonly referred to as *workflows* or *pipelines*. To agile this process of developing and executing *pipelines*, different Scientific Workflow System have emerged.

A Scientific Workflow System (SWS) is a platform which permits users to develop and execute *pipelines* composed of sequential and parallel steps.

There are multiple SWSs options such as *Galaxy*[30], *Nextflow*[32], *Ruffus*[34], *NGS-Pipes*[33] among others. Since there are so many options, it's not easy to decide which SWS to work with. When selecting a SWS we should take in consideration the following capabilities:

- Reproducibility - ability to repeat *pipeline* execution with the same or different arguments. It shouldn't be necessary to change *pipeline* itself in order to change the execution arguments;
- Portability - it should be possible to run a *pipeline* obtaining same result independently from the execution machine;
- Extensibility - ability to add new tools without recompiling previous code;
- Reusability - it should be possible to develop nested *pipelines*, meaning that we could use other *pipelines* inside a *pipeline*

Commonly SWSs are composed of two main components: `Language` (to describe *pipelines*) and *Engine* (to execute *pipelines*). This language can be either graphical or textual. On this thesis we will be focused on SWSs with a textual language component.

1.2 NGSPipes

NGSPipes is a Scientific Workflow System developed in the context of a computer science degree thesis. From now on we will use the term ***NGSPipesV1***, to refer the previous version of *NGSPipes*, and ***NGSPipesV2*** to refer the new version developed on the context of this thesis.

NGSPipesV1 was developed with three main goals:

- Help scientists to develop *pipelines* without the need for programming knowledge;
- Help programmers developing biological *pipelines*;
- Simplify the execution of *pipelines* and the management of tools required to run these *pipelines*

To answer to this three objectives, *NGSPipesV1* has four modules: `Repository`, `DSL`, *Engine* and `Editor`. On figure 1.1 we can observe how these modules relates with each other.

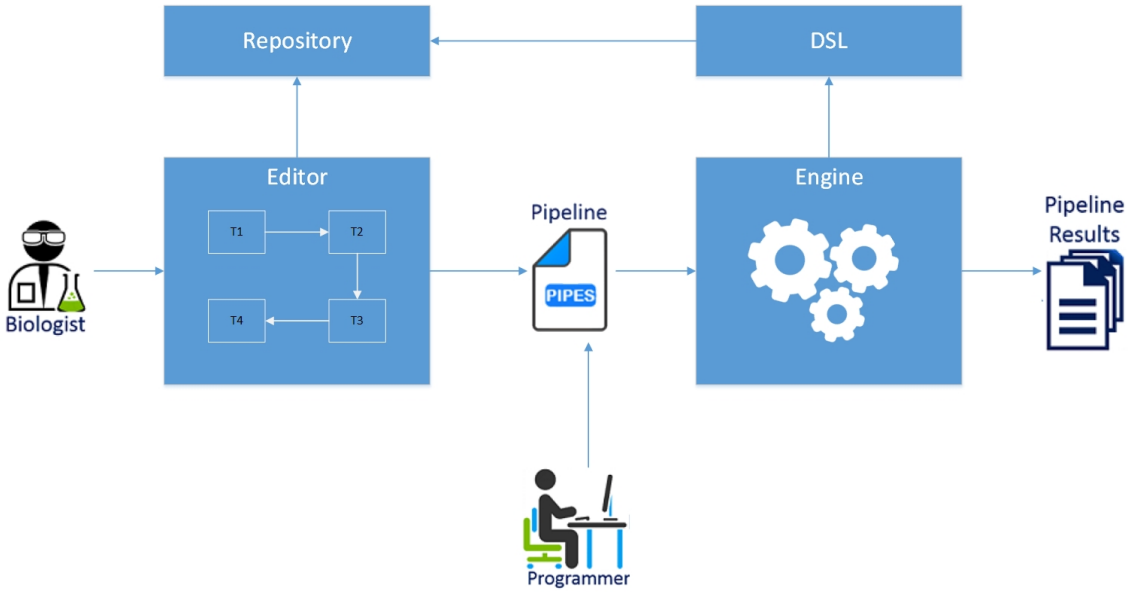

Figure 1.1: Modules of *NGSPipesV1*

NGSPipesV1 offers a language specification to define *pipelines* (`DSL` module). These *pipelines* could be written directly (by programmers) or produced through the

`Editor` (by biologists) and consumed by the *Engine*. This language specification, differently from other SWSs, is based on descriptors which are files that store tools meta-data. This meta-data is consumed by the *Engine* in order to build the command line to be executed. `DSL` and `Editor` use `Repository` to obtain the meta-data associated to the tools used within a *pipeline*.

NGSPipesV1, when compared with other state of the art SWSs, lacks some capabilities such as parallelism and nested *pipelines*. Although these limitations, *NGSPipesV1* has some advantages with its tools meta-data approach. By having meta-data associated with tools, *NGSPipesV1* permits to developers to have a unique syntax when defining arguments and also validate these arguments integrity. It is important to extend *NGSPipesV1* keeping its advantages and solving its limitations.

1.3 Thesis Statement

The aim of this thesis is to extend *NGSPipesV1* by proposing a language specification and a *pipeline* sharing platform to fulfil basic requirements of a SWS. The language specification must accommodate features such as **argument definition** and **task/data parallelism**.

The first step is to evaluate the state of the art and compare *NGSPipesV1* against different systems. After collecting *NGSPipesV1*'s limitations, our goal is to extend *NGSPipesV1* and solve these limitations. Along with this project is being developed another project thesis, *Parallel execution of workflows using bioinformatics tools* [26], which complements this thesis by implementing *Engine* module.

We defined the following objectives for the new language specification:

- Add the ability do produce task or data parallel *pipelines* (sections 3.2.2.5 and 3.2.2.6);
- Allow users to define *pipelines* logic without concrete data (section 3.2.2.2);
- Allow the ability to invoke another *pipeline* on a *pipeline*'s step (section 3.2.2.7);
- Use multiple repositories on a *pipeline* definition;
- Keep meta-data concept;

- Keep repository concept

We defined the following objectives for the sharing platform:

- Publish repositories external to our platform;
- Create new repositories;
- Edit repositories content (add/delete tools and *pipelines*)

To test our solution, a case study based on epidemic surveillance was developed. This case study results in a *pipeline* which represents a normal flow to process NGS data. We can see this case study in detail on Chapter 2.

1.4 Outline

This document is organized into five chapters. This first chapter introduces the context in which this thesis surged and the origin of *NGSPipesV1*. On second chapter we will cover the developed case study and its variants. On third chapter we will compare *NGSPipesV1* with other SWSs through a set of features. On fourth chapter we will discuss *NGSPipesV2* and how limitations found on previous chapter were solved. On fifth chapter we have final remarks and some goals to be achieved as future work.

2

Case Study

To evaluate different SWSs it was elaborated a case study based on epidemiological surveillance. The data used on this case study comes from the application of NGS(Next Generation Sequence) techniques to the genomes of pathogens like *Streptococcus pneumoniae*. By applying a processing *pipeline* to this data, it is possible to identify strains, the antibiotic resistance profile and the presence of virulence determinants.

2.1 Tools

The developed *pipeline* uses the following tools:

- **Trimmomatic** [3] based on FASTQ [29] files (`reads` containers or small DNA sequences) allows the extraction of `adapters` and `reads` with low quality;
- **Velvet** [4] based on FASTQ files, generally with low quality `reads` already filtered, obtains the genome schema which is composed of long DNA sequences resultant of multiple `reads`(`contigs`);
- **Blast** [2] based on `contigs` obtained from *Velvet*, allows to determine genes sequences and annotate them by comparison with multiple databases

In figure 2.1 we can observe the sequential steps of the developed *pipeline*.

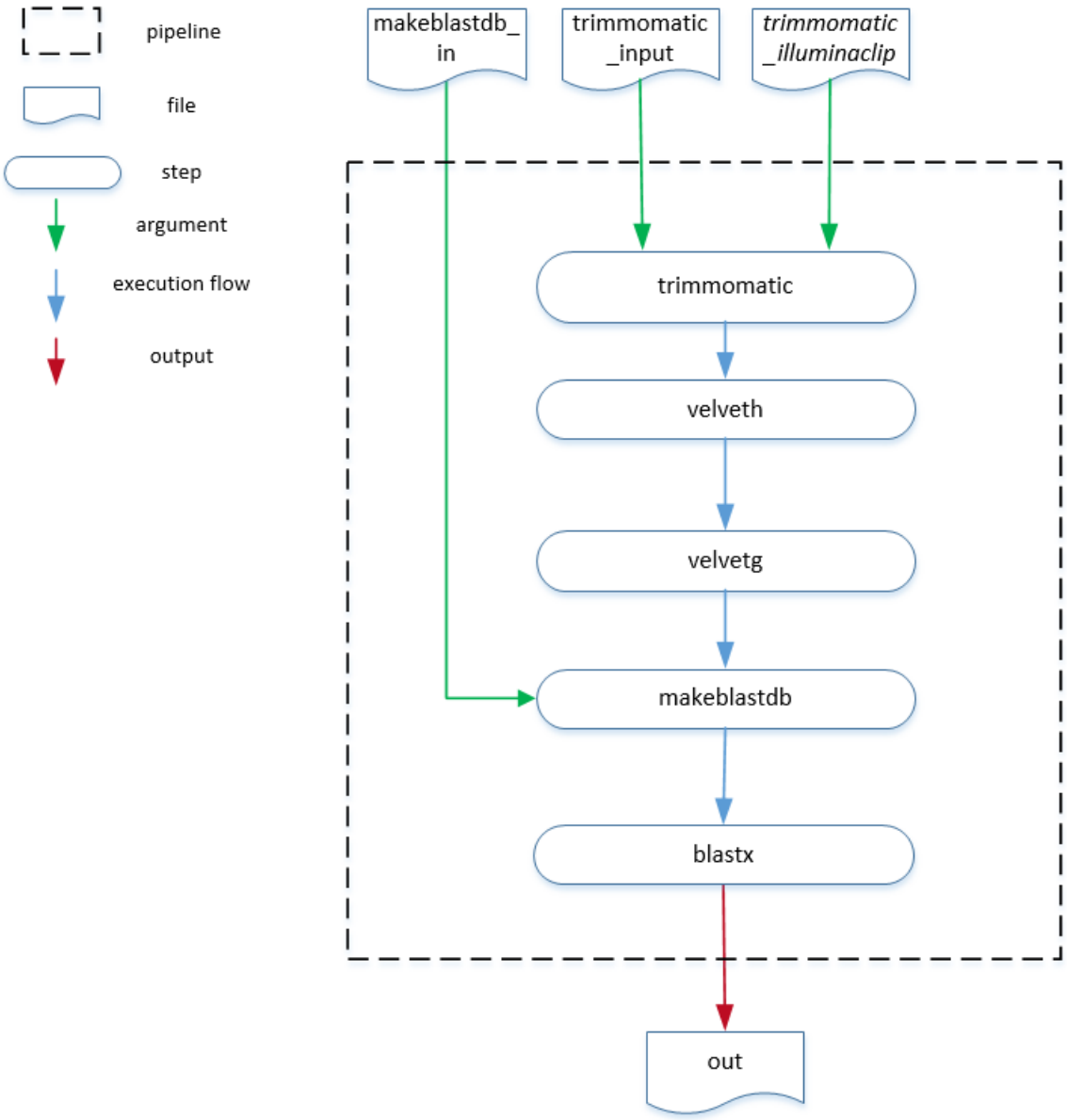

Figure 2.1: Sequential *pipeline* schematic.

This *pipeline* is composed of 5 steps:

- **trimmomatic** receives a `.fastq` file and removes `reads` with low quality;
- **velveth** receives a `.fastq` file filtered by `trimmomatic` step and constructs a dataset;

- **velvetg** based on the `velveth`'s output creates the genome schema;

- **makeblastdb** builds a database to be used by `blastx` step;

- **blastx** uses `makeblastdb`'s database and `velvetg` outputs to translate a nucleotide query and searches it against protein subject sequences or a protein database.

This *pipeline* receives 3 arguments:

- **trimmomatic_input** - a `.fastq` file which will be passed to `input_file` input of `trimmomatic` step;

- **trimmomatic_illuminaclip** - a `.fa` file which will be passed to `illuminaclip_file` input of `trimmomatic` step;

- **makeblastdb_in** - a `.pro` file which will be passed to `in` input of `makeblastdb` step

This *pipeline* produces 1 output:

- **out** - `out` output from `blastx` step

The order given by the arrows representing the execution flow, are inferred from the chain dependencies between step's inputs and outputs. On figure 2.2 we can see all dependencies between steps.

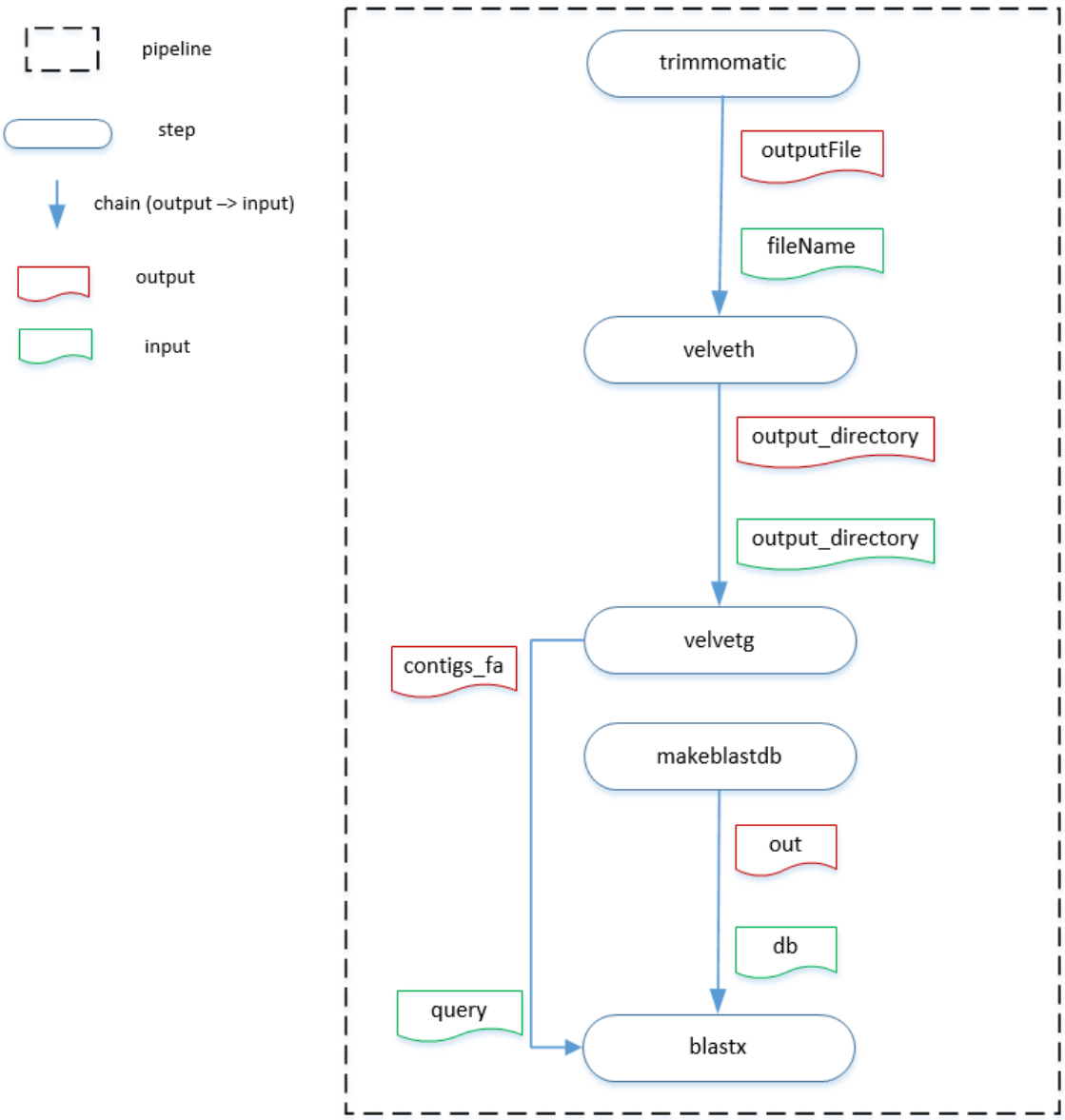

Figure 2.2: Chains *pipeline* schematic.

This *pipeline* contains 4 chains:

- **outputFile -> fileName** - the `outputFile` output from `trimmomatic` step is passed to `fileName` input of `velveth` step;
- **output_directory -> output_directory** - the `output_directory` output from `velveth` step is passed to `output_directory` input of `velvetg` step;

- **contigs_fa -> query** - the `contigs_fa` output from `velvetg` step is passed to `query` input of `makeblastdb` step;
- **out -> db** - the `out` output from `makeblastdb` step is passed to `db` input of `blastx` step

In order to test parallel capabilities of different systems, two variants of this case study were developed (task and data parallel variants).

2.2 Task Parallel Variant

The first variant is the same example but we will take steps dependencies in consideration. If we analyse these dependencies we will realize that `trimmomatic` step and `makeblastdb` step are independent and can be executed in parallel as it can be seen in figure 2.3.

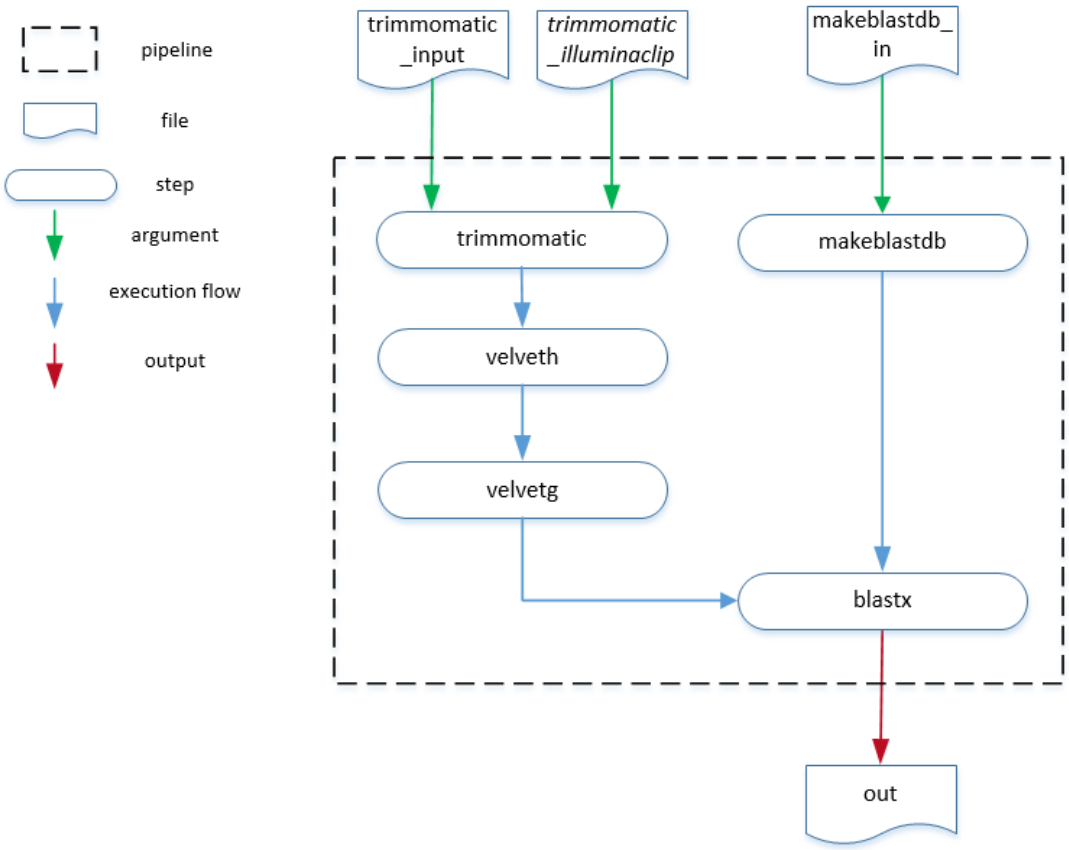

Figure 2.3: Task parallel *pipeline* schematic

2.3 Data Parallel Variant

The second variant mimics the previous example, but this time running `makeblastdb` step against three different databases. Similarly to previous example, `trimmomatic` and the three `makeblastdb` steps can be executed in parallel. In figure 2.4 we can observe a schematic for this variant.

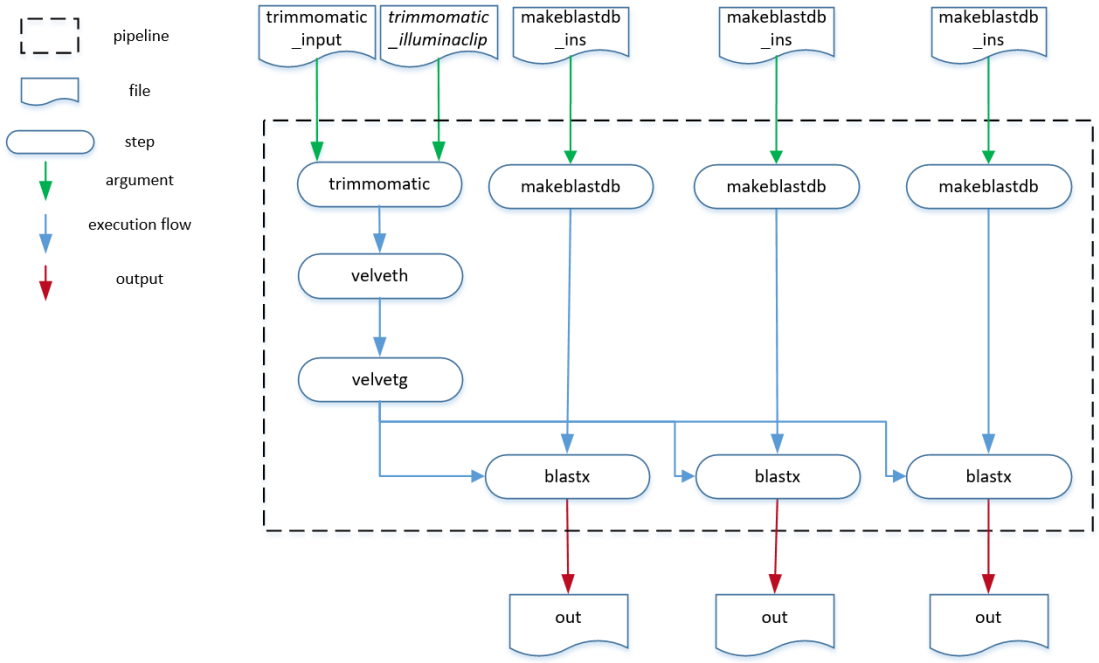

Figure 2.4: Data parallel *pipeline* schematic

3

Systems Comparison

In recent years multiple efforts[36][28] have been made in order to make the construction and execution of data analysis *pipelines* an easier task. Different Scientific Workflow Systems emerged with different approaches. In this chapter we will see the state of the art among some of the most used SWSs. To evaluate the state of the art we will compare these systems focusing on their *pipeline* language specification and the available *pipelines*/*tools* sharing mechanisms.

3.1 Scientific Workflow Systems

There are multiple SWSs available. In order to study the state of the art, some systems were selected to be studied. The selected SWSs are: *NGSPipesV1*, *Nextflow*, *CWL*[31], *Ruffus* and *Swift*[35]. These systems were selected due to the acceptance and usage among the community.

3.1.1 NGSPipesV1

NGSPipesV1 is a framework to design and execute *pipelines*, relying on state of the art of cloud technologies to execute them without users need to configure, install and manage tools.
NGSPipesV1 language was built with main purpose of users don't need to know

the syntax to invoke a command line tool.

3.1.2 Nextflow

Nextflow is a language modelled around the *UNIX*[23] pipe concept. *Nextflow* language was built to simplify writing portable, parallel and scalable *pipelines*. *Nextflow* seats on top of *Groovy*[12].

3.1.3 CWL

CWL is a specification to describe command line tools and create *pipelines* by connecting tools through inputs and outputs. Since *CWL* is a specification, artifacts described using *CWL* are portable across a variety of platforms that support the *CWL* standard.

3.1.4 Ruffus

Ruffus is a open-source *pipeline Python*[18] library. *Ruffus* was designed to allow data analysis to be automated with the least effort.

3.1.5 Swift

Swift is a open-source *software* which allows users to develop *pipelines* with an implicitly parallel programming language. *Swift* helps to distribute program execution across clusters, clouds, grids, and supercomputers.

3.2 Pipeline Specification Languages

In order to classify and compare each language and its syntax, a set of features were defined, namely:

- Methodology - DSL, Command Invocation;
- Syntax - Variables, Arguments, Chain Input with Outputs, Step Dependency, Task Parallelism, Data Parallelism, Nested *Pipelines*

All comparisons that involve *NGSPipes*, on this chapter, were made based on *NGSPipesV1* language.

3.2.1 Methodology

When defining a programming language, we can take different approaches by implementing it on top of an existing programming language or develop a new syntax directed to the context of a problem. In this chapter we classify each language's methodology.

3.2.1.1 DSL

Domain Specific Language (DSL)[27] is a language developed to turn the solution of a problem with a specific domain simpler and more comprehensible. DSLs can be divided in two types: internal or external. Internal DSLs are built on top of a programming language, such as *Java*[15] or *C#*[7]. An example of an internal DSL is *LINQ*[17], which is a *C#* library that allows developers to have a fluent syntax to query data. On Listing 3.1 we can observe an example of *LINQ*. External DSLs define their own syntax instead of being built on top of a programming language, meaning that they need a new interpreter. An example of an external DSL is *JSON*[14], which has its own syntax to represent data. On Listing 3.2 we can observe an example of *JSON*.

Listing 3.1: *LINQ* example

```
1  int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
2
3  var even =
4    from num in numbers
5    where (num % 2) == 0
6    select num;
7
8  foreach (int num in even)
9    Console.WriteLine(num);
10
```

Listing 3.2: *JSON* example

```
{
  "name":"John" ,
  "car":"seat"
}

```

Based on this definition, *Nextflow* and *Ruffus* are considered internal DSLs, since *pipelines* are written with *Groovy* and *Python* respectively. *NGSPipesV1* and *Swift* have no affiliation with other programming languages meaning that they are classified as external DSLs. *CWL* is an external DSL since it defines *pipelines* through *JSON/YAML*[21] files.

In table 3.1 we can see a resume of DSL comparison.

3.2.1.2 Command Invocation

A *pipeline* is composed by a set of steps, in which a command line tool is invoked. Scientific workflow systems offer DSLs which help users to build the command line to invoke the tool and orchestrate these steps. There are two approaches when defining how to declare the command line we want to execute:

- **Meta-data**, each tool has a description with its meta-data. Developer specifies that on a step he wants to run a specific tool, but the command is built by the *Engine*;
- **Raw**, there is no tool description concept. The command to be executed is explicit on *pipeline*

As we can see on listings 3.10, 3.13 and 3.15, *Nextflow*, *Ruffus* and *Swift* specify the command to be executed on step's declaration. *CWL* and *NGSPipesV1* instead of declaring how a tool is executed, declares which tool is to be executed. When we compare both approaches, we can be tempted to say that the meta-data approach is worse because adds the overhead of having meta-data associated to each tool. Although this overhead, this approach permits developers to have more abstraction and validation of tools usage. On this section we will be focused on meta-data oriented DSLs and its tools descriptiors.
Meta-data oriented languages, use descriptor files to describe tools. This descriptor is a file with tool meta-data usually including:

- name of program, which will be used to invoke program on command line (ex:"echo");
- parameters (name, type, syntax and others);
- outputs (name, type and others);
- tool description, author and others

These descriptors will be referenced on *pipeline*'s steps as you can see on listings 3.7 and 3.12. *NGSPipesV1*, differently from *CWL*, abstracts the origin of descriptor files. This is why you can see the reference to file (Descriptors/trimmomatic.cwl) on listing 3.12 and an id (trimmomatic) on listing 3.7. *NGSPipesV1 Engine* will use this id to obtain the descriptor from the specified repository.
Scientific workflow systems use these descriptors to validate and create the command line to invoke the required tool. If for one hand there is the additional step of having to create these descriptors on the other there are multiple advantages. First, descriptors once created can be shared among users, this will be discussed on section 3.3. Other advantage is that the *Engine* can use meta-data to check if all mandatory arguments are present or if all defined arguments are compatible with its type before invoking the tool.
Since descriptors have meta-data to describe the syntax to invoke a tool, user doesn't have to know the syntax for all used tools. Let's imagine that we want to write a *pipeline* using three tools. Each of these tools have its own syntax to define arguments:

- Command A – Name : Value (ex: `java -jar xptoA.jar name:Paul age:12`);
- Command B – Name - Value (ex: `java -jar xptoB.jar name-Paul age-12`);
- Command C – Value (ex: `java -jar xptoC.jar Paul 12`)

Not only user has to know the syntax for arguments, but also (Command C) has to know the order in which arguments have to be passed. Now imagine this for a *pipeline* with ten tools, you would spend majority of the time reading tool's API. With descriptors, the engines can build the command to be executed, facilitating users which will have a unique syntax to invoke tools and define their arguments. From the analysed SWSs, *CWL* and *NGSPipesV1* are the only meta-data oriented

languages.
In *NGSPipesV1*, descriptors are *JSON* files with an hierarquical structure. This structure is composed by a tool which is the root of the file. A tool contains commands (ex: *Velvet* tool contains the commands *velveth* and *velvetg*). A single command contains outputs and inputs (arguments).

In listing 3.3 we can observe how descriptor for *trimmomatic* tool looks like on *NGSPipesV1*.

Listing 3.3: Partial descriptior of *trimmomatic* tool on *NGSPipesV1*

```
{
  "name" : "Trimmomatic",
  ...
  "commands" :[
    {
      "name" : "trimmomatic",
      "command" : "java -jar /trimmomatic-0.33.jar",
      "arguments" : [
          {
            "name" : "mode",
            "argumentType" : "string",
            ...
          }
        ],
      "outputs" : [
        {
          "name" : "outputFile",
          ...
          "argument_name" : "outputFile"
        }
      ]
    }
  ]
}

```

Descriptors files on *CWL* are *YAML* files and have their root on *NGSPipesV1*'s command. While on *NGSPipesV1*, *velvet* descriptor is a single file which contains two commands (*velveth* and *velvetg*), on *CWL* there are two descriptor files, one for *velveth* and other for *velvetg*.

In listing 3.4 we can observe how descriptor for *trimmomatic* tool looks like on

CWL.

Listing 3.4: Partial descriptior of *trimmomatic* tool on *CWL*

```
...
class: CommandLineTool

baseCommand: [java, -jar]

arguments:
  - valueFrom: |
  ${
  var trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar";
  return trimmomaticDir;
  }

inputs:
  - id: mode
  type: string
  ...

outputs:
  - id: output
  type: File
  ...
```

CWL through *run* property(see listing 3.12) receives the path for tool's descriptor. *NGSPipesV1* abstracts this descriptor location with Repository concept. If we look to *NGSPipesV1 pipeline* root(see attachment A), we will realize that after primitive `Pipeline` there are two strings ("Github" and *Github*'s[9] uri). These two strings indicate to *NGSPipesV1 Engine* where descriptors are. This way when user defines tools and commands scopes, just needs to reference these artefacts through an id. Repository concept will be covered on section 3.3.

Both approaches have limitations. *CWL* forces that all descriptors need to be stored locally on the execution machine. *NGSPipesV1* forces that all tools used within a *pipeline* must be stored on a single repository.

In table 3.1 we can see a resume of meta-data comparison.

3.2.1.3 Methodology Summary

On table 3.1 we can see a resume of methodology of all SWS's languages.

Table 3.1: Languages Methodology comparison.

	DSL	Command Invocation
NGSPipesV1	External	Metadata
Nextflow	Internal	Raw
CWL	External	Metadata
Ruffus	Internal	Raw
Swift	External	Raw

3.2.2 Syntax

A *pipeline* is composed by a set of steps, in which a command line tool is invoked with a set of inputs. In this section we will see how a step and its inputs are declared on different SWSs. All examples presented on this chapter were extracted from the *pipelines* specifications defined to test each SWS mentioned before. You can find the complete *pipelines* on attachments.

NGSPipesV1 has a hierarchy of `tool` and `command` scopes. To define a step, with *NGSPipesV1*, user has to define two scopes(tool and command) with the syntax seen in listing 3.5.

Listing 3.5: Step syntax on *NGSPipesV1*

```
1 tool "[ToolId]" "[ExecutionContextId]"{
2   command "[CommandId]"{
3   }
4 }
```

`ToolId` and `CommandId` is how user indicates to *NGSPipesV1 Engine* that wants to run a certain command from a certain tool. As we already saw, these ids will be used to obtain the tools descriptors from repository. To define arguments there is `argument` primitive with the syntax seen in listing 3.6

Listing 3.6: Argument syntax on *NGSPipesV1*

```
1 argument "[argumentName]" "[argumentValue]"
```

If user wants to use a different command from same tool sequentially, `tool` scope can be reused (listing 3.8). On listing 3.7 we can see how *trimmomatic* command is invoked with its argument `mode` with value `SE`.

Listing 3.7: `Trimmomatic` step on *NGSPipesV1*

```
tool "Trimmomatic" "DockerConfig" {
  command "trimmomatic" {
    argument "mode" "SE"
    ...
  }
}
```

Listing 3.8: Reuse `tool` scope on *NGSPipesV1*

```
tool "Velvet" "DockerConfig" {
  command "velveth" {
    ...
  }
  command "velvetg" {
    ...
  }
}
```

To define a step on *Nextflow* there is the *process* primitive. Process primitive has the syntax seen in listing 3.9.

Listing 3.9: Step syntax on *Nextflow*

```
process "stepId" {
  """
  command to be executed
  """
}
```

Inside *process* scope, differently from *NGSPipesV1*, user defines the command to be executed. On listing 3.10 we can see how *trimmomatic* command is invoked with *Nextflow*.

Listing 3.10: `Trimmomatic` step on *Nextflow*

```
process trimmomatic {
  ...

  """
  java -jar $trimmomaticDir \
  SE
  ...
  """
}
```

On *CWL*, to define a step, user has to define a *YAML* object inside `steps` property. To define arguments, user can define step's property `in` as we can see in listing 3.11.

Listing 3.11: Step syntax on *CWL*

```
- id: "[stepId]"
  run: " [descriptorPath]"
  in:
    - id: "[argumentName]"
      valueFrom: "[argumentValue]"
```

Similarly to *NGSPipesV1* the property `run` which receives a descriptor file path, it's used to declare which command user wants to run. Listing 3.12 contains the invocation of command *trimmomatic* with *CWL*.

Listing 3.12: `Trimmomatic` step on *CWL*

```
- id: trimmomatic
  run: Descriptions/trimmomatic.cwl
  in:
    - id: mode
      valueFrom: "SE"
...
```

Ruffus, as a *Python* library, doesn't require a step to be a command invoked on command line. Steps with *Ruffus* are *Python* methods which can be annotated with *Python*'s decorators. On listing 3.13 we can see a *Ruffus* step invoking *trimmomatic*.

Listing 3.13: `Trimmomatic` step on *Ruffus*

```
def trimmomatic(input, output, illuminaclipFile):
  command = "java -jar " + trimmomaticDir + " " +\
  "SE "
  ...

  run(command)
```

Swift has the `app` primitive to declare a step. Similarly to *Nextflow*, the command to be executed on command line is defined inside `app` scope which has the syntax we can see in listing 3.14.

Listing 3.14: Step syntax on *Swift*

```
app ([outputType] [outputName]) [stepId] ([inputType] [inputName])
{
  [command to be executed]
  ;
}
```

On listing 3.15 we can observe how *trimmomatic* invocation looks like on *Swift*.

Listing 3.15: `Trimmomatic` step on *Swift*

```
app (file output) trimmomatic (file input, file illuminaclipFile)
{
  java "-jar" trimmomaticDir
  "SE"
  ...
  ;
}
```

Now lets focus on how different features are solved in each SWS.

3.2.2.1 Variables

Variables can make code cleaner, more readable and reusable. There are two types of variables:

- **Global** - variables that can be used anywhere on the *pipeline*;
- **Local** - variables that can be used inside a specific scope or context

Nextflow supports global variables with the following syntax: `[variableName] = [variableValue]`. In listing 3.16 we can see a global variable declared on *Nextflow*. This declaration example was extracted from attachment B.

Listing 3.16: *Nextflow* global variable

```
1 trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar"
```

CWL doesn't support variables directly. Through some properties like `inputBinding` and `arguments`, which supports *JavaScript*[16] expressions, we can declare local variables inside these blocks. We can see an example extracted from attachment C in listing 3.17.

Listing 3.17: *CWL* local variable

```
1 var trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar";
```

In *Ruffus*, since a *pipeline* is a *Python* file, this framework inherits global and local variables from *Python*. Both types have same syntax (`[variableName] = [variableValue]`) as shown in listing 3.18 extracted from attachment D.

Listing 3.18: *Ruffus* global and local variable

```
1 trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar";
```

Swift can also support both types of variables. By being a strongly typed language, *Swift* variables declaration also implies the variable type (`[variableType] [variableName] = [varialbleValue]`). To make a variable accessible globally, the key word `global` must be added as first term of the declaration. We can see both types of variables declared in listings 3.19 and 3.20 which were extracted from attachment E.

Listing 3.19: *Swift* local variable

```
1 string velvetDir = arg("publish_dir") + "/velvetDir";
```

Listing 3.20: *Swift* global variable

```
1 global string trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar";
```

In table 3.2 we can see a resume of variables comparison.

3.2.2.2 Arguments

A *pipeline* should be reusable, meaning that users should be able to run it multiple times with different arguments without having to change the *pipeline* itself. In order to achieve it, users must be able to define arguments outside the *pipeline*, either on a config file or through the command line when invoking the *Engine*.

On *Nextflow*'s *pipeline* definition user can access arguments through `params` object. This object can contain default values which can be overriden when invoking *Nextflow Engine*. We can see params usage and definition of argument `trimmomatic_input` in listings 3.21 and 3.22 respectivly. The whole *pipeline* with more arguments can be seen on attachment B.

Listing 3.21: Argument usage on *Nextflow*

```
1 ${params.trimmomatic_input}
```

Listing 3.22: Argument definition on *Nextflow*

```
1 --trimmomatic_input /home/dantas/Desktop/SharedFolder/_Common_/
    inputs/minimalInputs/ERR406040.fastq
```

CWL uses config files to achieve reusability. On *pipeline* definition, user can define an array called `inputs` in which declares the arguments required for that *pipeline*. These arguments can be defined on a config file which is given to *CWL Engine* at runtime. We can see in listings 3.23, 3.24 and 3.25 the declaration, usage and definition of argument `trimmomatic_input`. The whole *pipeline* with more arguments can be seen on attachment C.

Listing 3.23: Argument declaration on *CWL*

```
1 inputs:
2   - id: trimmomatic_input
3     type: File
```

Listing 3.24: Argument usage on *CWL*

```
1 in:
2 -  id: input_file
3   source: "#trimmomatic_input"
```

Listing 3.25: Argument definition on *CWL*

```
trimmomatic_input:
  class: File
  path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/
      minimalInputs/ERR406040.fastq
```

Ruffus allows to define arguments through comand line arguments when invoking the *pipeline*. For this, *Ruffus* offers a *Python* module (cmdline) which allows user to declare and parse *pipeline* arguments. In listings 3.26 and 3.27 we can observe the declaration of argument `trimmomatic_input` and its definition as comand line argument respectivly. The whole *pipeline* with more arguments can be seen on attachment D.

Listing 3.26: Argument usage on *Ruffus*

```
parser = cmdline.get_argparse()
parser.add_argument("--trimmomatic_input")
...

@files(params.trimmomatic_input, ...)
```

Listing 3.27: Argument definition on *Ruffus*

```
--trimmomatic_input /home/dantas/Desktop/SharedFolder/_Common_/inputs
    /minimalInputs/ERR406040.fastq
```

Swift also uses command line arguments to define arguments. To access arguments, user can use the method called `arg` which receives the name of the argument and returns the matching value. In listings 3.28 and 3.29 there is an example of this mechanism, extracted from attachment E.

Listing 3.28: Argument usage on *Swift*

```
arg("trimmomatic_input")
```

Listing 3.29: Argument definition on *Swift*

```
-trimmomatic_input=/home/dantas/Desktop/SharedFolder/_Common_/inputs/
    minimalInputs/ERR406040.fastq
```

In table 3.2 we can see a resume of arguments comparison.

3.2.2.3 Chain Outputs with Inputs

Pipelines are composed of multiple steps in which a command line tool is executed. As result of this execution the tool produces multiple outputs which will be used as inputs to other tools (invoked on other steps). Users can chain this outputs and inputs using files paths but since some SWSs run each step on a temporary and individual directory, it is necessary to have a mechanism to chain them. On figure 3.1 we can see a schematic in which the output `outputFile`, from `trimmomatic`, is chained with the input `filename` from `velveth`.

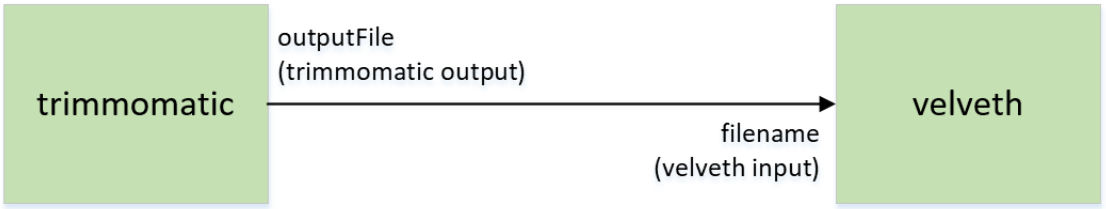

Figure 3.1: Chain `trimmomatic` output with `velveth` input schematic.

NGSPipesV1 uses `chain` directive inside a `command` scope which has the following syntax: `chain [currCommandArgumentName] [prevCommandOutputName]`. `Chain` directive passes the path of an output from the previous command as argument to the current command. There are some other variants of `chain` to chain outputs from further commands. In listing 3.30 we can observe a chain example on *NGSPipesV1* where `trimmomatic` output file, called `outputfile`, is chainned with `velveth filename` input. The whole *pipeline* with more chain examples can be seen on attachment A.

Listing 3.30: Chain inputs and outputs with *NGSPipesV1*

```
1 ...
2 command "trimmomatic" {
3   argument "outputFile" "ERR406040.filtered.fastq"
4 }
5
6 ...
7 command "velveth" {
8   chain "filename" "outputFile"
9 }
```

Nextflow uses `channel` concept to deal with chain mechanism. In each step user can define that the current step produces some outputs into a specific `channel`. This `channel` can be consumed by other steps to receive its inputs. Multiple outputs can be sent to a single channel but this `channel` can only be consumed by a single step. In listing 3.31 we can observe a chain example on *Nextflow* where `trimmomatic` output file is chainned as `velveth` input. The whole *pipeline* with more chain examples can be seen on attachment B.

Listing 3.31: Chain inputs and outputs with *Nextflow*

```
process trimmomatic {
  output:
  file params.trimmomatic_output into trimVelvChannel

  ...
}

process velveth {
  input:
  file velvetInput from trimVelvChannel

  """
  velveth \
  ...
  $velvetInput
  """
}
```

CWL lets user to define the outputs produced for each step. Every output declared needs an id, unique inside step's scope. This id in conjuntion with step's id is used when user wants to chain that output with an input of other step. The conjunction of ids from step and output has the following syntax: `#[stepId]/[outputId]`. In listing 3.32 we can observe a chain example on *CWL* where `trimmomatic` output is chainned as `velveth` input. The whole *pipeline* with more chain examples can be seen on attachment C.

Listing 3.32: Chain inputs and outputs with *CWL*

```
- id: trimmomatic
  ...
  out:
    - id: output

- id: velveth
  ...
  in:
    - id: file
      source: "#trimmomatic/output"
```

Ruffus offers *Python* decorators (`transform` and `originate`) which help users to process files and transform them originating new files. This mechanism helps to create data transformation *pipelines* but it doesn't suits chain idea. To chain outputs with inputs, users have to deal directly with files paths, meaning that users must specify a location for the output of a step and then pass that path to other step. In listing 3.33 we can observe a chain example on *Ruffus* where `trimmomatic` output is chainned as `velveth` input. The whole *pipeline* with more chain examples can be seen on attachment D.

Listing 3.33: Chain inputs and outputs with *Ruffus*

```
@files(params.trimmomatic_input, params.publish_dir + "/" + params.
    trimmomatic_output, params.trimmomatic_illuminaclip + ":2:30:10")
def trimmomatic(input, output, illuminaclipFile):
  ...

@files(params.publish_dir + "/" + params.trimmomatic_output, "
    velvetDir")
def velveth(input, output):
  ...
```

Swift allows chain mechanism with a syntax similar to what we usually see on object oriented languages like *Java* and *C#*. By declaring a variable, users can use this variable with three purposes:

- supply output location to the step which produces it;
- indentify which step produces it;
- use it as input to a consumer step

In listing 3.34 we can observe a chain example on *Swift* where `trimmomatic` output is chainned as `velveth` input. The whole *pipeline* with more chain examples can be seen on attachment E.

Listing 3.34: Chain inputs and outputs with *Swift*

```
1 ...
2 file trimOutput<single_file_mapper; file=arg("publish_dir")+"/"+arg("
     trimmomatic_output")>;
3 trimOutput = trimmomatic(trimInput, illuminaclipFile);
4
5 velvetHOutputs = velveth(trimOutput, velvetDir);
```

In table 3.2 we can see a resume of chain inputs and outputs comparison.

3.2.2.4 Steps Dependency

With steps producing multiple outputs to be consumed by different steps, a *pipeline* can easily become a giant nest. A step that depends from an output of other step must wait until the first finishes its job to run. In order to orchestrate these executions, the *Engine* must analyse step's dependencies from each other and stipulate an execution order to run each step. We can define two kinds of dependency declaration:

- **Implicit**, *Engine* can find out the order through declaration of chains between outputs and inputs;
- **Explicit**, user must declare on *pipeline* that a specific step depends from another

From the analysed SWSs, *NGSPipesV1* and *Ruffus* are the only systems that took explicit aproach, all the remainig systems use implicit declaration.
In listing 3.35 we can see how is declared that `velveh` depends from `trimmomatic` on *Ruffus pipeline*. This example was extracted from attachment D.

Listing 3.35: Dependency declaration on *Ruffus*

```
1 @follows(trimmomatic)
2 def velveth(input, output):
```

NGSPipesV1 runs steps with the same order of declaration, meaning that one step only runs after all steps declared before him have already finished. In listing 3.36 we can see how is declared that `velveh` depends from `trimmomatic` on *NGSPipesV1 pipeline*. This example was extracted from attachment A.

Listing 3.36: Dependency declaration on *NGSPipesV1*

```
tool "Trimmomatic" "DockerConfig" {
  command "trimmomatic" {
    ...
  }
}
tool "Velvet" "DockerConfig" {
  command "velveth" {
    ...
  }
}
```

In table 3.2 we can see a resume of dependency declaration comparison.

3.2.2.5 Task Parallelism

After analysing the dependency graph and define an execution order, there may be cases where multiple steps can have same order. If a step doesn't depend from another and both have their inputs available, these two steps should be able to execute in parallel. We can have two approaches to define the execution order:

- **Sequential**, steps run sequentially;
- **Parallel**, multiple steps run concurrently

Based on case study, since `trimmomatic` and `makeblatdb` don't depend from any other step, both can run concurrently. In figures 2.3 and 2.1 we can see the execution order from both approaches, parallel and sequential, respectively. On sequential execution, `makeblastdb` could run in any position, but always before `blastx`.

From the analysed SWSs, *NGSPipesV1* it's the only with sequential execution. All SWSs with parallel execution, don't require any declaration from the user since this parallel execution can be inferred from chain mechanism. In table 3.2 we can see a resume of task parallelism comparison.

3.2.2.6 Data Parallelism

Data files processed by SWSs are commonly huge (tens or even hundreds of gigabytes). To expedite the execution, these files can be split in smaller chunks and processed concurrently, taking advantage of available CPUs. In order to make this task easier, SWSs should supply some mechanism/primitives to split, process and join data files. Excluding *NGSPipesV1*, all SWSs support this idea.
To test this feature, it was made a derivation of the first case study in which the genome obtained from `velvetg` is compared against three data bases instead of a single one. To achieve this, `makeblastdb` step will be executed three times with different `allrefs` files, consequently `blastx` step will also run three times, once for each data base. In figure 2.4 we can see a diagram of this case study derivation.

In listing 3.37 we can observe how `makeblastdb` step is defined with *CWL*. If you compare this step, with the same step of variant 1 of the case study (attachment C), you will realise that is really similar. The only difference between both is the `scatter` property. Since at runtime the value of `in` and `title` will be arrays, as it can be seen on listing 3.38, `scatther` property indicates to *CWL Engine* that these arrays have to be scattered into different executions.
Since we have two inputs (`in` and `title`), to be scattered in different executions, the *Engine* has to know how to combine them either on a `one_to_one` combination or `one_to_many`. The answer for this problem relies on `scatterMethod` property which, in this case, has the value `dotproduct`, which will result in a *one to one* combination. You can see the whole *pipeline* in attachment G.

On figure 3.2 we can see how `one_to_one` and `one_to_many` strategies work.

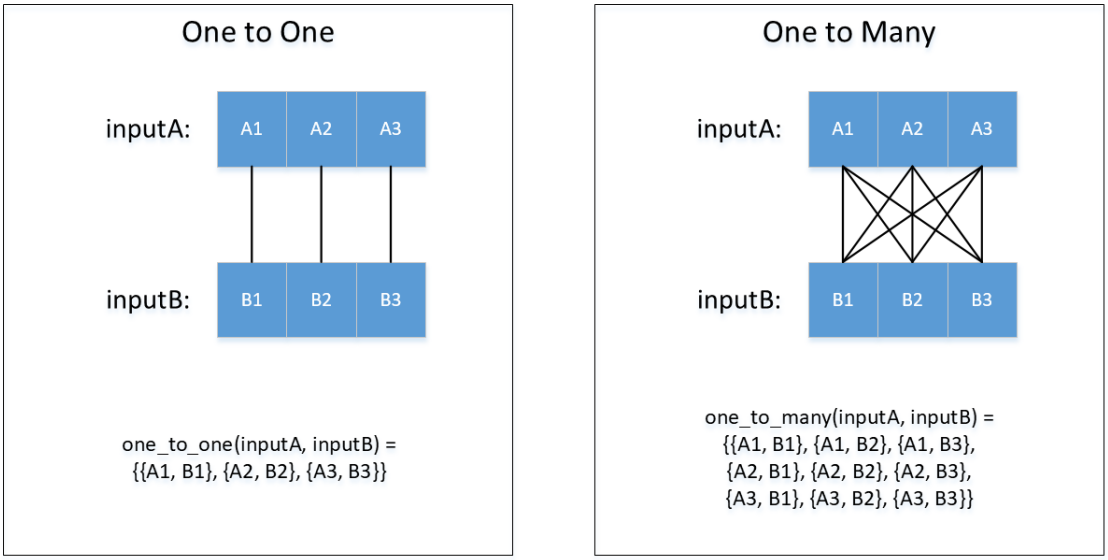

Figure 3.2: *NGSPipesV2* `strategy` primitive

Listing 3.37: Data parallelism *pipeline* on *CWL*

```
- id: makeblastdb
  run: Descriptions/makeblastdb.cwl
  scatter: [in, title]
  scatterMethod: dotproduct
  in:
  - id: in
    source: "#makeblastdb_ins"
  - id: title
    source: "#makeblastdb_titles"
  - id: dbtype
    valueFrom: "prot"
  out: [output, phr]
```

Listing 3.38: Data parallelism arguments on *CWL*

```
1 makeblastdb_ins:
2 - class: File
3 path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs
    /allrefs.fnaA.pro
4 - class: File
5 path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs
    /allrefs.fnaB.pro
6 - class: File
7 path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs
    /allrefs.fnaC.pro
8 makeblastdb_titles: [allrefsA, allrefsB, allrefsC]
```

As we already saw (section 3.2.2.3), a step in *Nextflow* can receive arguments through *Nextflow*'s `channel` concept. When there are multiple values passing through the channel, *Nextflow* runs the steps that consume this channel once for each value. These multiple executions are made in parallel.

As we can see on listing 3.39 (line 6), `makeblastdb` step, will consume through a channel the value returned by method `fromPath`. This method returns all files compliant with a certain glob pattern [11]. Once `fromPath` method will return multiple files for the value seen on listing 3.40 (line 2), `makeblastdb` step will be executed in parallel for each of those files. You can see the whole *pipeline* in attachment F.

Listing 3.39: Data parallelism *pipeline* on *Nextflow*

```
process makeblastdb {
  publishDir params.publish_dir, mode: 'copy', overwrite: true

  input:
    val title from Channel.from(params.makeblastdb_titles.split(','))
    file inFile from Channel.fromPath(params.makeblastdb_ins)

  output:
    file "${title}.*" into makeBlastBlastXChannel

  """
  makeblastdb \
  -out='${title}' \
  -dbtype=prot \
  -in='${inFile}' \
  -title='${title}'
  """
}
```

Listing 3.40: Data parallelism arguments on *Nextflow*

```
...
--makeblastdb_ins /home/dantas/Desktop/SharedFolder/_Common_/inputs/
    minimalInputs/allrefs.fna{A,B,C}.pro
--makeblastdb_titles allrefsA,allrefsB,allrefsC
--blastx_outs blastA.out,blastB.out,blastC.out
```

Ruffus achieves data parallelism through `files` decorator. This decorator receives a list of objects, each object contains the arguments for one execution of current step. This way, declaring three objects with different `allrefs` files, *Ruffus Engine* will run `makeblastdb` step three times parallely. In listing 3.41 we can observe `makeblastdb` step for this case study. You can see the whole *pipeline* in attachment H. Similarly to *Nextflow* example, we take advantage of glob patterns in order to specify the `allrefs` files. On listing 3.42 (line 2) we can see the glob pattern used for argument `makeblastdb_ins` which will be consumed by `glob` method (listing 3.41 lines 2, 3 and 4). This method has the same signature of *Nextflow*'s `fromPath`.

Listing 3.41: Data parallelism *pipeline* on *Ruffus*

```
@files([
[glob.glob(params.makeblastdb_ins)[0], None, params.publish_dir + "/"
    + params.makeblastdb_titles.split(",")[0], params.makeblastdb_
   titles.split(",")[0]],
[glob.glob(params.makeblastdb_ins)[1], None, params.publish_dir + "/"
    + params.makeblastdb_titles.split(",")[1], params.makeblastdb_
   titles.split(",")[1]],
[glob.glob(params.makeblastdb_ins)[2], None, params.publish_dir + "/"
    + params.makeblastdb_titles.split(",")[2], params.makeblastdb_
   titles.split(",")[2]]
])
def makeblastdb(input, output, outputDir, title):
  command = "makeblastdb " +\
  "-out=" + outputDir + " " +\
  "-dbtype=prot " +\
  "-in=" + input + " " +\
  "-title=" + title

  run(command)
```

Listing 3.42: Data parallelism arguments on *Ruffus*

```
...
--makeblastdb_ins /home/dantas/Desktop/SharedFolder/_Common_/inputs/
   minimalInputs/allrefs.fna\?.pro
--makeblastdb_titles allrefsA,allrefsB,allrefsC
--blastx_outs blastA.out,blastB.out,blastC.out
```

Swift language has `foreach` loops. The iterations of `foreach` loops are parallel. This way, to solve our problem we just need to iterate over an array containing the `allrefs` files as we can see in listing 3.43. You can see the whole *pipeline* in attachment I. Once again we use a glob pattern to define the argument `makeblastdb_ins` (listing 3.44 line 2). This argument is used on definition of `allrefs` array as we can see on listing 3.43 (line 14).

Listing 3.43: Data parallelism *pipeline* on *Swift*

```
app (file o[]) makeblastdb (string outDir, file allrefs, string title
    )
{
makeblastdb
"-out=" + outDir
"-dbtype=prot"
"-in=" + filename(allrefs)
"-title=" + title;
}

...

string inFilesPattern;
(inFilesLocation, inFilesPattern) = splitMakeBlastDBIn(arg("
    makeblastdb_ins"));
file allrefs[]<filesys_mapper; location=inFilesLocation, pattern=
    inFilesPattern>;

foreach inFile, idx in allrefs {
  string title = titles[idx];
  makeBlastDBOutputs[idx] = makeblastdb(arg("publish_dir")+"/"+title,
      inFile, title);
}
```

Listing 3.44: Data parallelism arguments on *Swift*

```
...
-makeblastdb_ins=/home/dantas/Desktop/SharedFolder/_Common_/inputs/
    minimalInputs/allrefs.fna?.pro
-makeblastdb_titles=allrefsA,allrefsB,allrefsC
-blastx_outs=blastA.out,blastB.out,blastC.out
```

Table 3.2 resumes data parallelism approaches discussed in this section.

3.2.2.7 Nested Pipelines

In our context a nested *pipeline* is implemented as a *pipeline* which can define steps that execute other *pipelines*. This way users can incorporate in their *pipelines*, *pipelines* developed by others expediting development process and reusing code. To support nested *pipelines* the language must allow user to deal with the *pipeline* as a whole without having to deal with its steps. On figure 3.3 we can see the

schematic for a nested *pipeline*. On this schematic, the main *pipeline* has a step, `velvet`, which will run another *pipeline* that contains two steps, `velvetg` and `velveth`.

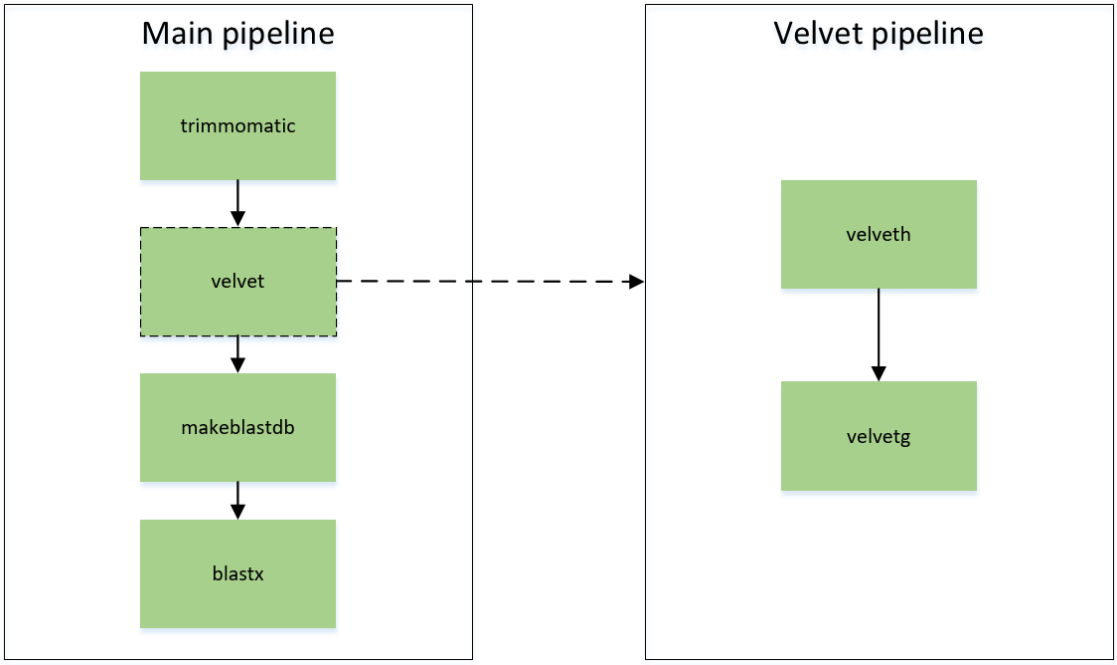

Figure 3.3: Nested *pipeline* schematic.

From all analysed systems, *CWL* is the only which supports nested *pipelines*. Although some SWSs such as *Ruffus* and *Swift* have an `import` primitive which allows to import all steps defined on another *pipeline*, this mechanism doesn't allow users to see this *pipeline* as a whole and have to deal with its steps.

A nested *pipeline* step, on *CWL*, has the same structure of a normal step which invokes a tool, but this time on the `run` primitive instead of defining the path to a tool description we supply the path to a *pipeline* definition. On listing 3.45 we can observe `velvet` step that invokes the *pipeline* (velvet.cwl) that runs `velveth` and `velvetg` steps. The complete *pipelines* can be seen on attachment N.

Listing 3.45: Invoke `velvet` *pipeline* on *CWL*

```
- id: velvet
  run: velvet.cwl
  in:
    - id: velvet_output_dir
      source: "#velvet_output_dir"
    - id: trimmomatic_output
      source: "#trimmomatic/output"
  out:
    - id: velvetgOutput
    - id: velvetgContigs
```

Listing 3.46: `Velvet` *pipeline* outputs definition on *CWL*

```
outputs:
  - id: velvetgOutput
    type: Directory
    outputSource: velvetg/output
  - id: velvetgContigs
    type: File
    outputSource: velvetg/contigs
```

Listing 3.47: Chain with `velvet` *pipeline* output on *CWL*

```
- id: blastx
  run: Descriptions/blastx.cwl
  in:
    - id: query
      source: "#velvet/velvetgContigs"
  ...
```

When dealing with chain mechanism for nested *pipelines* steps, we can face a recursion problem. In order to understand this problem, we will take in consideration the schematic seen on figure 3.4. This schematic shows an example where we have three *pipelines* which through nested *pipelines* steps, depend from each other. In order to chain, on `PipelineA`, an output from `PipelineC`, we would have to define the whole path to output, something like: `PipelineB /step inside PipelineB/ PipelineC / step inside PipelineC / output`. This scenario can get worse if we add more nesting levels.

In order to solve this problem, *CWL* allows users to define the outputs produced by a *pipeline* through `Outputs` primitive. With this approach the responsibility

is distributed by all *pipelines* which have to declare their outputs. This way we only need to specify the output with one level of nesting. On listings 3.46 and 3.47 we can see the definition of *pipelines*'s outputs and *pipeline* output chaining mechanism respectively.

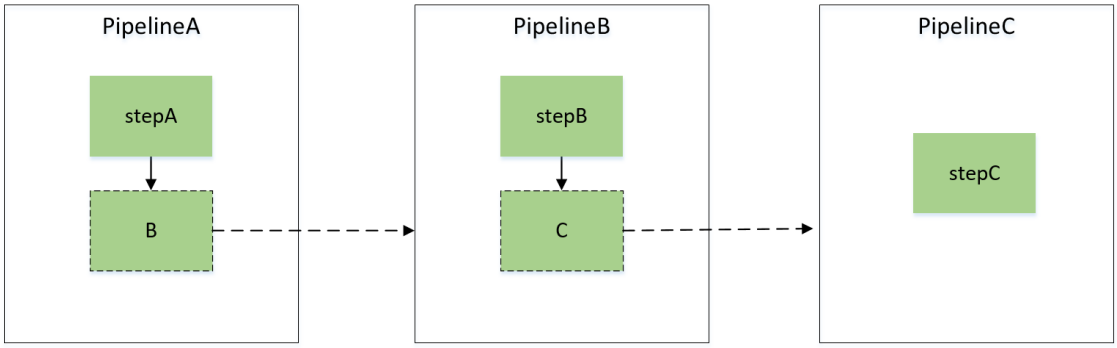

Figure 3.4: Recursive chain nested *pipeline* schematic.

On listings 3.48, 3.49 and 3.50, we can observe how we can define a step which invokes another *pipeline* with *Nextflow*, *Ruffus* and *Swift* respectively. The complete *pipelines* can be seen on attachments M, O and P. As you can conclude these steps are not more then regular steps which invoke any generic tool but this time invoking the *Engine* to run other *pipeline*.

Listing 3.48: Invoke *pipeline* on *Nextflow*

```
process velvet {
  input:
    file velvetInput from trimVelvChannel

  publishDir params.publish_dir, mode: 'copy', overwrite: true

  output:
    file "${params.velvet_output_dir}" into velvetOutputsChannel
    file "${params.velvet_output_dir}/contigs.fa" into
       velvGBlastXChannel

  """
  nextflow /home/dantas/Desktop/velvet.nf --publish_dir \$PWD --
     trimmomatic_output '${params.publish_dir}/$velvetInput' --velvet
     _output_dir ${params.velvet_output_dir}
  """
}
```

Listing 3.49: Invoke *pipeline* on *Ruffus*

```
@follows(trimmomatic)
@files(params.publish_dir + "/" + params.trimmomatic_output, params.
    velvet_output_dir)
def velvet(input, output):
  command = "python " +\
  "velvet.py " +\
  "--publish_dir " + params.publish_dir + " " +\
  "--trimmomatic_output " + input + " " +\
  "--velvet_output_dir " + output

  run(command)
```

Listing 3.50: Invoke *pipeline* on *Swift*

```
app (file velvetOutputFiles[]) velvet (file trimOutput)
{
  "/home/dantas/swift-0.96.2/bin/swift"
  "/home/dantas/Desktop/velvet.swift"
  "-publish_dir=" + arg("publish_dir")
  "-trimmomatic_output=" + filename(trimOutput)
  "-velvet_output_dir=" + arg("velvet_output_dir");
}
```

3.2.2.8 Syntax Summary

On table 3.2 we can see a resume of syntax of all SWS's languages.

3.3 Tools and Pipelines Sharing

Sharing is an important concept in any development. In science, sharing knowledge is essential for the development of the area. We can look to SWSs from the same perspective and say that all users would benefit if there was a community that shares their tools and *pipelines*.

From all studied systems, there is not a huge share community. *Nextflow* and *CWL* are the only systems in which we can find some sharing groups. These groups normally publish their tools and *pipelines* on *Git*[8] repositories such as *Github* and *Bitbucket*[6]. Due to this fact *Nextflow* supports the execution of *pipelines* stored on these repositories (*Github* and *Bitbutcket*).

With this sharing concept in mind, *NGSPipesV1* introduced repositories concept. As we already saw, the root of a *NGSPipesV1*'s *pipeline* is composed by the type and the location of a repository. This allows users to publish their tools to a supported repository (ex: *Github*) which allows other users to use it. This language support to access repositories directly it is only possible because *NGSPipesV1* repositories have a well defined structure. This structure enables the *Engine* to consult these repositories, which doesn't happen with the other analysed SWSs.

Despite these repositories there is no official platform were users can find all existent *pipelines*. This scenario is not ideal because all the artefacts are spread all over the place. It would be more advantageous to have a unique platform to share artefacts. Centralizing all available resources would make the search for tools and *pipelines* an easier task.

Table 3.2: Languages Syntax comparison.

	Variables	Arguments	Chain	Step Dependency	Task Parallelism	Data Parallelism	Nested Pipelines
NGSPipesV1	-	✗	✓	Explicit	✗	✗	✗
Nextflow	Global	✓	✓	Implicit	✓	✓	✗
CWL	Local	✓	✓	Implicit	✓	✓	✓
Ruffus	Global / Local	✓	✗	Explicit	✓	✓	✗
Swift	Global / Local	✓	✓	Implicit	✓	✓	✗

4

Solution

As seen on previous chapter, when compared with other systems, *NGSPipesV1* has some limitations on parallelism and nesting *pipelines* which are essential features for *pipeline* development. Although some limitations, *NGSPipesV1* also has some advantages such as repository and descriptors concept. While developing this solution one of our main goals was to keep the advantages of *NGSPipesV1*.

To develop this solution there were three main tasks to be developed:

- Extend *NGSPipesV1* modules in order to add support for new features;
- Develop a new language specification;
- Develop a sharing platform

Throughout next sections we will discuss each of these tasks and the solutions for the mentioned problems.

4.1 Architecture

This section presents the initial architecture of *NGSPipesV1* and the transformations made with the development of *NGSPipesV2*. Figure 4.1 shows a macro visualization of modules within the approach and how they communicate with each other.

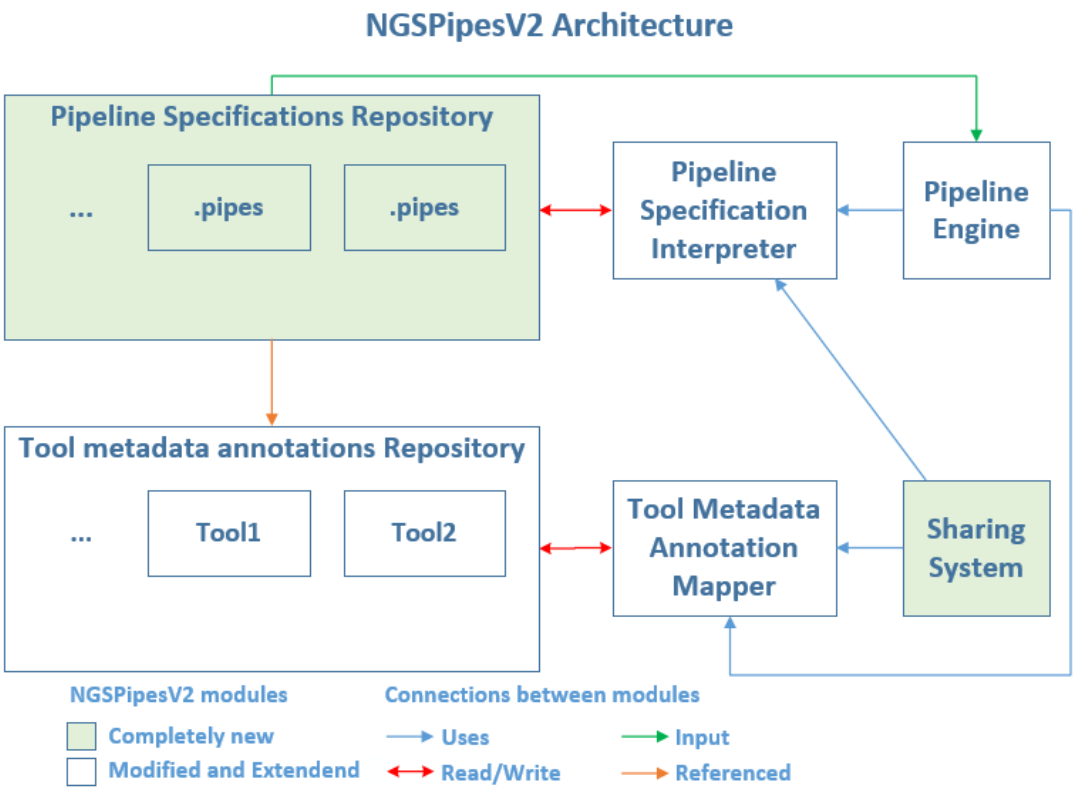

Figure 4.1: Architecture and changes from previous work

Follows a brief description:

1. *Pipeline Specifications Repository-* represents the physical repositories containing *pipelines* specification files (ex: *Github*);
2. *Pipeline Specifications Interpreter-* converts *pipeline* specifications, based on *NGSPipesV2* DSL, to *Java* objects;
3. *Pipeline Engine-* executes *pipelines*;
4. *Tool metadata annotations repository-* represents the physical repositories containing tools metadata annotation files (ex: *Github*);
5. *Tool metadata annotations mapper-* maps tools metadata annotations, based on *JSON* format, into *Java* objects;
6. *Sharing System-* supplies a WEB application which permits *NGSPipesV2*'s users to share *pipelines* and tools.

In conjunction with this project is being developed another project thesis (*Parallel execution of pipelines using bioinformatics tools* by Calmenelias Fleitas), which is responsible for the development of modules *Pipeline Engine* and *Tool Metadata Annotation Mapper*. Figure 4.2 presents all modules within *NGSPipesV2* solution, as well as the project that is responsible for each module.

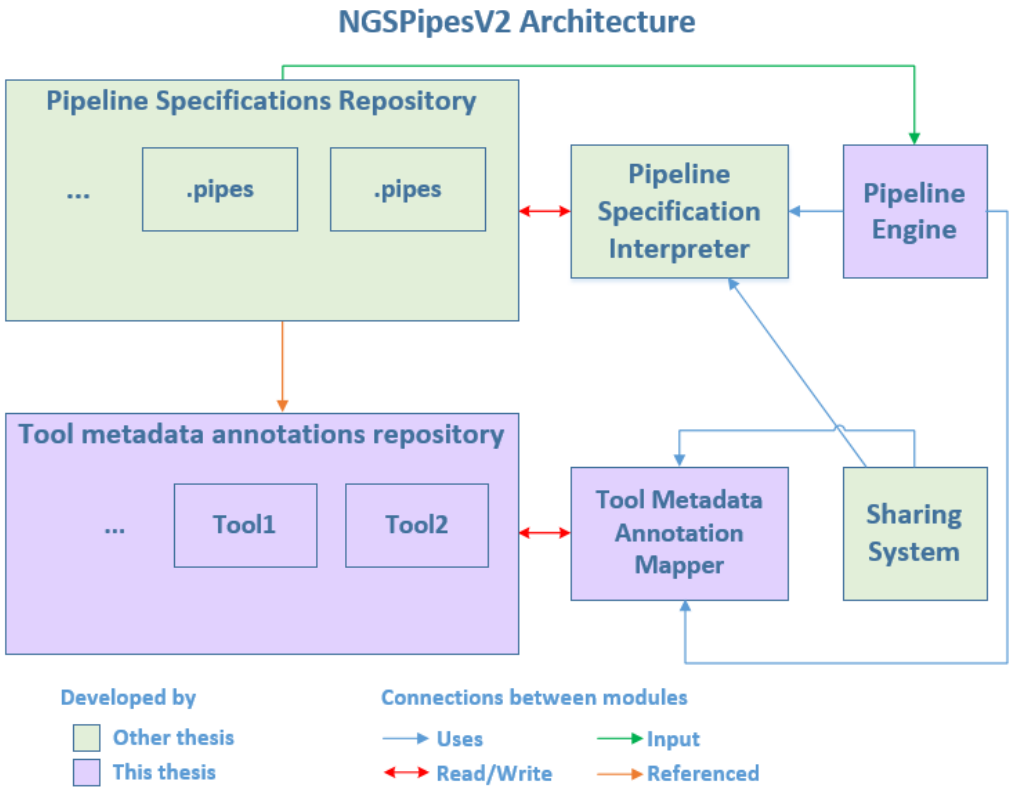

Figure 4.2: Architecture (modules distribution)

The following sections describe with more detail each part of the solution, namely the new and extended components – *pipeline sharing platform* and *pipeline specification language*.

4.2 NGSPipesV2 Language

As we saw in chapter 3, *NGSPipesV1* language has some limitations such as task parallelism, data parallelism among others. In addition to these limitations, *NGSPipesV1* has some syntax issues such as:

- *NGSPipesV1* language was developed taking in consideration that all tools used within a *pipeline* were on a single repository. Due to this assumption, the root of a *pipeline* only supported the definition of one repository. As we saw one of our goals was to enable users to use multiple repositories on a *pipeline* definition;

- As we saw on section 3.2.2.3, in other to reference a specific step, on chain mechanism, users have to indicate the index in which the step appear. This happens because the syntax didn't support the definition of an id for each step;

- Since users can reuse `tool` scope for different commands, the definition of what is a step is not intuitive

Due to these limitations, we decided to define a new language specification. This section will discuss *NGSPipesV2* and its features comparing, when possible, with *NGSPipesV1*.

4.2.1 Language Specification

NGSPipesV2 language is composed by three main primitives:

- **Repositories**;

- **Outputs**;

- **Steps**

There is a fourth optional primitive, called `Properties`. `Properties` can include some meta-data like author of *pipeline*, version of *pipeline* among others. Since this is not relevant for our scenario, let's put it aside for now.

Throughout next sections we will discuss these primitives. In order to do it we will take in consideration an `Hello World` example (listing 4.1).

Listing 4.1: Hello world example with *NGSPipesV2*

```
Repositories: [
  ToolRepository repo: {
    location: "/home/dantas/Desktop/Repository"
  }
]

Outputs: {
  result: Write[out]
}

Steps: [
  Step Write: {
    exec: repo[cmd][echo]
    inputs: {
      text: "Hello World!"
      file: "hello_world.txt"
    }
  }
]
```

`Hello World` example has a single step (`Write`) and assumes that there is a repository of tools located at `/home/dantas/Desktop/Repository`. This repository contains the `cmd` tool with `echo` command. `echo` command has two inputs:

- `text` - text to be written;
- `file` - output file

Our goal is to run the command: `echo "Hello World!" "hello_world.txt"`.

4.2.1.1 Repositories

As we already discussed, *NGSPipesV1* abstracts the source from where the tools descriptors came from with repository concept. In order to declare the repository for all tools used within a *pipeline*, on *NGSPipesV1*, the root of *pipeline* (listing 4.2) was defined with primitive `Pipeline` followed by two string. These two strings are `type` and `location` respectively.

Listing 4.2: *NGSPipesV1 pipeline* root

```
1 Pipeline "Local" "/home/dantas/Desktop/Repository" {
2 ...
3 }
```

This approach imposes that all used tools has to be on a single repository. This limitation is not practical neither coherent with a platform which wants to converge for a sharing system where users share tools descriptors and *pipelines*. Taking this in consideration, one of the main goals of *NGSPipesV2* language was to allow the usage of multiple repositories on *pipeline*.
NGSPipesV2 language has the `Repositories` scope in which user must define all used repositories, indicating for each an id and its location. Later on `Steps` scope, user will reference these repositories through their id and index them with tool's id. In listing 4.3 we can see an extract of *Antlr*[1] grammar definition for `Repositories` scope. The entire grammar definition can be seen in attachment J.

Listing 4.3: *Antlr* grammar for `Repositories` scope.

```
1  repositories: 'Repositories' ':' '[' repository (repository)* ']';
2  repository: toolRepository | pipelineRepository;
3  toolRepository: 'ToolRepository' repositoryId ':' '{'
       locationProperty configProperty? '}';
4  pipelineRepository: 'PipelineRepository' repositoryId ':' '{'
       locationProperty configProperty? '}';
5  repositoryId: ID;
6  locationProperty: 'location' ':' locationValue;
7  locationValue: STRING;
8  configProperty: 'config' ':' '{' config* '}';
9  config: configName ':' configValue;
10 configName: ID;
11 configValue: value;
```

You may have noticed that a repository (listing 4.3 line 2) can be either a tool repository (`ToolRepository`) or a *pipeline* repository (`PipelineRepository`). To implement a nested *pipeline* pattern, users need to define the location of the *pipeline* that they want to execute. To implement this feature, we took the same approach that *NGSPipesV1* has for tool's meta-data and introduced repositories of *pipelines*. With this abstraction, when defining nested *pipelines*, users only need

to reference the *pipeline* though its id (similarly to tools). This nested *pipeline* pattern will be covered on subsection 4.2.2.2.

If we look back to listing 4.1, we can now interpret it and say that, this *pipeline* uses one repository of tools (`ToolRepository`) which has the id `repo`.

4.2.1.2 Steps

Since a *pipeline* is a composition of multiple steps, this was translated directly to *NGSPipesV2*. This is reflected by `Steps` scope in which user can define all *pipelines*'s steps. The same approach can be found in *CWL*. In listing 4.4 we can see an *Antlr* grammar definition for `Steps` scope.

Listing 4.4: *Antlr* grammar for `Steps` scope.

```
steps: 'Steps' ':' '[' step (step)* ']';
step: 'Step' stepId ':' '{' execProperty executionContextProperty?
    inputsProperty? spreadProperty? '}';
stepId: ID;
execProperty: 'exec' ':' (commandReference | pipelineReference);
commandReference: repositoryId '[' toolName ']' '[' commandName ']';
toolName: ID;
commandName: ID;
pipelineReference: repositoryId '[' pipelineName ']';
pipelineName: ID;
executionContextProperty: 'execution_context' ':' value;
inputsProperty: 'inputs' ':' '{' inputProperty* '}';
inputProperty: inputName ':' inputValue;
inputName: ID;
inputValue: value | chain;
chain: stepId '[' outputName ']';
spreadProperty: 'spread' ':' '{' spreadInputsToSpreadProperty
    spreadStrategyProperty?'}';
spreadStrategyProperty: 'strategy' ':' combineStrategy;
strategyValue: combineStrategy | inputName;
combineStrategy: oneToOneStrategy | oneToManyStrategy;
oneToOneStrategy: 'one_to_one' '(' strategyValue ',' strategyValue ')
    ';
oneToManyStrategy: 'one_to_many' '(' strategyValue ',' strategyValue
    ')';
spreadInputsToSpreadProperty: 'inputs_to_spread' ':' '[' inputName ('
    ',' inputName)* ']';
```

`Steps` scope contains primitives that allow users to:

- declare which tool/*pipeline* they want to execute (`exec`);
- declare values for the inputs of tool/*pipeline* (`inputs`);
- orchestrate parallel executions (`spread`)

All these primitives will be analysed on section 4.2.2.

Let's look to `Hello World` example (listing 4.1) and interpret it. This *pipeline* has a single step called `Write`. `Write` step will execute the command `echo` from tool `cmd` which will be obtained from `repo` repository. `echo` command will be executed with the following inputs:

- `text` = `Hello world`;
- `file` = `hello_world.txt`

4.2.1.3 Outputs

`Outputs` scope appears to resolve two problems: reference an output when dealing with nested *pipelines* and the wasted time to transfer *pipeline*'s outputs when executing *pipelines* on a cloud platform.

As result of execution, the outputs produced by a *pipeline* can be huge. When executing a *pipeline* on a cloud platform there is a time overload to collect all outputs and transfer them to the local machine. To minimize this time overload, users should be able to specify which outputs should be collected after *pipeline*'s execution.

As we already discussed, when we need to chain into an input, an output coming from a step which invoked a *pipeline* (nested *pipeline*), the syntax can become quite complex. To keep the *chain* syntax simple, *NGSPipesV2* just allows users to *chain* outputs which are declared on respective *pipelines*. This way the syntax for chain mechanism is always: `[inputName]:[stepId][outputName]`, regardless of the step's type (tool or *pipeline*). As we saw on section 3.2.2.7, *CWL* has a similar approach. On *pipeline* definition, user can define through `outputs` primitive, the outputs produced by *pipeline*.

Within `Outputs` scope, user can declare the outputs produced by *pipeline*. An output is defined by an `id` (which will be used on *chain* mechanism), the `stepId` (which produces it) and `outputName` (defined on tool's descriptor). In listing 4.5 we can see an extract of *Antlr* grammar definition for `Outputs` scope. The entire grammar definition can be seen in Attachment J.

Listing 4.5: *Antlr* grammar for `Outputs` scope.

```
1 outputs: 'Outputs' ':' '{' output* '}';
2 output: outputId ':' outputValue;
3 outputId: ID;
4 outputValue: stepId '[' outputName ']';
5 outputName: ID;
```

Let's look to `Hello World` example (listing 4.1) and interpret it. This *pipeline* has a single output called `result`. `result` output is the `out` output produced by `Write` step.

4.2.2 Language Comparison

On this section we will discuss how the features analysed on chapter 3 were solved on *NGSPipesV2*.

On tables 4.1 and 4.2 we can see a resume comparing *NGSPipesV1* and *NGSPipesV2* language specification. Through next sections we will analyse each column of these tables.

Table 4.1: *NGSPipesV1* and *NGSPipesV2* language methodology comparison.

	DSL	Command Invocation
NGSPipesV1	External	Metadata
NGSPipesV2	External	Metadata

4.2.2.1 Methodology

DSL Although *NGSPipesV2* language specification is totally new and not compatible with *NGSPipesV1*, we kept the same approach and developed an external DSL. This decision was made because DSLs can be designed for a specific domain and are not limited by other languages syntax limitations.

Table 4.2: *NGSPipesV1* and *NGSPipesV2* language syntax comparison.

	Variables	Arguments	Chain	Step Dependency	Task Parallelism	Data Parallelism	Nested Pipelines
NGSPipesV1	-	✗	✓	Explicit	✗	✗	✗
NGSPipesV2	Global	✓	✓	Implicit	✓	✓	✓

Command Invocation One advantage of *NGSPipesV1* was the usage of tool descriptors to store tool's meta-data. This meta-data simplifies the definition of the *pipeline* but also enables the *Engine* to validate the *pipeline*. For this reason on *NGSPipesV2* language we decided to keep the same approach.

4.2.2.2 Syntax

On listing 4.6 we can see the definition of `trimmomatic` step on *NGSPipesV2*.

Listing 4.6: Step on *NGSPipesV2*

```
Step trimmomatic: {
  exec: repo[Trimmomatic][trimmomatic]
  execution_context: "DockerConfig"
  inputs: {
    mode: "SE"
    ...
  }
}
```

Variables On *NGSPipesV2* we decided to support global variables. To declare a variable, user can define it anywhere in the document, outside the four main scopes (`Properties`, `Repositories`, `Steps`, `Outputs`). The variable definition has the syntax: `name : value`. On listing 4.7 we can see how we declare the variable `repoLocation` to use it latter on `location` property.

Listing 4.7: Variable on *NGSPipesV2*

```
repoLocation = "https://github.com/ngspipes2/tools_support"

Repositories: [
  ToolRepository repo: {
    location: repoLocation
  }
]
```

Arguments As we saw on section 3.2.2.2, arguments, are essential when we want to develop a *pipeline* and abstract it from the concrete data. To deal with arguments, we decided to take the same approach as *Nextflow* (section 3.2.2.2). User can access arguments through object `params`. On listing 4.8 we can see the usage of argument `trimmomatic_input` to define the value of input `inputFile` of

step `trimmomatic`. The full *pipeline* with more arguments can be seen on attachment K.

Listing 4.8: Argument on *NGSPipesV2*

```
1 Step trimmomatic: {
2   ...
3   inputs: {
4     inputFile: params.trimmomatic_input
5     ...
6   }
7 }
```

Chain Outputs with Inputs On *NGSPipesV1* language the definition of inputs had two main problems. Firstly users had to use `argument` or `chain` primitives depending if the value was static or an output from another step. Secondly since there was no id to reference a certain step, user had to specify though numbers the index from which step the output, to chain, came from. To simplify the definition of inputs, on *NGSPipesV2*, users only have the syntax: `name:value`. *Chain* is a special case where `value` has the syntax: `stepId[outputName]`.

On listings 4.9 (line 3) and 4.10 (line 4), we can see how the static input (`out`) looks like on *NGSPipesV1* and *NGSPipesV2* respectively. Realize that regardless the type of input (static or chain), the syntax of input definition, on *NGSPipesV2*, is the same.

Listing 4.9: Chain on *NGSPipesV2*

```
1 tool "Blast" "DockerConfig" {
2   command "blastx" {
3     argument "-out" "blast.out"
4     ...
5   }
6 }
```

Listing 4.10: Chain on *NGSPipesV2*

```
1 Step blastx: {
2   ...
3   inputs: {
4     out: "blast.out"
5     ...
6   }
7 }
```

On listings 4.11 (line 3) and 4.12 (line 4), we can see how we chain `query` input from `blastx` step with the `contigs_fa` output from `velvetg` step, on *NGSPipesV1* and *NGSPipesV2* respectively. The full *pipelines* can be seen on attachment A and K.

Listing 4.11: Chain on *NGSPipesV2*

```
tool "Blast" "DockerConfig" {
  command "blastx" {
    chain "-query" "Velvet" "
       velvetg" "contigs_fa"
  }
}
```

Listing 4.12: Chain on *NGSPipesV2*

```
Step blastx: {
  ...
  inputs: {
    query: velvetg[contigs_fa]
    ...
  }
}
```

Steps Dependency On *NGSPipesV1* language, steps execution was sequential, meaning that steps would run in the same order of its definition. On *NGSPipesV2* language, the dependency of steps is inferred from the chain mechanism, similar to what happen on all analysed systems as we saw on section 3.2.2.4. This allows users to write steps in any order and organize the *pipeline* in the most logical way.

Task Parallelism As already mentioned, *NGSPipesV1* language had a sequential execution. Being able to execute steps in parallel is an essential feature for any SWS. On *NGSPipesV2*, the execution of a *pipeline* parallels steps when there is no dependency between them. For this users don't need any special declaration since the *Engine* infers dependencies through chain mechanism. This approach is common among all analysed SWS.

Data Parallelism To run a step in parallel with different data, user must pass an array of values to an input. If an input is declared, on tool's descriptor, as being of type `String` then the value passed must be and array of `String`. Then through `spread.inputs_to_spread` user must declare which inputs the *Engine* must spread in parallel executions. In order to define how to combine the inputs user must group them into pairs with property `spread.strategy`. The `strategy` property enables users to combine the inputs in *one to one* (`one_to_one`) or *one to many* (`one_to_many`) strategy. `spread` property is similar to `scatter` approach of *CWL*. On listing 4.13 we can see how to write data parallelism variant of case study seen on chapter 2. The full *pipeline* can be seen on attachment L. Keep in mind that `out`, `title` and `in` are declared, on *makeblastdb* descriptor, as being of type *String*, *String* and *File* respectively.

Listing 4.13: Data parallelism on *NGSPipesV2*

```
Step makeblastdb: {
  ...
  inputs: {
    out: ["allrefs", "allrefsB", "allrefsC"]
    title: ["allrefs", "allrefsB", "allrefsC"]
    in: ["E:\...\allrefs.fna.pro", "E:\...\allrefs.fnaB.pro","E:\...\
        allrefs.fnaC.pro"]
  }
  spread: {
    inputs_to_spread: [in, out, title]
    strategy: one_to_one(in, one_to_one(out, title))
  }
}
```

Nested Pipelines A nested *pipeline* step, on *NGSPipesV2*, has the same structure of a normal step which invokes a tool. The difference between a normal and a nested *pipeline* step is that the `exec` primitive instead of indexing a tool repository, indexes a *pipeline* repository. On listing 4.14 we can observe `velvet` step that invokes the `velvet` *pipeline* which runs `velveth` and `velvetg` steps. The full *pipeline* can be seen on attachment Q.

Listing 4.14: Invoke *pipeline* on *NGSPipesV2*

```
Step velvet: {
  exec: pipelines[velvet]
  inputs: {
    trimmomatic_output: trimmomatic[outputFile]
    velvet_output_dir: "velvetDir"
  }
}
```

Listing 4.15: *Pipeline* outputs definition on *NGSPipesV2*

```
Outputs: {
  output1: trimmomatic[outputFile]
  output2: blastx[out]
}
```

Listing 4.16: Chain with *pipeline* output on *NGSPipesV2*

```
1 Step blastx: {
2   ...
3   inputs: {
4     query: velvet[contigs]
5     ...
6   }
7 }
```

As we already saw, to solve the problem of chaining outputs from a nested *pipeline* step, we took the same approach of *CWL*. Users can define the outputs produced by a *pipeline* on the *pipeline*'s definition through `Outputs` primitive. On listings 4.15 and 4.16 we can see the definition of *pipelines*'s outputs and *pipeline* output chaining mechanism.

When we compare listings 4.14, 4.15 and 4.16 with the same example that has no nested *pipelines*, we will realize that the syntax is the same. This makes the syntax simpler for the user and easy to get used to.

4.3 NGSPipes Share Platform

As we saw on chapter 3, *NGSPipesV1* language didn't support neither the usage of multiple repositories nor nested *pipelines*. With *NGSPipesV2* language these problems were solved and we can build our *pipelines* reusing components implemented by others. This is helpful but it doesn't solve all problems we detected on chapter 3, since we do not offer any solution in order to share our *pipelines* and tools.
One possible solution it would be to have an official *Git* repository were users would publish their artefacts. This solution would lead us to another problem with *pipelines* and tools already existent on other repositories. This way we would force users copy their artefacts into this repository which is inconvenient and creates duplicates. To solve these problems, *NGSPipes Share Platform* was developed. *NGSPipes Share Platform* has the following main goals:

- allow creation of tools and *pipelines* repositories dynamically;
- allow publication of already existent repositories;

- allow to edit repositories content

On figure 4.3 we can see the *NGSPipes Share Platform* architecture. During next sections we will discuss each module of this architecture.

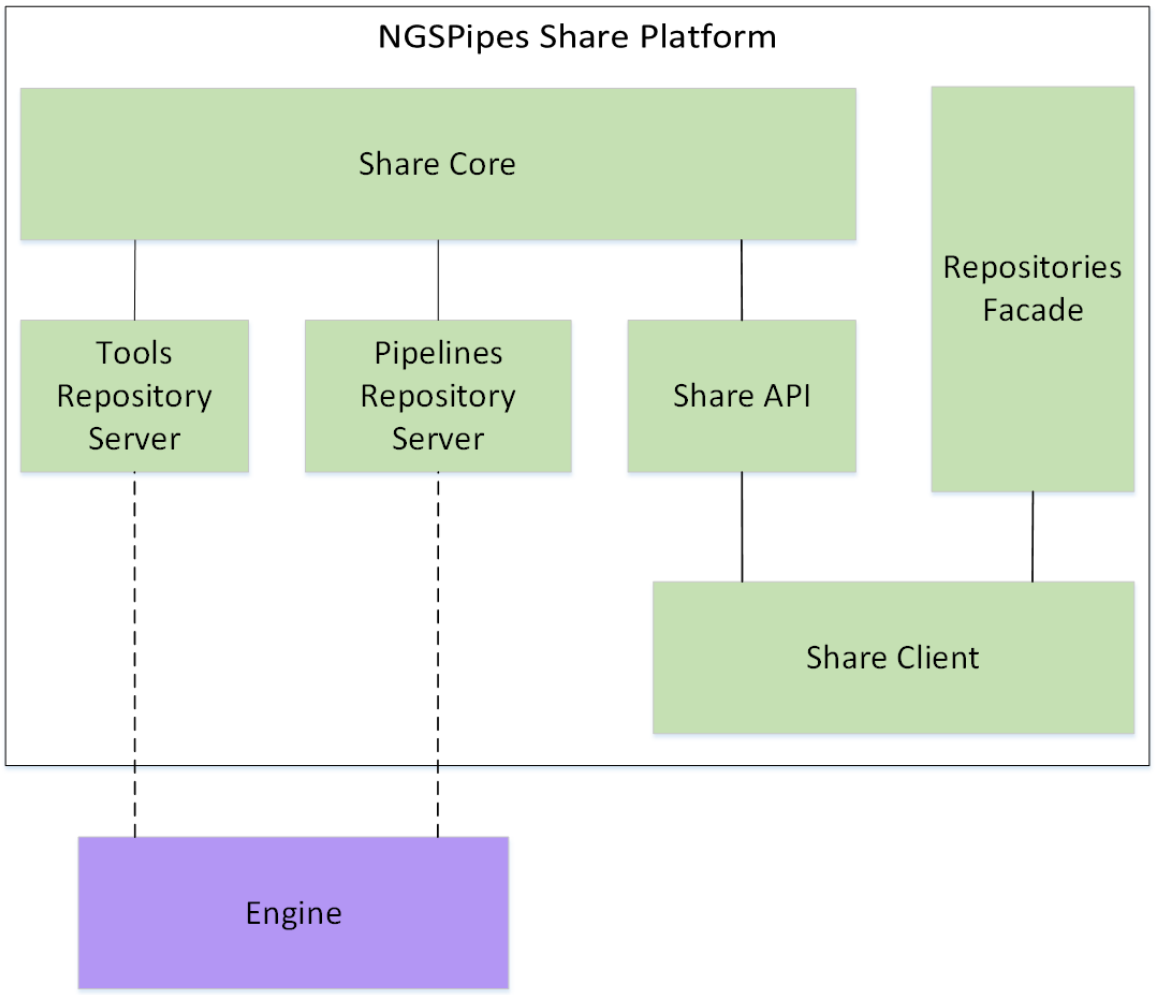

Figure 4.3: *NGSPipes Share Platform* architecture

4.3.1 Share Core

Share Core module has the entity model and services to support all *Repository Server* and *Share API* operations. On figure 4.4 we can see a UML[20] diagram of the entity model of *NGSPipes Share Platform*.

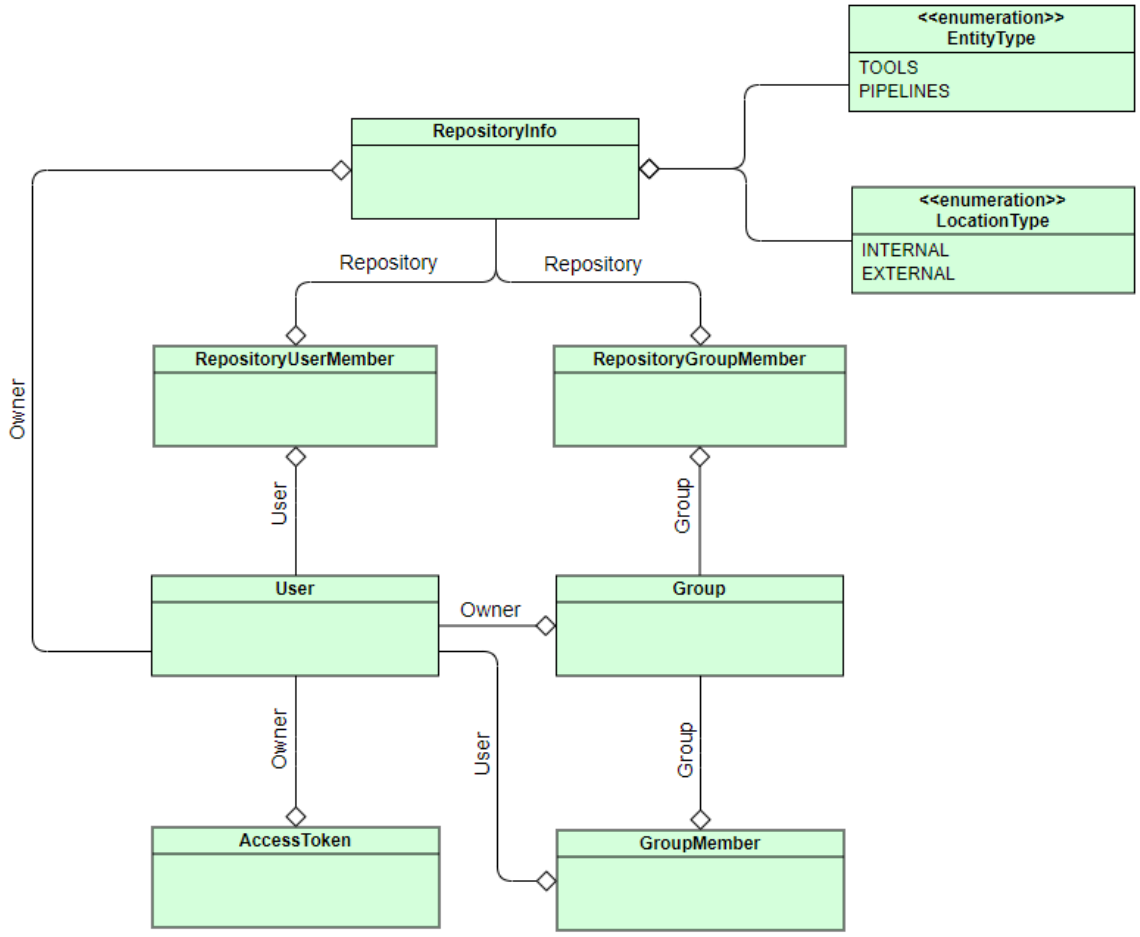

Figure 4.4: UML entity model of *NGSPipes Share Platform*

From the diagram seen on figure 4.4 we can highlight the following entities:

- `RepositoryInfo` - represents all repositories existent on the platform. These repositories can be either of tools or *pipelines*. Repositories can be internal or external. Internal repositories are repositories created by the platform. External repositories are repositories that were created outside the platform (ex: *Github*) but the owner wants to share it with the community;
- `User` - represents all users of the platform. Users can create/publish and edit repositories or they can be on the platform only to access repositories of others;
- `Group` - represents a group of users. This entity helps users to define who has access to their repositories. If a group is member of a repository, then all its users will have access to the repository

4.3.2 Tools and Pipelines Repository Servers

On *NGSPipesV2* architecture there are two modules (*ToolMetadataMapper* and *PipelineSpecificationInterpreter*) responsible for defining the interfaces of repositories to access tools and *pipelines*. On figures 4.5 and 4.6 we can observe these interfaces and the supported implementations.

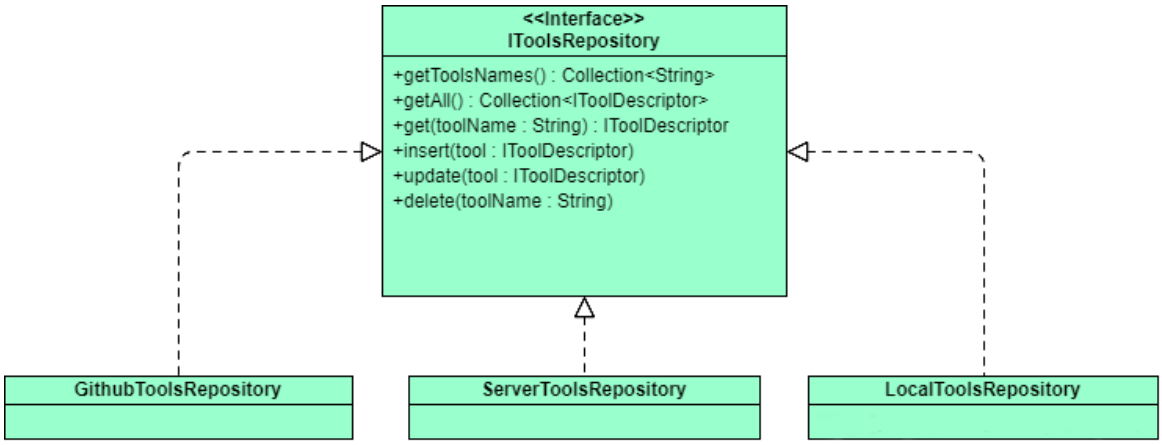

Figure 4.5: `IToolsRepository` class diagram

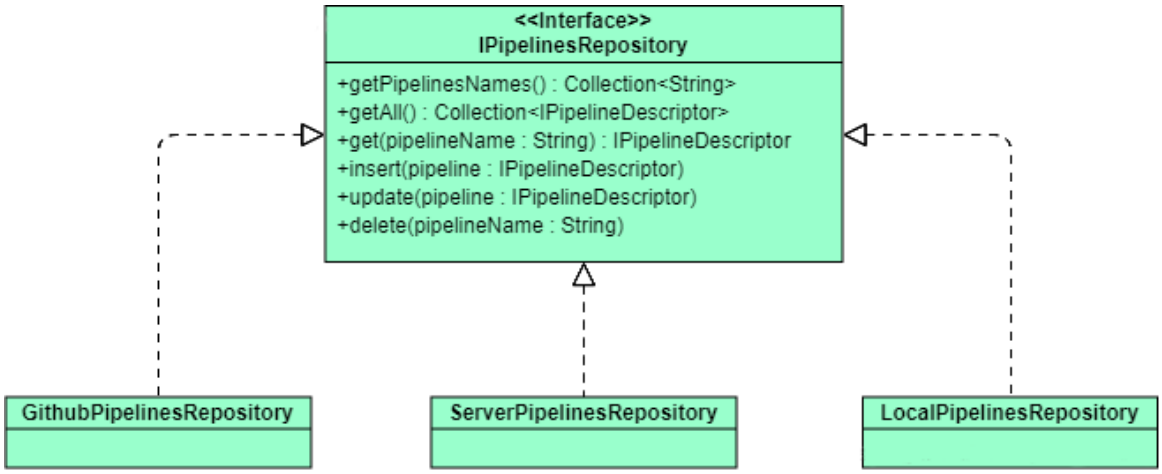

Figure 4.6: `IPipelinesRepository` class diagram

In order to allow users to publish their tools and *pipelines*, *NGSPipes Share Platform* needs to supply a solution compatible with one of the supported implementations. Since local implementations don't fit our needs to share artefacts with other

users, we have to choose between *Github* or *Server* implementations. Both implementations are based on http requests, `GithubRepository` uses the *Github* API [10] and `ServerRepository` has a set of predefined routes. The solution adopted was to implement a server compatible with `ServerToolsRepository` and `ServerPipelinesRepository`. This solution gives to the platform the ability to change where it stores tools and *pipelines*, without changing the way users interact with the server. If we had opted for *Github*, users would have to change their logic to access the artefacts, when the platform decided to stop using *Github*.

Tools Repository Server is the module responsible for implementing the interface of `IToolsRepository`. *Pipelines Repository Server* has the same responsibility of *Tools Repository Server* but dealing with *pipelines* instead of tools.
As we saw on section 4.2, users can use on their *pipelines*, repositories of tools and *pipelines* by declaring then on `Repositories` scope. Let's imagine now, the following scenario. If we had our `IToolsRepository` and `IPipelinesRepository` interfaces implemented for the same server and for some reason this server fails, all *pipelines* which depends from our repository implementations wouldn't run. In order to have a decoupled solution and to be more fault tolerant these two modules were implemented on different servers. This way if one of the systems fails, for example *Tools Repository Server*, the *pipelines* depending only from *Pipelines Repository Server* will run.

4.3.3 Share API

Share API module contains an API with all operations required by the *Share Client*. This API is built with *Spring*[19] framework. On table 4.3 we can se all controllers of *Share API* module.

Table 4.3: Controllers of *Share API* module

Controller	Entity	Description
Session Controller	-	Contains the login operation route.
AccessToken Controller	AccessToken	Contains routes for CRUD operations for `AccessToken` entity.
User Controller	User	Contains routes for CRUD operations for `User` entity.
Group Controller	Group	Contains routes for CRUD operations for `Group` entity.
GroupMember Controller	GroupMember	Contains routes for CRUD operations for `GroupMember` entity.
RepositoryInfo Controller	RepositoryInfo	Contains routes for CRUD operations for `RepositoryInfo` entity.
Repository GroupMember Controller	RepositoryGroupMember	Contains routes for CRUD operations for `RepositoryGroupMember` entity.
Repository UserMember Controller	RepositoryUserMember	Contains routes for CRUD operations for `RepositoryUserMember` entity.
Export Controller	-	Contains routes to export tools and *pipelines* to `.zip` file.
Import Controller	-	Contains routes to import tools and *pipelines* from `.zip` file.

The documentation of this API can be found at https://ngspipes-share-api.herokuapp.com/swagger-ui.html.

4.3.4 Repositories Facade

One of main goals of *Share Client* is to edit the content of tools and *pipelines* repositories. As we discussed on section 4.3.2 there are multiple implementations of repositories. Since these implementations have their own logic implemented in our *Java* library and *Share Client* is implemented with *JavaScript*, we would have to repeat all this logic in *JavaScript* in order to access all types of repositories. To solve this problem, *Repository Facade* module has a server which its only job is to implement repositories facades and call the concrete implementation of repository. Figure 4.7 shows the interfaces implemented by *Repository Facade* module. This interfaces are equal to the ones seen on figures 4.5 and 4.6, the only difference is that each method receives an object of type `DataWrapper`. This `DataWrapper` contains the data relative to which repository we want to communicate with. *Repository Facade* module uses this data object to instantiate a concrete implementation of repository and invokes the desired operation. This way when we add a new implementation of repository to our *Java* library, *Share Client* will also be compatible with it without changes on its code.

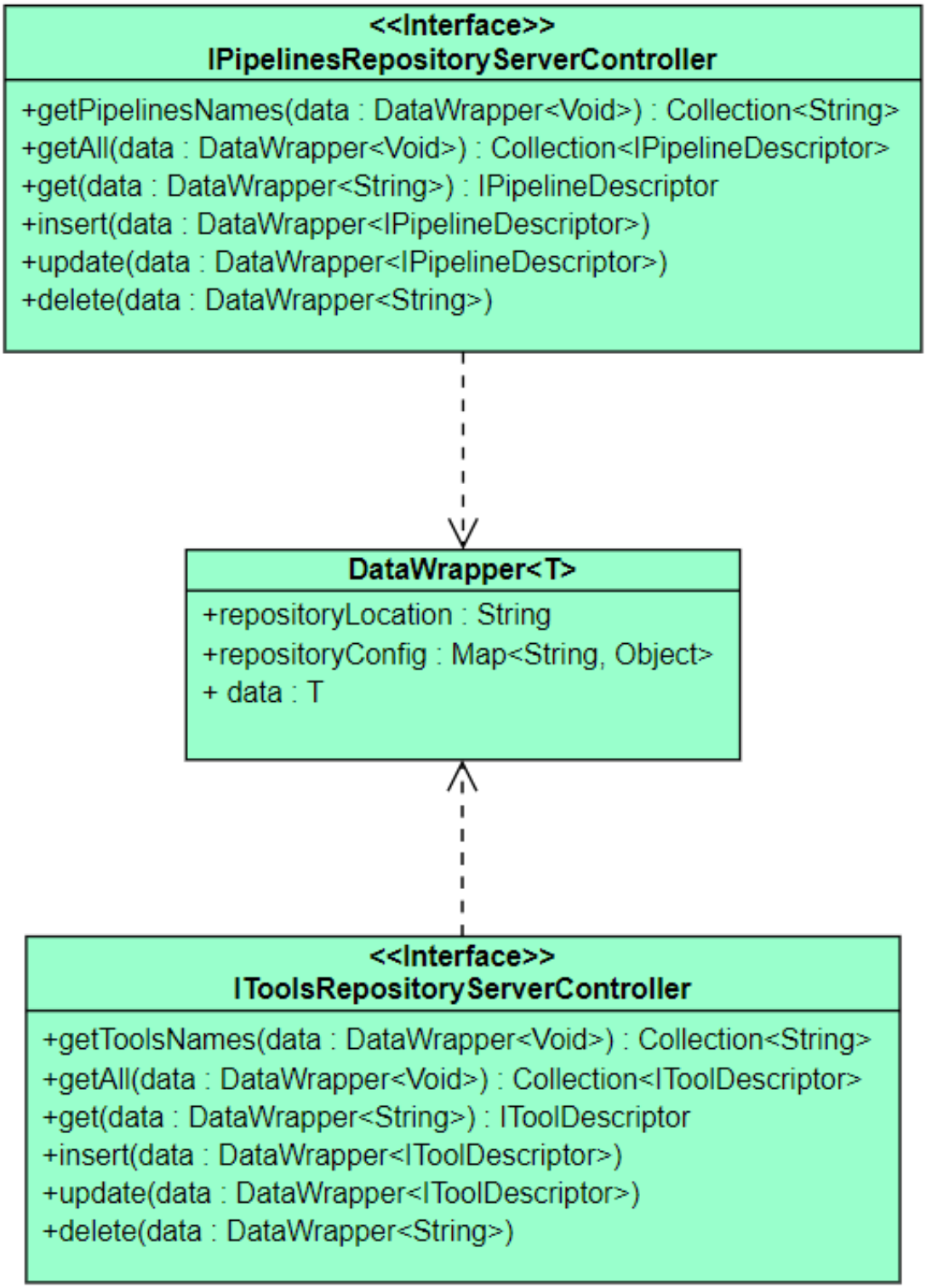

Figure 4.7: Interfaces of controllers implemented by *Repository Facade* server

4.3.5 Share Client

Share Client module is a web client developed in *Angular*[5]. This client allows users to:

- Create new *pipeline* repositories;
- Publish existent *pipeline* repositories;
- Create new tool repositories;
- Publish existent tool repositories;
- Edit content of repositories

This client is hosted on *Heroku*[13] and accessible through: `https://ngspipes-client.herokuapp.com`.

From figure 4.8 to 4.10 we can see how to publish a *pipeline* on *Share Client*. Notice that `FirstStudyCase` *pipeline* is listed on section A of figure 4.10.

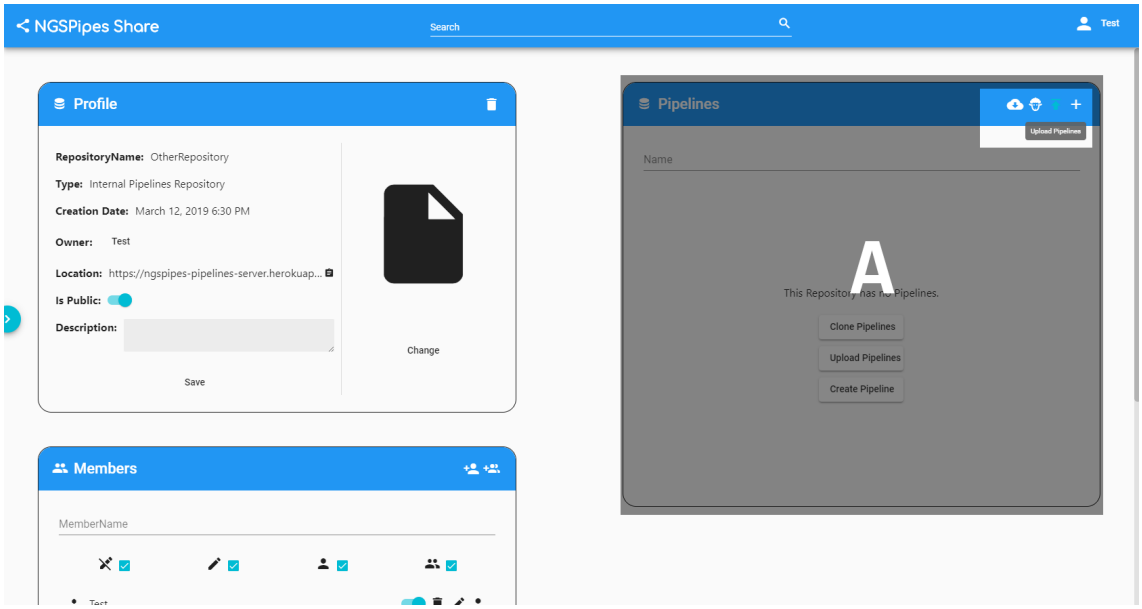

Figure 4.8: Publish *pipeline*. Click on `Publish Pipeline` button on top right corner of `Pipelines` area (section A).

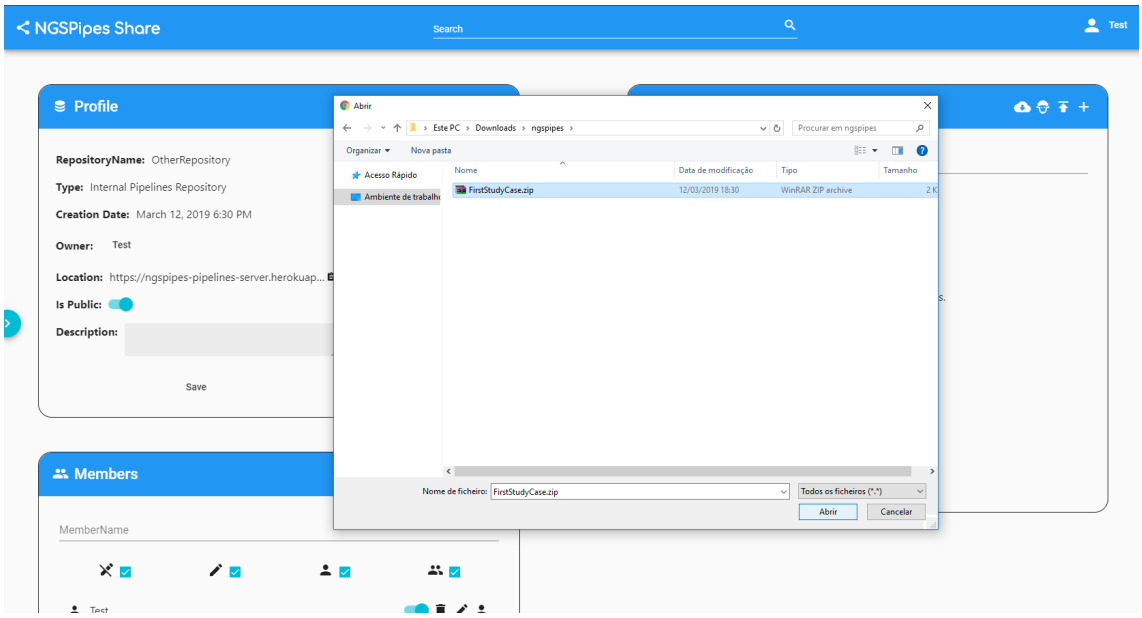

Figure 4.9: Select *pipeline* to publish. Select the `.zip` file containing the *pipeline*.

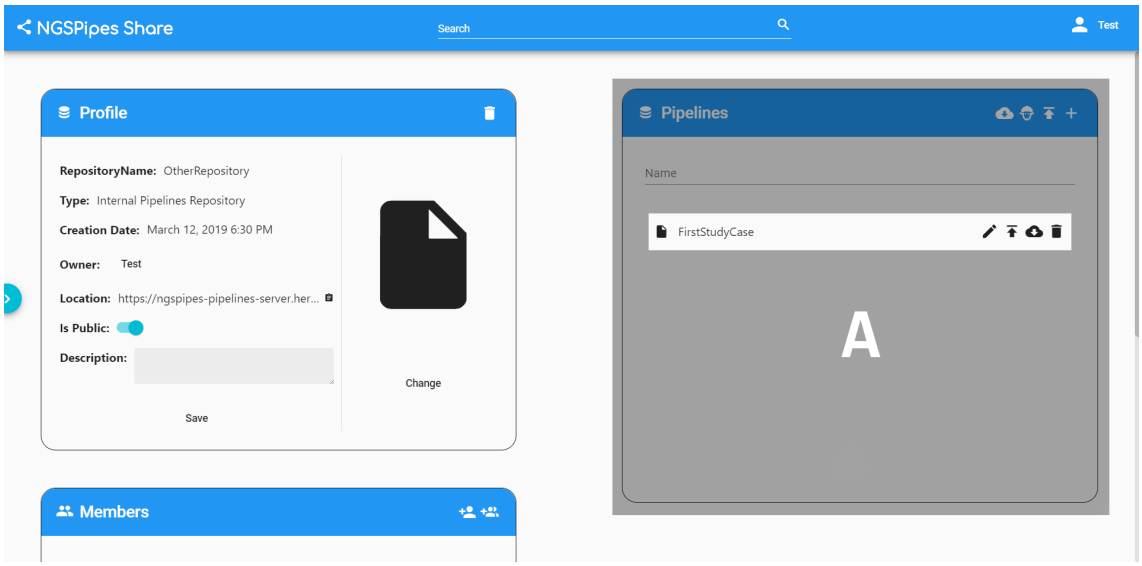

Figure 4.10: `FirstStudyCase` *pipeline* published. The published *pipeline* is listed on `Pipeline` area (section A).

From figure 4.11 to 4.13 we can see how to publish an existent repository on client. On this example we are publishing an existent repository located at `https://github.com/ngspipes2/pipelines_support`. Take a look at figure 4.11 and see that the `FirstStudyCase` *pipeline*, existent on this repository, is listed on

section A of figure 4.13.

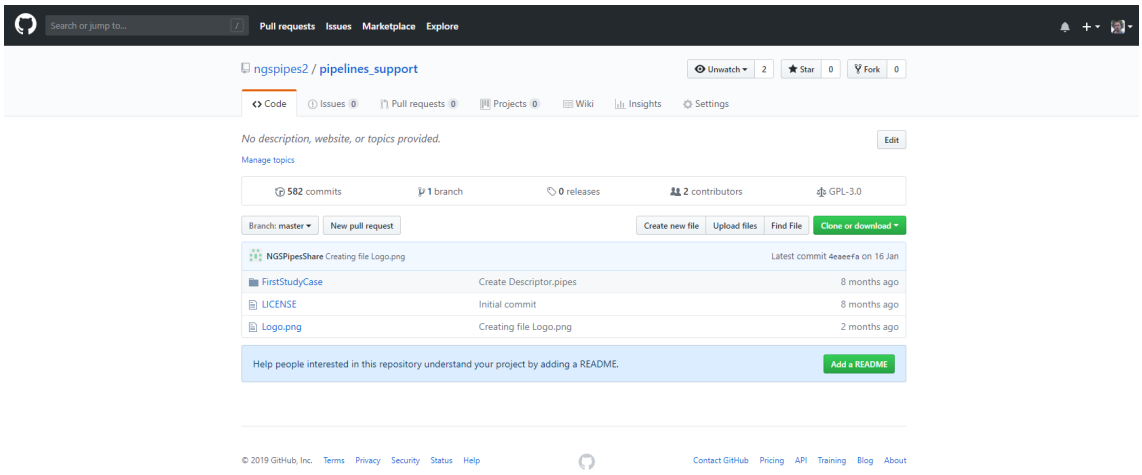

Figure 4.11: Repository to be published. This repository is a pre-existent repository, external to our *NGSPipes Share Platform*. The showed repository contains one *pipeline* called `FirstSutdyCase`.

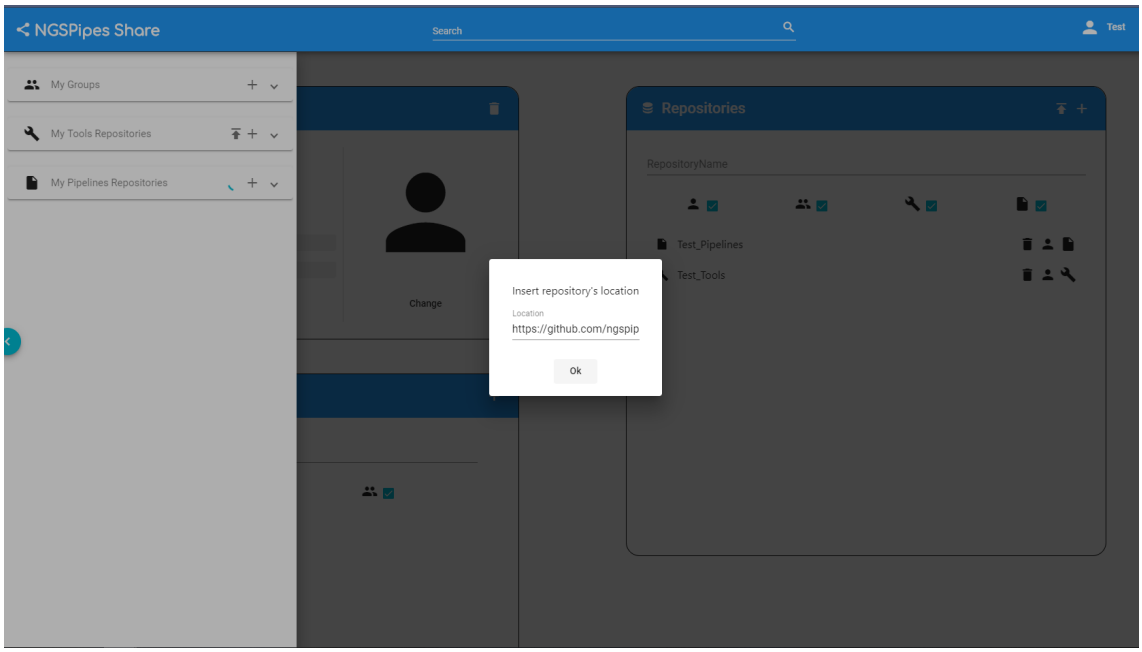

Figure 4.12: Define repository location.

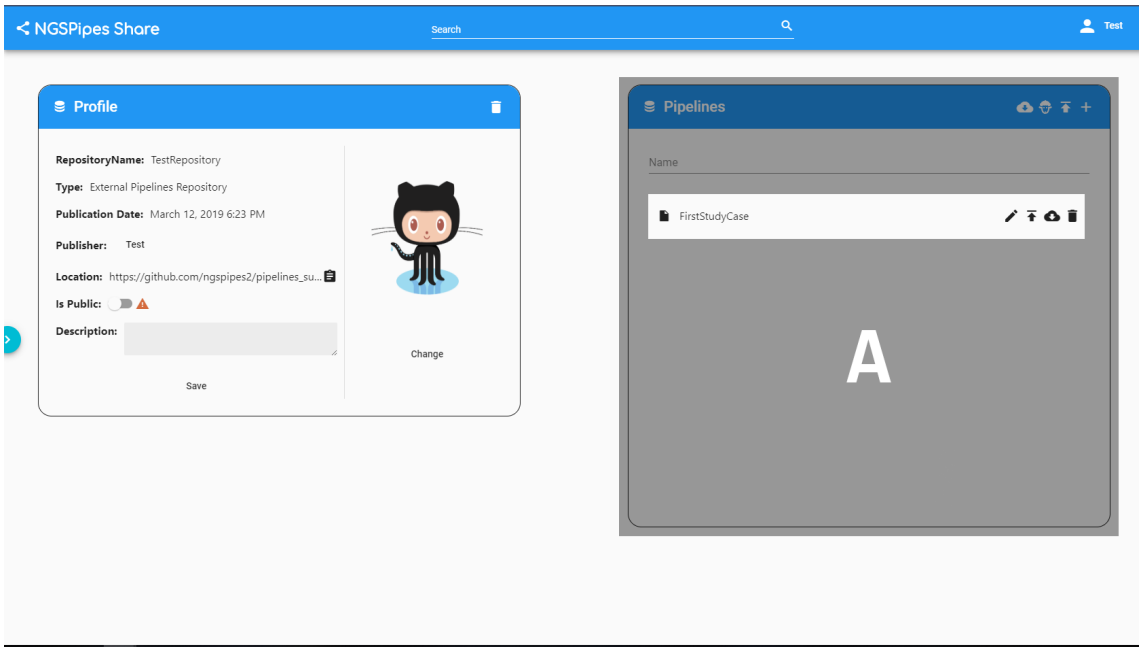

Figure 4.13: Repository content.

We can see more examples of how to use *Share Client* at `https://github.com/ngspipes2/share_share_client/wiki`.

In order to access any repository content, we need to supply the required configuration. For example, if we need to access a repository which needs credentials, like *Github*, we need to configure these credentials on *Share Client*. To navigate to *Repository Config* page click on username at top right corner of screen and select the option `Repository Config`. On section A of figure 4.14 we can see the configuration to access a repository called `MyRepository` which is an external repository hosted on *Github*. This configuration is composed of two properties (`username` and `token`). This configuration is sent to *Repository Facade* which will be used in order to communicate with the respective repository.

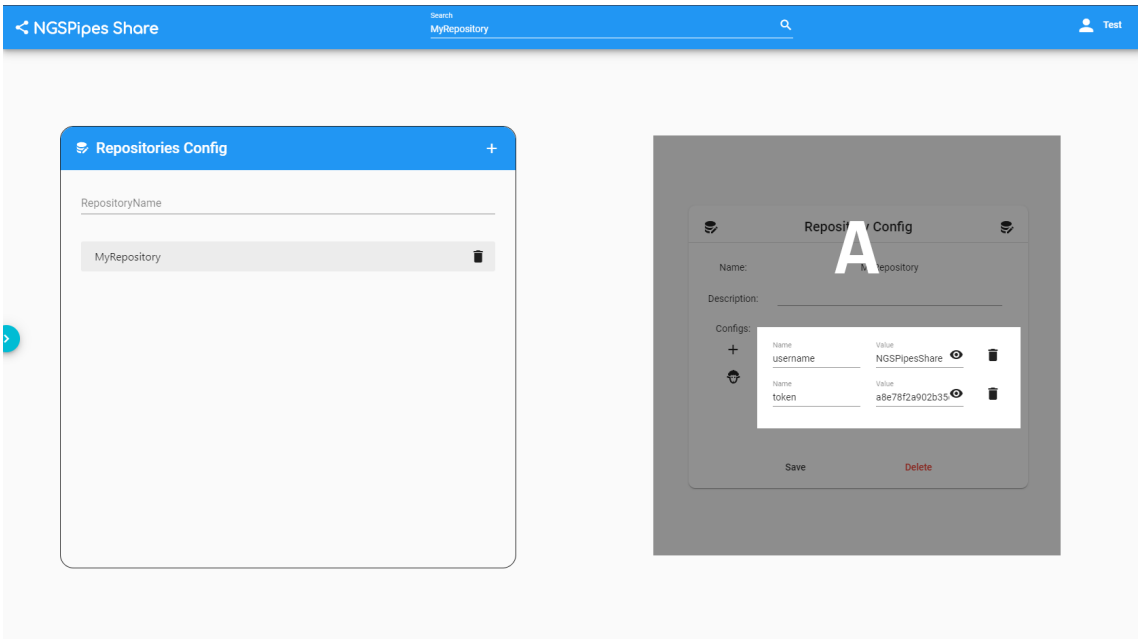

Figure 4.14: Configuration of `MyRepository` repository.

5

Conclusion

The aim of this thesis was to extend *NGSPipesV1* by proposing a language specification and a *pipeline* sharing platform to fulfil basic requirements of a SWS.

After comparing different SWSs languages against a set of features, we could define the advantages and disadvantages of *NGSPipesV1*. We also detected the lack of a sharing system in all SWSs.

With *NGSPipesV2*'s language specification we fulfil the weaknesses discussed on chapter 3 on parallelism and arguments features. While developing the *NGSPipesV2* language specification we wanted keep the advantages of *NGSPipesV1* namely: meta-data and repository concept. Firstly, having repository concept solves the requirement of not force users to have all required tools meta-data locally when running the *pipeline*. Secondly, tools meta-data permits a level of abstraction where users will be focused on what they want to do without having to know the details of tools execution.

With *NGSPipes Share Platform*, we achieved our goal of having a platform where users can share their tools and *pipelines* with the community. To develop this system, two important decisions were made:

- Develop a platform were users can create repositories dynamically;

- Develop a platform were users can publish external repositories

Creating repositories dynamically enables users to have their repositories organized with a well defined target. By allowing to publish external repositories on our platform we centralize all pieces spread on a single place.

We extended *NGSPipesV1* and added support to parallelism, argument definition and developed a share mechanism but a lot more can be done as future work. From all the possible extensions we can highlight the following:

- Tools support repository - develop a repository of utilitarian tools capable of split data files or download files from a remote location;
- Development platform - integrate *NGSPipes Share Platform* with another platform which allows users to create tools and *pipelines* with a visual representation through drag and drop;
- CWL mapper - develop a mapper to convert *CWL*'s descriptors into *NGSPipesV2*'s descriptors and vice-versa in order to add interoperability

Bibliography

[1] Antlr, JUN 2018. URL https://github.com/antlr/antlr4/blob/master/doc/index.md. (p. 50)

[2] Blast: Basic local alignment search tool, JAN 2018. URL http://blast.ncbi.nlm.nih.gov/Blast.cgi. (p. 7)

[3] usadellab - trimmomatic: A flexible read trimming tool for illumina ngs data, JAN 2018. URL http://www.usadellab.org/cms/?page=trimmomatic. (p. 7)

[4] Velvet: a sequence assembler for very short reads, JAN 2018. URL https://www.ebi.ac.uk/~zerbino/velvet/. (p. 7)

[5] Angular, JAN 2019. URL https://angular.io/. (p. 67)

[6] Bitbucket, JAN 2019. URL https://confluence.atlassian.com/bitbucket. (p. 42)

[7] C# guide, JAN 2019. URL https://docs.microsoft.com/en-us/dotnet/csharp/. (p. 15)

[8] Git, FEV 2019. URL https://git-scm.com/. (p. 42)

[9] Github guides, JAN 2019. URL https://guides.github.com/. (p. 19)

[10] Github api v3, FEV 2019. URL https://developer.github.com/v3/. (p. 63)

[11] Glob linux manual page, FEV 2019. URL http://man7.org/linux/man-pages/man7/glob.7.html. (p. 34)

[12] The apache groovy, JAN 2019. URL http://groovy-lang.org/. (p. 14)

[13] Heroku dev center, FEV 2019. URL https://devcenter.heroku.com/. (p. 67)

[14] Json, JAN 2019. URL http://json.org/json-pt.html. (p. 15)

[15] Java platform, JAN 2019. URL https://docs.oracle.com/javase/8/docs/. (p. 15)

[16] Javascript, JAN 2019. URL https://developer.mozilla.org/pt-PT/docs/Web/JavaScript. (p. 24)

[17] Java linq, JAN 2019. URL https://docs.microsoft.com/en-us/dotnet/standard/using-linq. (p. 15)

[18] Python, JAN 2019. URL https://www.python.org/. (p. 14)

[19] Spring, FEV 2019. URL https://spring.io/. (p. 63)

[20] The unified modeling language, FEV 2019. URL https://www.uml-diagrams.org/. (p. 60)

[21] The official yaml, JAN 2019. URL https://yaml.org/. (p. 16)

[22] Preimplantation genetic screening by ion torrent - pt, JUL 2019. URL https://www.thermofisher.com/pt/en/home/life-science/sequencing/dna-sequencing/preimplantation-genetic-screening.html. (p. 1)

[23] The unix standard, JAN 2019. URL https://www.opengroup.org/membership/forums/platform/unix. (p. 14)

[24] 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. (pp. ix and xi)

[25] Johan Tordsson Erik Elmroth, Francisco Hernandez. Three fundamental dimensions of scientific workflow interoperability: Model of computation, language and execution environment. *UMINF*, 2005. (p. 1)

[26] Calmenelias Pino Fleitas. Parallel execution of workflows using bioinformatics tools. December 2018. (p. 4)

[27] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010. (p. 15)

[28] Jeremy Leipzig. A review of bioinformatic pipeline frameworks. *Briefings in Bioinformatics*, 2016. (p. 13)

[29] Naohisa Goto Michael L. Heuer Peter J. A. Cock, Christopher J. Fields and Peter M. Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Oxford University Press*, 2010. (p. 7)

[30] Galaxy Project. Galaxy project @ONLINE, January 2019. URL https://galaxyproject.org/. (p. 2)

[31] CWL team. Cwl documentation @ONLINE, October 2017. URL http://www.commonwl.org/user_guide/. (p. 13)

[32] Nextflow team. Nextflow documentation @ONLINE, October 2017. URL https://www.nextflow.io/docs/latest/getstarted.html. (p. 2)

[33] NGSPipes team. Ngspipes documentation @ONLINE, October 2017. URL http://ngspipes.readthedocs.io/en/latest/. (p. 2)

[34] Ruffus team. Ruffus documentation @ONLINE, October 2017. URL http://www.ruffus.org.uk/. (p. 2)

[35] Swift team. Swift documentation @ONLINE, November 2017. URL http://swift-lang.org/docs/index.php. (p. 13)

[36] Lars H. Hansen Samuel Jacquiod Soren J. Sorensen Zhuofei Xu, Martin Asser Hansen. Bioinformatic approaches reveal metagenomic characterization of soil microbial community. *PLoS One*, 2014. (pp. 1 and 13)

Task Parallelism with *NGSPipesV1*

Listing A.1: *NGSPipesV1* task parallel *pipeline* for study case 1.

```
Pipeline "Github" "https://github.com/ngspipes/tools" {
  tool "Trimmomatic" "DockerConfig" {
    command "trimmomatic" {
      argument "mode" "SE"
      argument "quality" "-phred33"
      argument "inputFile" "ERR406040.fastq"
      argument "outputFile" "ERR406040.filtered.fastq"
      argument "fastaWithAdaptersEtc" "TruSeq3-SE.fa"
      argument "seed mismatches" "2"
      argument "palindrome clip threshold" "30"
      argument "simple clip threshold" "10"
      argument "windowSize" "4"
      argument "requiredQuality" "15"
      argument "leading quality" "3"
      argument "trailing quality" "3"
      argument "minlen length" "36"
    }
  }
  tool "Velvet" "DockerConfig" {
    command "velveth" {
      argument "output_directory" "velvetdir"
      argument "hash_length" "21"
      argument "file_format" "-fastq"
      chain "filename" "outputFile"
```

```
    }
    command "velvetg" {
      argument "output_directory" "velvetdir"
      argument "-cov_cutoff" "5"
    }
  }
  tool "Blast" "DockerConfig" {
    command "makeblastdb" {
      argument "-dbtype" "prot"
      argument "-out" "allrefs"
      argument "-title" "allrefs"
      argument "-in" "allrefs.fna.pro"
    }
    command "blastx" {
      chain "-db" "-out"
      chain "-query" "Velvet" "velvetg" "contigs_fa"
      argument "-out" "blast.out"
    }
  }
}
```

Listing A.2: Command line to invoke *NGSPipesV1* and execute task parallel *pipeline* for study case 1.

```
./bin/engine -in /home/dantas/Desktop/SharedFolder/_Common_/inputs/
   minimalInputs -out /home/dantas/Desktop/SharedFolder/NGSPipes/
   FirstExample/Outputs/minimalOutputs -pipes /home/dantas/Desktop/
   pipeline.pipes
```

B

Task Parallelism with *Nextflow*

Listing B.1: *Nextflow* task parallel *pipeline* for study case 1.

```
trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar"

process trimmomatic {
  publishDir params.publish_dir, mode: 'copy', overwrite: true

  output:
  file params.trimmomatic_output into trimVelvChannel

  """
  java -jar $trimmomaticDir \
  SE \
  -phred33 \
  '${params.trimmomatic_input}' \
  '${params.trimmomatic_output}' \
  ILLUMINACLIP:'${params.trimmomatic_illuminaclip}':2:30:10 \
  SLIDINGWINDOW:4:15 \
  LEADING:3 \
  TRAILING:3 \
  MINLEN:36
  """
}

process velveth {
  input:
```

```
25   file velvetInput from trimVelvChannel
26
27   publishDir params.publish_dir, mode: 'copy', overwrite: true
28
29   output:
30   file "velvetDir" into velvhVelvgChannel
31
32   """
33   velveth \
34   velvetDir \
35   21 \
36   -fastq \
37   $velvetInput
38   """
39 }
40
41
42 process velvetg {
43   input:
44   file velvetGInput from velvhVelvgChannel
45
46   publishDir params.publish_dir, mode: 'copy', overwrite: true
47
48   output:
49   file "$velvetGInput" into velvgOutputsChannel
50   file "$velvetGInput/contigs.fa" into velvGBlastXChannel
51
52   """
53   velvetg \
54   $velvetGInput \
55   -cov_cutoff 5
56   """
57 }
58
59 process makeblastdb {
60   publishDir params.publish_dir, mode: 'copy', overwrite: true
61
62   output:
63   file "allrefs.*" into makeBlastBlastXChannel
64
65   """
66   makeblastdb \
67   -out=allrefs \
68   -dbtype=prot \
69   -in='${params.makeblastdb_in}' \
```

```
  -title=allrefs
  """
}

process blastx {
  input:
  file blastDir from makeBlastBlastXChannel
  file query from velvGBlastXChannel

  publishDir params.publish_dir, mode: 'copy', overwrite: true

  output:
  file params.blastx_out

  """
  blastx \
  -out='${params.blastx_out}' \
  -db='${params.publish_dir}'/allrefs \
  -query=$query
  """
}
```

Listing B.2: Command line to invoke *Nextflow* and execute task parallel *pipeline* for study case 1.

```
nextflow ./pipeline.nf --publish_dir /home/dantas/Desktop/SharedFolder/
    Nextflow/FirstExample/Outputs/minimalOutputs --trimmomatic_input /
    home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs/
    ERR406040.fastq --trimmomatic_illuminaclip /home/dantas/Desktop/
    SharedFolder/_Common_/inputs/minimalInputs/TruSeq3-SE.fa --
    trimmomatic_output ERR406040.filtered.fastq --makeblastdb_in /home/
    dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs/allrefs.
    fna.pro --blastx_out blast.out
```

Task Parallelism with *CWL*

Listing C.1: *CWL* task parallel *pipeline* for study case 1.

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

cwlVersion: cwl:v1.0
class: Workflow

requirements:
  - class: StepInputExpressionRequirement
  - class: InlineJavascriptRequirement

inputs:
  - id: trimmomatic_input
    type: File
  - id: trimmomatic_illuminaclip
    type: File
  - id: trimmomatic_output
    type: string
  - id: makeblastdb_in
    type: File
  - id: blastx_out
    type: string

outputs:
  - id: trimmomaticOutput
    type: File
```

```
    outputSource: trimmomatic/output
  - id: velvetgOutput
    type: Directory
    outputSource: velvetg/output
  - id: makeBOutput
    type:
    type: array
      items: File
      outputSource: makeblastdb/output
  - id: blastOutput
    type: File
    outputSource: blastx/output

steps:
  - id: trimmomatic
    run: Descriptions/trimmomatic.cwl
    in:
      - id: mode
      valueFrom: "SE"
      - id: quality
      valueFrom: "-phred33"
      - id: input_file
      source: "#trimmomatic_input"
      - id: output_file
      source: "#trimmomatic_output"
      - id: SLIDINGWINDOW
      valueFrom: "4:15"
      - id: LEADING
      valueFrom: "3"
      - id: TRAILING
      valueFrom: "3"
      - id: MINLEN
      valueFrom: "36"
      - id: illuminaclip_file
      source: "#trimmomatic_illuminaclip"
      - id: ILLUMINACLIP
      valueFrom: ${ return inputs.illuminaclip_file.location.replace("
          file://", "") + ":2:30:10";}
    out:
      - id: output

  - id: velveth
    run: Descriptions/velveth.cwl
    in:
      - id: output_directory
```

```
    valueFrom: "velvetDir"
    - id: hash_length
    default: 21
    - id: file_format
    valueFrom: "-fastq"
    - id: file
    source: "#trimmomatic/output"
  out:
    - id: output

- id: velvetg
  run: Descriptions/velvetg.cwl
  in:
    - id: output_directory
    source: "#velveth/output"
    - id: cov_cutoff
    default: 5
  out:
    - id: output
    - id: contigs

- id: makeblastdb
  run: Descriptions/makeblastdb.cwl
  in:
    - id: in
    source: "#makeblastdb_in"
    - id: title
    valueFrom: "allrefs"
    - id: dbtype
    valueFrom: "prot"
  out: [output, phr]

- id: blastx
  run: Descriptions/blastx.cwl
  in:
    - id: out
    source: "#blastx_out"
    - id: phrFile
    source: "#makeblastdb/phr"
    - id: db
    valueFrom: ${return inputs.phrFile["location"].replace("file://",
        "").replace(".phr", "");}
    - id: query
    source: "#velvetg/contigs"
  out:
```

```
113     - id: output
```

Listing C.2: *CWL* arguments file for task parallel *pipeline* for study case 1.

```
1  trimmomatic_input:
2    class: File
3    path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs
         /ERR406040.fastq
4  trimmomatic_illuminaclip:
5    class: File
6    path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs
         /TruSeq3-SE.fa
7  trimmomatic_output: ERR406040.filtered.fastq
8
9  makeblastdb_in:
10   class: File
11   path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs
         /allrefs.fna.pro
12
13 blastx_out: blast.out
```

Listing C.3: Command line to invoke *CWL* and execute task parallel *pipeline* for study case 1.

```
1  cwl-runner --outdir ./SharedFolder/CWL/FirstExample/Outputs/
       minimalOutputs pipeline.cwl inputs.yml
```

D

Task Parallelism with *Ruffus*

Listing D.1: *Ruffus* task parallel *pipeline* for study case 1.

```
from ruffus import *
import os

parser = cmdline.get_argparse()
parser.add_argument("--publish_dir")
parser.add_argument("--trimmomatic_input")
parser.add_argument("--trimmomatic_illuminaclip")
parser.add_argument("--trimmomatic_output")
parser.add_argument("--makeblastdb_in")
parser.add_argument("--blastx_out")
params = parser.parse_args()

trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar"

def run(command):
  print("::RUNNING:" + command)
  os.system(command)

@files(params.trimmomatic_input, params.publish_dir + "/" + params.
    trimmomatic_output, params.trimmomatic_illuminaclip + ":2:30:10")
def trimmomatic(input, output, illuminaclipFile):
  command = "java -jar " + trimmomaticDir + " " +\
  "SE " +\
  "-phred33 " +\
```

```
24    input + " " +\
25    output + " " +\
26    "ILLUMINACLIP:" + illuminaclipFile + " " +\
27    "SLIDINGWINDOW:4:15 " +\
28    "LEADING:3 " +\
29    "TRAILING:3 " +\
30    "MINLEN:36"
31
32    run(command)
33
34  @follows(trimmomatic)
35  @mkdir("velvetDir")
36  @files(params.publish_dir + "/" + params.trimmomatic_output, "velvetDir
        ")
37  def velveth(input, output):
38    command = "velveth " +\
39    output + " " +\
40    "21 " +\
41    "-fastq " +\
42    input
43
44    run(command)
45
46    command = "cp -a " + output + " " + params.publish_dir
47
48    run(command)
49
50  @follows(velveth)
51  @files("velvetDir", None)
52  def velvetg(input, output):
53    command = "velvetg " +\
54    input + " " +\
55    "-cov_cutoff 5"
56
57    run(command)
58
59    command = "cp -a " + input + " " + params.publish_dir
60
61    run(command)
62
63  @files(params.makeblastdb_in, None, params.publish_dir + "/allrefs")
64  def makeblastdb(input, output, outputDir):
65    command = "makeblastdb " +\
66    "-out=" + outputDir + " " +\
67    "-dbtype=prot " +\
```

```
68   "-in=" + input + " " +\
69   "-title=allrefs"
70
71   run(command)
72
73 @follows(velvetg)
74 @follows(makeblastdb)
75 @files(None, params.publish_dir + "/" + params.blastx_out, params.
       publish_dir + "/allrefs", params.publish_dir + "/velvetDir/contigs.
       fa")
76 def blastx(input, output, db, faFile):
77   command = "blastx " +\
78   "-out=" + output + " " +\
79   "-db=" + db + " " +\
80   "-query=" + faFile
81
82   run(command)
83
84
85
86 pipeline_run([blastx])
```

Listing D.2: Command line to invoke *Ruffus* and execute task parallel *pipeline* for study case 1.

```
1 /home/dantas/swift-0.96.2/bin/swift pipeline.swift -publish_dir=/home/
      dantas/Desktop/SharedFolder/Swift/FirstExample/Outputs/
      minimalOutputs -trimmomatic_input=/home/dantas/Desktop/SharedFolder
      /_Common_/inputs/minimalInputs/ERR406040.fastq -trimmomatic_
      illuminaclip=/home/dantas/Desktop/SharedFolder/_Common_/inputs/
      minimalInputs/TruSeq3-SE.fa -trimmomatic_output=ERR406040.filtered.
      fastq -makeblastdb_in=/home/dantas/Desktop/SharedFolder/_Common_/
      inputs/minimalInputs/allrefs.fna.pro -blastx_out=blast.out
```

Task Parallelism with *Swift*

Listing E.1: *Swift* task parallel *pipeline* for study case 1.

```
type file;


global string trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar";


app (file output) trimmomatic (file input, file illuminaclipFile)
{
  java "-jar" trimmomaticDir
  "SE"
  "-phred33"
  filename(input)
  filename(output)
  "ILLUMINACLIP:" + filename(illuminaclipFile) + ":2:30:10"
  "SLIDINGWINDOW:4:15"
  "LEADING:3"
  "TRAILING:3"
  "MINLEN:36";
}

app (file velvetHOutputFiles[]) velveth (file trimOutput, string
    velvetDir)
{
  velveth
```

```
  "__root__" + velvetDir
  "21"
  "-fastq"
  filename(trimOutput);
}

app (file velvetGOutputFiles[]) velvetg (file velvetHOutputFiles[],
    string velvetDir)
{
  velvetg
  "__root__" + velvetDir
  "-cov_cutoff"
  "5";
}

app (file o[]) makeblastdb (string outDir, file allrefs, string title)
{
  makeblastdb
  "-out=" + outDir
  "-dbtype=prot"
  "-in=" + filename(allrefs)
  "-title=" + title;
}

app blastx (file makeBlastDBOutputs[], file velvetGOutputs[], string
    out, string db, file query)
{
  blastx
  "-out=" + out
  "-db=" + db
  "-query=" + filename(query);
}


file illuminaclipFile<single_file_mapper; file=arg("trimmomatic_
    illuminaclip")>;
file trimInput<single_file_mapper; file=arg("trimmomatic_input")>;
file trimOutput<single_file_mapper; file=arg("publish_dir")+"/"+arg("
    trimmomatic_output")>;
trimOutput = trimmomatic(trimInput, illuminaclipFile);

string velvetDir = arg("publish_dir") + "/velvetDir";

string velvetHFiles[] = [velvetDir+"/Log", velvetDir+"/Roadmaps",
    velvetDir+"/Sequences"];
```

```
64 file velvetHOutputs[] <array_mapper; files=velvetHFiles>;
65 velvetHOutputs = velveth(trimOutput, velvetDir);
66
67 string velvetGFiles[] = [velvetDir+"/contigs.fa", velvetDir+"/Graph",
       velvetDir+"/LastGraph", velvetDir+"/PreGraph", velvetDir+"/stats.
       txt"];
68 file velvetGOutputs[] <array_mapper; files=velvetGFiles>;
69 velvetGOutputs = velvetg(velvetHOutputs, velvetDir);
70
71 file allrefs<single_file_mapper; file=arg("makeblastdb_in")>;
72 file makeBlastDBOutputs[] <filesys_mapper; location=arg("publish_dir"),
        pattern="allrefs*">;
73 makeBlastDBOutputs = makeblastdb(arg("publish_dir")+"/allrefs", allrefs
       , "allrefs");
74
75 string blastOut = arg("publish_dir") + "/" + arg("blastx_out");
76 string blastDB = arg("publish_dir") + "/allrefs";
77 blastx(makeBlastDBOutputs, velvetGOutputs, blastOut, blastDB,
       velvetGOutputs[0]);
```

Listing E.2: Command line to invoke *Swift* and execute task parallel *pipeline* for study case 1.

```
1 /home/dantas/swift-0.96.2/bin/swift pipeline.swift -publish_dir=/home/
      dantas/Desktop/SharedFolder/Swift/FirstExample/Outputs/
      minimalOutputs -trimmomatic_input=/home/dantas/Desktop/SharedFolder
      /_Common_/inputs/minimalInputs/ERR406040.fastq -trimmomatic_
      illuminaclip=/home/dantas/Desktop/SharedFolder/_Common_/inputs/
      minimalInputs/TruSeq3-SE.fa -trimmomatic_output=ERR406040.filtered.
      fastq -makeblastdb_in=/home/dantas/Desktop/SharedFolder/_Common_/
      inputs/minimalInputs/allrefs.fna.pro -blastx_out=blast.out
```

Data Parallelism with *Nextflow*

Listing F.1: *Nextflow* data parallel *pipeline* for study case 1.

```
trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar"

process trimmomatic {
  publishDir params.publish_dir, mode: 'copy', overwrite: true

  output:
    file params.trimmomatic_output into trimVelvChannel

  """
  java -jar $trimmomaticDir \
  SE \
  -phred33 \
  '${params.trimmomatic_input}' \
  '${params.trimmomatic_output}' \
  ILLUMINACLIP:'${params.trimmomatic_illuminaclip}':2:30:10 \
  SLIDINGWINDOW:4:15 \
  LEADING:3 \
  TRAILING:3 \
  MINLEN:36
  """
}

process velveth {
  input:
```

```
25    file velvetInput from trimVelvChannel
26
27  publishDir params.publish_dir, mode: 'copy', overwrite: true
28
29  output:
30    file "${params.velvet_output_dir}" into velvhVelvgChannel
31
32  """
33  velveth \
34  '${params.velvet_output_dir}' \
35  21 \
36  -fastq \
37  $velvetInput
38  """
39 }
40
41 process velvetg {
42  input:
43    file velvetGInput from velvhVelvgChannel
44
45  publishDir params.publish_dir, mode: 'copy', overwrite: true
46
47  output:
48    file "$velvetGInput" into velvgOutputsChannel
49    file "$velvetGInput/contigs.fa" into velvGBlastXChannel
50
51  """
52  velvetg \
53  $velvetGInput \
54  -cov_cutoff 5
55  """
56 }
57
58 process makeblastdb {
59  publishDir params.publish_dir, mode: 'copy', overwrite: true
60
61  input:
62    val title from Channel.from(params.makeblastdb_titles.split(','))
63    file inFile from Channel.fromPath(params.makeblastdb_ins)
64
65  output:
66    file "${title}.*" into makeBlastBlastXChannel
67
68  """
69  makeblastdb \
```

```
  -out='${title}' \
  -dbtype=prot \
  -in='${inFile}' \
  -title='${title}'
  """
}

process blastx {
  input:
    val out from Channel.from(params.blastx_outs.split(','))
    val title from Channel.from(params.makeblastdb_titles.split(','))
    file blastDir from makeBlastBlastXChannel
    file query from velvGBlastXChannel

  publishDir params.publish_dir, mode: 'copy', overwrite: true

  output:
    file "${out}"

  """
  blastx \
  -out='${out}' \
  -db='${params.publish_dir}/${title}' \
  -query=$query
  """
}
```

Listing F.2: Command line to invoke *Nextflow* and execute data parallel *pipeline* for study case 1.

```
nextflow ./pipeline.nf --publish_dir /home/dantas/Desktop/SharedFolder/
    Nextflow/DataParallelism/Outputs/minimalOutputs --trimmomatic_input
     /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs/
    ERR406040.fastq --trimmomatic_illuminaclip /home/dantas/Desktop/
    SharedFolder/_Common_/inputs/minimalInputs/TruSeq3-SE.fa --
    trimmomatic_output ERR406040.filtered.fastq --velvet_output_dir
    velvetDir --makeblastdb_ins /home/dantas/Desktop/SharedFolder/_
    Common_/inputs/minimalInputs/allrefs.fna{A,B,C}.pro --makeblastdb_
    titles allrefsA,allrefsB,allrefsC --blastx_outs blastA.out,blastB.
    out,blastC.out 2>&1 | tee ./execution.out
```

Data Parallelism with *CWL*

Listing G.1: *CWL* data parallel *pipeline* for study case 1.

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

cwlVersion: cwl:v1.0
class: Workflow

requirements:
  - class: StepInputExpressionRequirement
  - class: InlineJavascriptRequirement
  - class: ScatterFeatureRequirement

inputs:
  -  id: trimmomatic_input
    type: File
  -  id: trimmomatic_illuminaclip
    type: File
  -  id: trimmomatic_output
    type: string
  -  id: velvet_output_dir
    type: string
  -  id: makeblastdb_ins
    type: File[]
  -  id: makeblastdb_titles
    type: string[]
  -  id: blastx_outs
```

```
25    type: string[]
26
27 outputs:
28   -  id: trimmomaticOutput
29     type: File
30     outputSource: trimmomatic/output
31   -  id: velvetgOutput
32     type: Directory
33     outputSource: velvetg/output
34   -  id: makeBOutput
35     type:
36       type: array
37       items:
38       type: array
39       items: File
40     outputSource: makeblastdb/output
41   -  id: blastOutput
42     type: File[]
43     outputSource: blastx/output
44
45 steps:
46   -  id: trimmomatic
47     run: Descriptions/trimmomatic.cwl
48     in:
49       -  id: mode
50         valueFrom: "SE"
51       -  id: quality
52         valueFrom: "-phred33"
53       -  id: input_file
54         source: "#trimmomatic_input"
55       -  id: output_file
56         source: "#trimmomatic_output"
57       -  id: SLIDINGWINDOW
58         valueFrom: "4:15"
59       -  id: LEADING
60         valueFrom: "3"
61       -  id: TRAILING
62         valueFrom: "3"
63       -  id: MINLEN
64         valueFrom: "36"
65       -  id: illuminaclip_file
66         source: "#trimmomatic_illuminaclip"
67       -  id: ILLUMINACLIP
68         valueFrom: ${ return inputs.illuminaclip_file.location.replace("
             file://", "") + ":2:30:10";}
```

```
    out:
      - id: output

  - id: velveth
    run: Descriptions/velveth.cwl
    in:
      - id: output_directory
        source: "#velvet_output_dir"
      - id: hash_length
        default: 21
      - id: file_format
        valueFrom: "-fastq"
      - id: file
        source: "#trimmomatic/output"
    out:
      - id: output

  - id: velvetg
    run: Descriptions/velvetg.cwl
    in:
      - id: output_directory
        source: "#velveth/output"
      - id: cov_cutoff
        default: 5
    out:
      - id: output
      - id: contigs

  - id: makeblastdb
    run: Descriptions/makeblastdb.cwl
    scatter: [in, title]
    scatterMethod: dotproduct
    in:
      - id: in
        source: "#makeblastdb_ins"
      - id: title
        source: "#makeblastdb_titles"
      - id: dbtype
        valueFrom: "prot"
    out: [output, phr]

  - id: blastx
    run: Descriptions/blastx.cwl
    scatter: [out, phrFile]
    scatterMethod: dotproduct
```

```
114    in:
115      -  id: out
116       source: "#blastx_outs"
117      -  id: phrFile
118       source: "#makeblastdb/phr"
119      -  id: db
120       valueFrom: ${return inputs.phrFile["location"].replace("file://",
              "").replace(".phr", "");}
121      -  id: query
122       source: "#velvetg/contigs"
123    out:
124      -  id: output
```

Listing G.2: *CWL* arguments file for data parallel *pipeline* for study case 1.

```
1  trimmomatic_input:
2    class: File
3    path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs
         /ERR406040.fastq
4  trimmomatic_illuminaclip:
5    class: File
6    path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs
         /TruSeq3-SE.fa
7  trimmomatic_output: ERR406040.filtered.fastq
8
9  velvet_output_dir: velvetDir
10
11 makeblastdb_ins:
12   - class: File
13   path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs
         /allrefs.fnaA.pro
14   - class: File
15   path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs
         /allrefs.fnaB.pro
16   - class: File
17   path: /home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs
         /allrefs.fnaC.pro
18   makeblastdb_titles: [allrefsA, allrefsB, allrefsC]
19
20 blastx_outs: [blastA.out, blastB.out, blastC.out]
```

Listing G.3: Command line to invoke *CWL* and execute data parallel *pipeline* for

study case 1.

```
cwl-runner --outdir ./SharedFolder/CWL/DataParallelism/Outputs/
    minimalOutputs pipeline.cwl inputs.yml 2>&1 | tee ./execution.out
```

Data Parallelism with *Ruffus*

Listing H.1: *Ruffus* data parallel *pipeline* for study case 1.

```
from ruffus import *
import os
import multiprocessing
import glob

parser = cmdline.get_argparse()
parser.add_argument("--publish_dir")
parser.add_argument("--trimmomatic_input")
parser.add_argument("--trimmomatic_illuminaclip")
parser.add_argument("--trimmomatic_output")
parser.add_argument("--velvet_output_dir")
parser.add_argument("--makeblastdb_ins")
parser.add_argument("--makeblastdb_titles")
parser.add_argument("--blastx_outs")
params = parser.parse_args()

trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar"

def run(command):
  print("::RUNNING:" + command)
  os.system(command)

@files(params.trimmomatic_input, params.publish_dir + "/" + params.
    trimmomatic_output, params.trimmomatic_illuminaclip + ":2:30:10")
```

```
24 def trimmomatic(input, output, illuminaclipFile):
25   command = "java -jar " + trimmomaticDir + " " +\
26       "SE " +\
27       "-phred33 " +\
28       input + " " +\
29       output + " " +\
30       "ILLUMINACLIP:" + illuminaclipFile + " " +\
31       "SLIDINGWINDOW:4:15 " +\
32       "LEADING:3 " +\
33       "TRAILING:3 " +\
34       "MINLEN:36"
35
36   run(command)
37
38 @follows(trimmomatic)
39 @files(params.publish_dir + "/" + params.trimmomatic_output, params.
     publish_dir + "/" + params.velvet_output_dir)
40 def velveth(input, output):
41   command = "velveth " +\
42       output + " " +\
43       "21 " +\
44       "-fastq " +\
45       input
46
47   run(command)
48
49 @follows(velveth)
50 @files(params.publish_dir + "/" + params.velvet_output_dir, None)
51 def velvetg(input, output):
52   command = "velvetg " +\
53       input + " " +\
54       "-cov_cutoff 5"
55
56   run(command)
57
58 @files([
59 [glob.glob(params.makeblastdb_ins)[0], None, params.publish_dir + "/" +
     params.makeblastdb_titles.split(",")[0], params.makeblastdb_titles
    .split(",")[0]],
60 [glob.glob(params.makeblastdb_ins)[1], None, params.publish_dir + "/" +
     params.makeblastdb_titles.split(",")[1], params.makeblastdb_titles
    .split(",")[1]],
61 [glob.glob(params.makeblastdb_ins)[2], None, params.publish_dir + "/" +
     params.makeblastdb_titles.split(",")[2], params.makeblastdb_titles
    .split(",")[2]]
```

```
62 ])
63 def makeblastdb(input, output, outputDir, title):
64   command = "makeblastdb " +\
65        "-out=" + outputDir + " " +\
66        "-dbtype=prot " +\
67        "-in=" + input + " " +\
68        "-title=" + title
69
70   run(command)
71
72 @follows(velvetg)
73 @follows(makeblastdb)
74 @files([
75 [None, params.publish_dir + "/" + params.blastx_outs.split(",")[0],
    params.publish_dir + "/" + params.makeblastdb_titles.split(",")[0],
     params.publish_dir + "/" + params.velvet_output_dir + "/contigs.fa
    "],
76 [None, params.publish_dir + "/" + params.blastx_outs.split(",")[1],
    params.publish_dir + "/" + params.makeblastdb_titles.split(",")[1],
     params.publish_dir + "/" + params.velvet_output_dir + "/contigs.fa
    "],
77 [None, params.publish_dir + "/" + params.blastx_outs.split(",")[2],
    params.publish_dir + "/" + params.makeblastdb_titles.split(",")[2],
     params.publish_dir + "/" + params.velvet_output_dir + "/contigs.fa
    "]
78 ])
79 def blastx(input, output, db, faFile):
80   command = "blastx " +\
81        "-out=" + output + " " +\
82        "-db=" + db + " " +\
83        "-query=" + faFile
84
85   run(command)
86
87
88
89 pipeline_run([blastx], multiprocess=multiprocessing.cpu_count())
```

Listing H.2: Command line to invoke *Ruffus* and execute data parallel *pipeline* for study case 1.

```
python pipeline.py --publish_dir /home/dantas/Desktop/SharedFolder/
    Ruffus/DataParallelism/Outputs/minimalOutputs --trimmomatic_input /
    home/dantas/Desktop/SharedFolder/_Common_/inputs/minimalInputs/
    ERR406040.fastq --trimmomatic_illuminaclip /home/dantas/Desktop/
    SharedFolder/_Common_/inputs/minimalInputs/TruSeq3-SE.fa --
    trimmomatic_output ERR406040.filtered.fastq --velvet_output_dir
    velvetDir --makeblastdb_ins /home/dantas/Desktop/SharedFolder/_
    Common_/inputs/minimalInputs/allrefs.fna\?.pro --makeblastdb_titles
     allrefsA,allrefsB,allrefsC --blastx_outs blastA.out,blastB.out,
    blastC.out 2>&1 | tee ./execution.out
```

I

Data Parallelism with *Swift*

Listing I.1: *Swift* data parallel *pipeline* for study case 1.

```
type file;


global string trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar";


app (file output) trimmomatic (file input, file illuminaclipFile)
{
  java "-jar" trimmomaticDir
  "SE"
  "-phred33"
  filename(input)
  filename(output)
  "ILLUMINACLIP:" + filename(illuminaclipFile) + ":2:30:10"
  "SLIDINGWINDOW:4:15"
  "LEADING:3"
  "TRAILING:3"
  "MINLEN:36";
}

app (file velvetHOutputFiles[]) velveth (file trimOutput, string
    velvetDir)
{
  velveth
```

```
  "__root__" + velvetDir
  "21"
  "-fastq"
  filename(trimOutput);
}

app (file velvetGOutputFiles[]) velvetg (file velvetHOutputFiles[],
    string velvetDir)
{
  velvetg
  "__root__" + velvetDir
  "-cov_cutoff"
  "5";
}

app (file o[]) makeblastdb (string outDir, file allrefs, string title)
{
  makeblastdb
  "-out=" + outDir
  "-dbtype=prot"
  "-in=" + filename(allrefs)
  "-title=" + title;
}

app blastx (file makeBlastDBOutputs[], file velvetGOutputs[], string
    out, string db, file query)
{
  blastx
  "-out=" + out
  "-db=" + db
  "-query=" + filename(query);
}

(string location, string pattern) splitMakeBlastDBIn(string arg)
{
  string path[] = strsplit(arg, "/");
  pattern = path[length(path)-1];

  string locationAux[];
  foreach directory, idx in path {
    if(idx != length(path)-1) {
      locationAux[idx] = directory;
    }
  }

```

```
67   location = strjoin(locationAux, "/");
68 }
69
70
71
72 file illuminaclipFile<single_file_mapper; file=arg("trimmomatic_
       illuminaclip")>;
73 file trimInput<single_file_mapper; file=arg("trimmomatic_input")>;
74 file trimOutput<single_file_mapper; file=arg("publish_dir")+"/"+arg("
       trimmomatic_output")>;
75 trimOutput = trimmomatic(trimInput, illuminaclipFile);
76
77 string velvetDir = arg("publish_dir") + "/" + arg("velvet_output_dir");
78
79 string velvetHFiles[] = [velvetDir+"/Log", velvetDir+"/Roadmaps",
       velvetDir+"/Sequences"];
80 file velvetHOutputs[] <array_mapper; files=velvetHFiles>;
81 velvetHOutputs = velveth(trimOutput, velvetDir);
82
83 string velvetGFiles[] = [velvetDir+"/contigs.fa", velvetDir+"/Graph",
       velvetDir+"/LastGraph", velvetDir+"/PreGraph", velvetDir+"/stats.
       txt"];
84 file velvetGOutputs[] <array_mapper; files=velvetGFiles>;
85 velvetGOutputs = velvetg(velvetHOutputs, velvetDir);
86
87 string inFilesLocation;
88 string inFilesPattern;
89 (inFilesLocation, inFilesPattern) = splitMakeBlastDBIn(arg("makeblastdb
       _ins"));
90 file allrefs[]<filesys_mapper; location=inFilesLocation, pattern=
       inFilesPattern>;
91 string titles[] = strsplit(arg("makeblastdb_titles"), ",");
92 file makeBlastDBOutputs[][];
93 foreach inFile, idx in allrefs {
94   string title = titles[idx];
95   makeBlastDBOutputs[idx] = makeblastdb(arg("publish_dir")+"/"+title,
         inFile, title);
96 }
97
98 string blastOuts[] = strsplit(arg("blastx_outs"), ",");
99 foreach makeBlastDBOutput, idx in makeBlastDBOutputs {
100   string title = titles[idx];
101   string blastDB = arg("publish_dir") + "/" + title;
102   string blastOut = arg("publish_dir") + "/" + blastOuts[idx];
```

```
103  blastx(makeBlastDBOutput, velvetGOutputs, blastOut, blastDB,
         velvetGOutputs[0]);
104 }
```

Listing I.2: Command line to invoke *Swift* and execute data parallel *pipeline* for study case 1.

```
1 /home/dantas/swift-0.96.2/bin/swift pipeline.swift -publish_dir=/home/
    dantas/Desktop/SharedFolder/Swift/DataParallelism/Outputs/
    minimalOutputs -trimmomatic_input=/home/dantas/Desktop/SharedFolder
    /_Common_/inputs/minimalInputs/ERR406040.fastq -trimmomatic_
    illuminaclip=/home/dantas/Desktop/SharedFolder/_Common_/inputs/
    minimalInputs/TruSeq3-SE.fa -trimmomatic_output=ERR406040.filtered.
    fastq -velvet_output_dir=velvetDir -makeblastdb_ins=/home/dantas/
    Desktop/SharedFolder/_Common_/inputs/minimalInputs/allrefs.fna?.pro
     -makeblastdb_titles=allrefsA,allrefsB,allrefsC -blastx_outs=blastA
    .out,blastB.out,blastC.out 2>&1 | tee ./execution.out
```

NGSPipesV2 Antlr Grammar Definition

Listing J.1: *NGSPipesV2* anltrl grammar definition.

```
grammar Pipes;

root: valueDeclaration* properties? valueDeclaration* repositories
    valueDeclaration* outputs? valueDeclaration* steps EOF;

valueDeclaration: parameterDeclaration | variableDeclaration;
parameterDeclaration: parameterName '=' parameterValue;
parameterName: 'params.' ID;
parameterValue: directValue;
variableDeclaration: variableName '=' variableValue;
variableName: ID;
variableValue: directValue;

properties: 'Properties' ':' '{' authorProperty? descriptionProperty?
    versionProperty? documentationProperty? '}';
authorProperty: 'author' ':' STRING;
descriptionProperty: 'description' ':' STRING;
versionProperty: 'version' ':' STRING;
documentationProperty: 'documentation' ':' '[' STRING? (',' STRING)* ']
    ';

```

```
repositories: 'Repositories' ':' '[' repository (repository)* ']';
repository: toolRepository | pipelineRepository;
toolRepository: 'ToolRepository' repositoryId ':' '{' locationProperty
    configProperty? '}';
pipelineRepository: 'PipelineRepository' repositoryId ':' '{'
    locationProperty configProperty? '}';
repositoryId: ID;
locationProperty: 'location' ':' locationValue;
locationValue: STRING;
configProperty: 'config' ':' '{' config* '}';
config: configName ':' configValue;
configName: ID;
configValue: value;

outputs: 'Outputs' ':' '{' output* '}';
output: outputId ':' outputValue;
outputId: ID;
outputValue: stepId '[' outputName ']';
outputName: ID;

steps: 'Steps' ':' '[' step (step)* ']';
step: 'Step' stepId ':' '{' execProperty executionContextProperty?
    inputsProperty? spreadProperty? '}';
stepId: ID;
execProperty: 'exec' ':' (commandReference | pipelineReference);
commandReference: repositoryId '[' toolName ']' '[' commandName ']';
toolName: ID;
commandName: ID;
pipelineReference: repositoryId '[' pipelineName ']';
pipelineName: ID;
executionContextProperty: 'execution_context' ':' value;
inputsProperty: 'inputs' ':' '{' inputProperty* '}';
inputProperty: inputName ':' inputValue;
inputName: ID;
inputValue: value | chain;
chain: stepId '[' outputName ']';
spreadProperty: 'spread' ':' '{' spreadInputsToSpreadProperty
    spreadStrategyProperty?'}';
spreadStrategyProperty: 'strategy' ':' combineStrategy;
strategyValue: combineStrategy | inputName;
combineStrategy: oneToOneStrategy | oneToManyStrategy;
oneToOneStrategy: 'one_to_one' '(' strategyValue ',' strategyValue ')';
oneToManyStrategy: 'one_to_many' '(' strategyValue ',' strategyValue ')
    ';
```

```
58 spreadInputsToSpreadProperty: 'inputs_to_spread' ':' '[' inputName (',' inputName)* ']';
59
60 value: directValue | indirectValue;
61 indirectValue: parameterName | variableName;
62 directValue: STRING | NUMBER | BOOLEAN | array;
63 array: '[' directValue (',' directValue)* ']' | '[' ']';
64
65 ID: ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
66 STRING: '"' ( '\\"' | . )*? '"';
67 BOOLEAN: ('true' | 'false');
68 NUMBER: '-'? INT ('.' [0-9] +)?;
69 fragment INT: '0' | [1-9] [0-9]*;
70
71 COMMENT: '/*' .*? '*/' -> skip; // .*? matches anything until the first */
72 WS: [ \t\r\n]+ -> skip; // skip spaces, tabs, newlines
```

Task Parallelism with *NGSPipesV2*

Listing K.1: *NGSPipesV2* task parallel *pipeline* for study case 1.

```
1
2  Properties: {
3    author: "NGSPipes Team"
4    description: "Study case 1"
5    version: "1.0"
6    documentation: ["http://ngspipes.readthedocs.io/en/latest/
         RunningExamples.html"]
7  }
8
9  repoLocation = "https://github.com/ngspipes2/tools_support"
10
11 Repositories: [
12   ToolRepository repo: {
13     location: repoLocation
14   }
15 ]
16
17 Outputs: {
18   output1: trimmomatic[outputFile]
19   output2: blastx[out]
20 }
21
22 Steps: [
23   Step trimmomatic: {
```

```
24    exec: repo[Trimmomatic][trimmomatic]
25    execution_context: "DockerConfig"
26    inputs: {
27      mode: "SE"
28      quality: "phred33"
29      inputFile: params.trimmomatic_input
30      outputFile: params.trimmomatic_output
31      fastaWithAdaptersEtc: params.trimmomatic_illuminaclip
32      seed_mismatches: 2
33      palindrome_clip_threshold: 30
34      simple_clip_threshold: 10
35      windowSize: 4
36      requiredQuality: 15
37      leading_quality: 3
38      trailing_quality: 3
39      minlen_length: 36
40    }
41  }
42  Step blastx: {
43    exec: repo[Blast][blastx]
44    execution_context: "DockerConfig"
45    inputs: {
46      db: makeblastdb[out]
47      query: velvetg[contigs_fa]
48      out: params.blastx_out
49    }
50  }
51  Step velveth: {
52    exec: repo[Velvet][velveth]
53    execution_context: "DockerConfig"
54    inputs: {
55      output_directory: "velvetdir"
56      hash_length: 21
57      file_format: "fastq"
58      filename: trimmomatic[outputFile]
59    }
60  }
61  Step velvetg: {
62    exec: repo[Velvet][velvetg]
63    execution_context: "DockerConfig"
64    inputs: {
65      output_directory: velveth[output_directory]
66      cov_cutoff: 5
67    }
68  }
```

```
69   Step makeblastdb: {
70     exec: repo[Blast][makeblastdb]
71     execution_context: "DockerConfig"
72     inputs: {
73       dbtype: "prot"
74       out: "allrefs"
75       title: "allrefs"
76       in: params.makeblastdb_in
77     }
78   }
79 ]
```

L

Data Parallelism with *NGSPipesV2*

Listing L.1: *NGSPipesV2* data parallel *pipeline* for study case 1.

```
1
2  Properties: {
3  author: "NGSPipes Team"
4  description: "Study case 1"
5  version: "1.0"
6  documentation: ["http://ngspipes.readthedocs.io/en/latest/
       RunningExamples.html"]
7  }
8
9  Repositories: [
10 ToolRepository repo: {
11 location: "E:\\Work\\NGSPipes\\ngspipes2\\main\\engine\\engine_core\\
       src\\test\\resources\\tools_support"
12 }
13 ]
14
15 Outputs: {
16 output1: trimmomatic[outputFile]
17 output2: blastx[outFile]
18 }
19
20 Steps: [
21   Step trimmomatic: {
22     exec: repo[Trimmomatic][trimmomatic]
```

```
23    execution_context: "DockerConfig"
24    inputs: {
25      mode: "SE"
26      quality: "phred33"
27      inputFile1: params.trimmomatic_input
28      output: trimmomatic_output
29      fastaWithAdaptersEtc: params.trimmomatic_illuminaclip
30      seedMismatches: 2
31      palindromeClipThreshold: 30
32      simpleClipThreshold: 10
33      windowSize: 4
34      requiredQuality: 15
35      leadingQuality: 3
36      trailingQuality: 3
37      minlenLength: 36
38    }
39  }
40  Step velveth: {
41    exec: repo[Velvet][velveth]
42    execution_context: "DockerConfig"
43    inputs: {
44      outputDirectory: "velvetdir"
45      hashLength: 21
46      fileFormat: "fastq"
47      filename: trimmomatic[outputFile]
48    }
49  }
50  Step velvetg: {
51    exec: repo[Velvet][velvetg]
52    execution_context: "DockerConfig"
53    inputs: {
54      outputDirectory: velveth[outDir]
55      covCutoff: 5
56    }
57  }
58  Step makeblastdb: {
59    exec: repo[Blast][makeblastdb]
60    execution_context: "DockerConfig"
61    inputs: {
62      dbtype: "prot"
63      out: ["allrefs", "allrefsB", "allrefsC"]
64      title: ["allrefs", "allrefsB", "allrefsC"]
```

```
65      in: ["E:\\Desktop\\NGSPipesTeam\\Demo\\Inputs\\caseStudy1Minimal\\
            allrefs.fna.pro", "E:\\Desktop\\NGSPipesTeam\\Demo\\Inputs\\
            caseStudy1Minimal\\allrefs.fnaB.pro","E:\\Desktop\\NGSPipesTeam
            \\Demo\\Inputs\\caseStudy1Minimal\\allrefs.fnaC.pro"]
66    }
67    spread: {
68      inputs_to_spread: [in, out, title]
69      strategy: one_to_one(in, one_to_one(out, title))
70    }
71  }
72  Step blastx: {
73    exec: repo[Blast][blastx]
74    execution_context: "DockerConfig"
75    inputs: {
76      db: makeblastdb[outFileName]
77      query: velvetg[contigsFa]
78      out: ["blast.out", "blastB.out", "blastC.out"]
79    }
80    spread: {
81      inputs_to_spread: [db, out]
82      strategy: one_to_one(db, out)
83    }
84  }
85]
```

M

Nested *Pipeline* with *Nextflow*

Listing M.1: *Nextflow* nested *pipeline* for study case 1.

```
1
2  trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar"
3
4  process trimmomatic {
5    publishDir params.publish_dir, mode: 'copy', overwrite: true
6
7    output:
8      file params.trimmomatic_output into trimVelvChannel
9
10   """
11   java -jar $trimmomaticDir \
12   SE \
13   -phred33 \
14   '${params.trimmomatic_input}' \
15   '${params.trimmomatic_output}' \
16   ILLUMINACLIP:'${params.trimmomatic_illuminaclip}':2:30:10 \
17   SLIDINGWINDOW:4:15 \
18   LEADING:3 \
19   TRAILING:3 \
20   MINLEN:36
21   """
22 }
23
24 process velvet {
```

```
25  input:
26    file velvetInput from trimVelvChannel
27
28  publishDir params.publish_dir, mode: 'copy', overwrite: true
29
30  output:
31    file "${params.velvet_output_dir}" into velvetOutputsChannel
32    file "${params.velvet_output_dir}/contigs.fa" into
          velvGBlastXChannel
33
34  """
35  nextflow /home/dantas/Desktop/velvet.nf --publish_dir \$PWD --
      trimmomatic_output '${params.publish_dir}/$velvetInput' --velvet_
      output_dir ${params.velvet_output_dir}
36  """
37 }
38
39 process makeblastdb {
40  publishDir params.publish_dir, mode: 'copy', overwrite: true
41
42  output:
43    file "${params.makeblastdb_title}.*" into makeBlastBlastXChannel
44
45  """
46  makeblastdb \
47  -out='${params.makeblastdb_title}' \
48  -dbtype=prot \
49  -in='${params.makeblastdb_in}' \
50  -title='${params.makeblastdb_title}'
51  """
52 }
53
54 process blastx {
55  input:
56    file blastDir from makeBlastBlastXChannel
57    file query from velvGBlastXChannel
58
59  publishDir params.publish_dir, mode: 'copy', overwrite: true
60
61  output:
62    file params.blastx_out
63
64  """
65  blastx \
66  -out='${params.blastx_out}' \
```

```
  -db='${params.publish_dir}/${params.makeblastdb_title}' \
  -query=$query
  """
}
```

Listing M.2: `Velvet` steps *pipeline* with *Nextflow*.

```
process velveth {
  publishDir params.publish_dir, mode: 'copy', overwrite: true

  output:
    file "${params.velvet_output_dir}" into velvhVelvgChannel

  """
  velveth \
  '${params.velvet_output_dir}' \
  21 \
  -fastq \
  '${params.trimmomatic_output}'
  """
}

process velvetg {
  input:
    file velvetGInput from velvhVelvgChannel

  publishDir params.publish_dir, mode: 'copy', overwrite: true

  output:
    file "$velvetGInput"
    file "$velvetGInput/contigs.fa"

  """
  velvetg \
  $velvetGInput \
  -cov_cutoff 5
  """
}
```

Nested *Pipeline* with *CWL*

Listing N.1: *CWL* nested *pipeline* for study case 1.

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

cwlVersion: cwl:v1.0
class: Workflow

requirements:
  - class: StepInputExpressionRequirement
  - class: InlineJavascriptRequirement
  - class: SubworkflowFeatureRequirement

inputs:
  - id: trimmomatic_input
    type: File
  - id: trimmomatic_illuminaclip
    type: File
  - id: trimmomatic_output
    type: string
  - id: velvet_output_dir
    type: string
  - id: makeblastdb_in
    type: File
  - id: makeblastdb_title
    type: string
  - id: blastx_out
```

```
    type: string

outputs:
  - id: trimmomaticOutput
    type: File
    outputSource: trimmomatic/output
  - id: velvetgOutput
    type: Directory
    outputSource: velvet/velvetgOutput
  - id: makeBOutput
    type:
      type: array
      items: File
      outputSource: makeblastdb/output
  - id: blastOutput
    type: File
    outputSource: blastx/output

steps:
  - id: trimmomatic
    run: Descriptions/trimmomatic.cwl
    in:
      - id: mode
        valueFrom: "SE"
      - id: quality
        valueFrom: "-phred33"
      - id: input_file
        source: "#trimmomatic_input"
      - id: output_file
        source: "#trimmomatic_output"
      - id: SLIDINGWINDOW
        valueFrom: "4:15"
      - id: LEADING
        valueFrom: "3"
      - id: TRAILING
        valueFrom: "3"
      - id: MINLEN
        valueFrom: "36"
      - id: illuminaclip_file
        source: "#trimmomatic_illuminaclip"
      - id: ILLUMINACLIP
        valueFrom: ${ return inputs.illuminaclip_file.location.replace("
            file://", "") + ":2:30:10";}
    out:
      - id: output
```

```
69
70  - id: velvet
71    run: velvet.cwl
72    in:
73      - id: velvet_output_dir
74        source: "#velvet_output_dir"
75      - id: trimmomatic_output
76        source: "#trimmomatic/output"
77    out:
78      - id: velvetgOutput
79      - id: velvetgContigs
80
81  - id: makeblastdb
82    run: Descriptions/makeblastdb.cwl
83    in:
84      - id: in
85        source: "#makeblastdb_in"
86      - id: title
87        source: "#makeblastdb_title"
88      - id: dbtype
89        valueFrom: "prot"
90    out: [output, phr]
91
92  - id: blastx
93    run: Descriptions/blastx.cwl
94    in:
95      - id: out
96        source: "#blastx_out"
97      - id: phrFile
98        source: "#makeblastdb/phr"
99      - id: db
100       valueFrom: ${return inputs.phrFile["location"].replace("file://",
              "").replace(".phr", "");}
101     - id: query
102       source: "#velvet/velvetgContigs"
103   out:
104     - id: output
```

Listing N.2: `Velvet` steps *pipeline* with *CWL*.

```
1 #!/usr/bin/env cwl-runner
2
3 cwlVersion: cwl:v1.0
4 class: Workflow
```

```
5
6  requirements:
7    - class: StepInputExpressionRequirement
8    - class: InlineJavascriptRequirement
9
10 inputs:
11   - id: velvet_output_dir
12     type: string
13   - id: trimmomatic_output
14     type: File
15
16 outputs:
17   - id: velvetgOutput
18     type: Directory
19     outputSource: velvetg/output
20   - id: velvetgContigs
21     type: File
22     outputSource: velvetg/contigs
23
24 steps:
25   - id: velveth
26     run: Descriptions/velveth.cwl
27     in:
28       - id: output_directory
29         source: "#velvet_output_dir"
30       - id: hash_length
31         default: 21
32       - id: file_format
33         valueFrom: "-fastq"
34       - id: file
35         source: "#trimmomatic_output"
36     out:
37       - id: output
38
39   - id: velvetg
40     run: Descriptions/velvetg.cwl
41       in:
42       - id: output_directory
43         source: "#velveth/output"
44       - id: cov_cutoff
45         default: 5
46     out:
47       - id: output
48       - id: contigs
```

Nested *Pipeline* with *Ruffus*

Listing O.1: *Ruffus* nested *pipeline* for study case 1.

```
from ruffus import *
import os
import multiprocessing

parser = cmdline.get_argparse()
parser.add_argument("--publish_dir")
parser.add_argument("--trimmomatic_input")
parser.add_argument("--trimmomatic_illuminaclip")
parser.add_argument("--trimmomatic_output")
parser.add_argument("--velvet_output_dir")
parser.add_argument("--makeblastdb_in")
parser.add_argument("--makeblastdb_title")
parser.add_argument("--blastx_out")
params = parser.parse_args()

trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar"

def run(command):
  print("::RUNNING:" + command)
  os.system(command)

@files(params.trimmomatic_input, params.publish_dir + "/" + params.
    trimmomatic_output, params.trimmomatic_illuminaclip + ":2:30:10")
def trimmomatic(input, output, illuminaclipFile):
```

```
24   command = "java -jar " + trimmomaticDir + " " +\
25   "SE " +\
26   "-phred33 " +\
27   input + " " +\
28   output + " " +\
29   "ILLUMINACLIP:" + illuminaclipFile + " " +\
30   "SLIDINGWINDOW:4:15 " +\
31   "LEADING:3 " +\
32   "TRAILING:3 " +\
33   "MINLEN:36"
34
35   run(command)
36
37 @follows(trimmomatic)
38 @files(params.publish_dir + "/" + params.trimmomatic_output, params.
       velvet_output_dir)
39 def velvet(input, output):
40   command = "python " +\
41   "velvet.py " +\
42   "--publish_dir " + params.publish_dir + " " +\
43   "--trimmomatic_output " + input + " " +\
44   "--velvet_output_dir " + output
45
46   run(command)
47
48 @files(params.makeblastdb_in, None, params.publish_dir + "/" + params.
       makeblastdb_title)
49 def makeblastdb(input, output, outputDir):
50   command = "makeblastdb " +\
51   "-out=" + outputDir + " " +\
52   "-dbtype=prot " +\
53   "-in=" + input + " " +\
54   "-title=" + params.makeblastdb_title
55
56   run(command)
57
58 @follows(velvet)
59 @follows(makeblastdb)
60 @files(None, params.publish_dir + "/" + params.blastx_out, params.
       publish_dir + "/" + params.makeblastdb_title, params.publish_dir +
       "/" + params.velvet_output_dir + "/contigs.fa")
61 def blastx(input, output, db, faFile):
62   command = "blastx " +\
63   "-out=" + output + " " +\
64   "-db=" + db + " " +\
```

```
65   "-query=" + faFile
66
67   run(command)
68
69
70
71 pipeline_run([blastx], multiprocess=multiprocessing.cpu_count())
```

Listing O.2: `Velvet` steps *pipeline* with *Ruffus*.

```
1  from ruffus import *
2  import os
3  import multiprocessing
4
5  parser = cmdline.get_argparse()
6  parser.add_argument("--publish_dir")
7  parser.add_argument("--trimmomatic_output")
8  parser.add_argument("--velvet_output_dir")
9  params = parser.parse_args()
10
11 def run(command):
12   print("::RUNNING:" + command)
13   os.system(command)
14
15 @files(params.trimmomatic_output, params.publish_dir + "/" + params.
       velvet_output_dir)
16 def velveth(input, output):
17   command = "velveth " +\
18   output + " " +\
19   "21 " +\
20   "-fastq " +\
21   input
22
23   run(command)
24
25 @follows(velveth)
26 @files(params.publish_dir + "/" + params.velvet_output_dir, None)
27 def velvetg(input, output):
28   command = "velvetg " +\
29   input + " " +\
30   "-cov_cutoff 5"
31
32   run(command)
33
```

```
34
35
36 pipeline_run([velvetg], multiprocess=multiprocessing.cpu_count())
```

Neste *Pipeline* with *Swift*

Listing P.1: *Swift* nested *pipeline* for study case 1.

```
type file;


global string trimmomaticDir = "/home/dantas/trimmomatic-0.32.jar";


app (file output) trimmomatic (file input, file illuminaclipFile)
{
  java "-jar" trimmomaticDir
  "SE"
  "-phred33"
  filename(input)
  filename(output)
  "ILLUMINACLIP:" + filename(illuminaclipFile) + ":2:30:10"
  "SLIDINGWINDOW:4:15"
  "LEADING:3"
  "TRAILING:3"
  "MINLEN:36";
}

app (file velvetOutputFiles[]) velvet (file trimOutput)
{
  "/home/dantas/swift-0.96.2/bin/swift"
  "/home/dantas/Desktop/velvet.swift"
```

```
25   "-publish_dir=" + arg("publish_dir")
26   "-trimmomatic_output=" + filename(trimOutput)
27   "-velvet_output_dir=" + arg("velvet_output_dir");
28 }
29
30 app (file o[]) makeblastdb (string outDir, file allrefs, string title)
31 {
32   makeblastdb
33   "-out=" + outDir
34   "-dbtype=prot"
35   "-in=" + filename(allrefs)
36   "-title=" + title;
37 }
38
39 app blastx (file makeBlastDBOutputs[], file velvetGOutputs[], string
       out, string db, file query)
40 {
41   blastx
42   "-out=" + out
43   "-db=" + db
44   "-query=" + filename(query);
45 }
46
47
48
49 file illuminaclipFile<single_file_mapper; file=arg("trimmomatic_
       illuminaclip")>;
50 file trimInput<single_file_mapper; file=arg("trimmomatic_input")>;
51 file trimOutput<single_file_mapper; file=arg("publish_dir")+"/"+arg("
       trimmomatic_output")>;
52 trimOutput = trimmomatic(trimInput, illuminaclipFile);
53
54 string velvetDir = arg("publish_dir") + "/" + arg("velvet_output_dir");
55 string velvetFiles[] = [velvetDir+"/Log", velvetDir+"/Roadmaps",
       velvetDir+"/Sequences"];
56 file velvetOutputs[] <array_mapper; files=velvetFiles>;
57 velvetOutputs = velvet(trimOutput);
58
59 file allrefs<single_file_mapper; file=arg("makeblastdb_in")>;
60 file makeBlastDBOutputs[] <filesys_mapper; location=arg("publish_dir"),
        pattern=arg("makeblastdb_title")+"*">;
61 makeBlastDBOutputs = makeblastdb(arg("publish_dir")+"/"+arg("
       makeblastdb_title"), allrefs, arg("makeblastdb_title"));
62
63 string blastOut = arg("publish_dir") + "/" + arg("blastx_out");
```

```
string blastDB = arg("publish_dir") + "/" + arg("makeblastdb_title");
blastx(makeBlastDBOutputs, velvetOutputs, blastOut, blastDB,
    velvetOutputs[0]);
```

Listing P.2: `Velvet` steps *pipeline* with *Swift*.

```
type file;

app (file velvetHOutputFiles[]) velveth (string trimOutput, string
    velvetDir)
{
  velveth
  "__root__" + velvetDir
  "21"
  "-fastq"
  trimOutput;
}

app (file velvetGOutputFiles[]) velvetg (file velvetHOutputFiles[],
    string velvetDir)
{
  velvetg
  "__root__" + velvetDir
  "-cov_cutoff"
  "5";
}

string dir = arg("publish_dir") + "/" + arg("velvet_output_dir");

string velvetHFiles[] = [dir+"/Log", dir+"/Roadmaps", dir+"/Sequences"
    ];
file velvetHOutputs[] <array_mapper; files=velvetHFiles>;
velvetHOutputs = velveth(arg("trimmomatic_output"), dir);

string velvetGFiles[] = [dir+"/contigs.fa", dir+"/Graph", dir+"/
    LastGraph", dir+"/PreGraph", dir+"/stats.txt"];
file velvetGOutputs[] <array_mapper; files=velvetGFiles>;
velvetGOutputs = velvetg(velvetHOutputs, dir);
```

Nested *Pipeline* with *NGSPipesV2*

Listing Q.1: *NGSPipesV2* nested *pipeline* for study case 1.

```
Properties: {
  author: "NGSPipes Team"
  description: "Study case 1"
  version: "1.0"
  documentation: ["http://ngspipes.readthedocs.io/en/latest/
      RunningExamples.html"]
}

Repositories: [
  ToolRepository repo: {
    location: "https://github.com/ngspipes2/tools_support"
  }
  PipelineRepository pipelines: {
    location: "https://github.com/ngspipes2/tools_support"
  }
]

Outputs: {
  output1: trimmomatic[outputFile]
  output2: blastx[out]
}

Steps: [
  Step trimmomatic: {
```

```
    exec: repo[Trimmomatic][trimmomatic]
    execution_context: "DockerConfig"
    inputs: {
      mode: "SE"
      quality: "phred33"
      inputFile: "ERR406040.fastq"
      outputFile: "ERR406040.filtered.fastq"
      fastaWithAdaptersEtc: "TruSeq3-SE.fa"
      seed_mismatches: 2
      palindrome_clip_threshold: 30
      simple_clip_threshold: 10
      windowSize: 4
      requiredQuality: 15
      leading_quality: 3
      trailing_quality: 3
      minlen_length: 36
    }
  }
  Step blastx: {
    exec: repo[Blast][blastx]
    execution_context: "DockerConfig"
    inputs: {
      db: makeblastdb[out]
      query: velvet[contigs]
      out: params.blastx_out
    }
  }
  Step velvet: {
    exec: pipelines[velvet]
    inputs: {
      trimmomatic_output: trimmomatic[outputFile]
      velvet_output_dir: "velvetDir"
    }
  }
  Step makeblastdb: {
    exec: repo[Blast][makeblastdb]
    execution_context: "DockerConfig"
    inputs: {
      dbtype: "prot"
      out: "allrefs"
      title: "allrefs"
      in: "allrefs.fna.pro"
    }
  }
]
```

Listing Q.2: `Velvet` steps *pipeline* with *NGSPipesV2*.

```
Properties: {
  author: "NGSPipes Team"
  description: "Study case 1"
  version: "1.0"
  documentation: ["http://ngspipes.readthedocs.io/en/latest/
      RunningExamples.html"]
}

Repositories: [
  ToolRepository repo: {
    location: "https://github.com/ngspipes2/tools_support"
  }
]

Outputs: {
  contigs: velveth[contigs_fa]
}

Steps: [
  Step velveth: {
    exec: repo[Velvet][velveth]
    execution_context: "DockerConfig"
    inputs: {
      output_directory: params.velvet_output_dir
      hash_length: 21
      file_format: "fastq"
      filename: params.trimmomatic_output
    }
  }
  Step velvetg: {
    exec: repo[Velvet][velvetg]
    execution_context: "DockerConfig"
    inputs: {
      output_directory: velveth[output_directory]
      cov_cutoff: 5
    }
  }
]
```