NGSPipes: From Specification to Automatic Deployment of NGS pipelines*

Bruno Dantas[1,2], Calmenelias Fleitas[1,2], Alexandre P Francisco[1,3], José Simão[1,2], and Cátia Vaz[1,2]

[1] INESC-ID Lisboa
[2] Instituto Superior de Engenharia de Lisboa (ISEL)
[3] Instituto Superior Técnico, Universidade de Lisboa

Abstract. Biosciences have been revolutionized by next generation sequencing (NGS) technologies in last years, leading to new perspectives in medical, industrial and environmental applications. And although our motivation comes from biosciences, the following is true for many areas of science: published results are usually hard to reproduce either because data is not available or tools are not readily available, which delays the adoption of new methodologies and hinders innovation. Our focus is on tool readiness and pipelines availability. Even though most tools are freely available, pipelines are in general barely described and their configuration is far from trivial, with many parameters to be tuned. In this paper we discuss how to effectively build and use pipelines, relying on state of the art computing technologies to execute them without users need to configure, install and manage tools, servers and complex workflow management systems. A framework is also proposed showing that we can have public pipelines ready to process and analyse very high volume experimental data, produced for instance by high-throughput technologies, and that can be executed by users without effort. The NGSPipes framework and underlying architecture provides a major step towards open science and true collaboration in what concerns tools and pipelines among computational biology researchers and practitioners, which may share and replicate results in an easier and transparent way. It is freely available at http://ngspipes.github.io/.

1 Introduction

Nowadays most scientific experiments that employ next-generation sequencing (NGS) rely on running and refining a series of intertwined computational analysis and visualization tasks on large amounts of data. These so called analyses pipelines, or more generally workflows, start with voluminous raw sequences and end with detailed structural, functional, and evolutionary results. Pipelines involve the use of multiple software tools and data resources in a staged fashion,

*This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

with the output of one tool being passed as input to the next one. A personalized medicine pipeline based on NGS technology can start for instance with short DNA sequences (reads) of an individual human genome and end with a diagnostic and prognostic report [13,25]. It can even end with a treatment plan if clinical data are available. This kind of pipelines depends on the use of multiple software tools to assess the quality of reads, to map them to a reference human genome, to identify sequence variations, to query databases for the sake of associating variations to diseases, and to check for novel variants. All these tools must be parametrized, a task that is in general far from trivial.

The data made available and the description of NGS scientific and industrial experiment analyses that is kept, sometimes in publications or database records, are also almost never sufficient to repeat those analyses or even to verify and assess results [11]. At the same time, the amount of data generated in scientific experiments is outpacing enhancements in computational power and storage capabilities. This is especially true for life sciences, where new technologies increased the sequencing throughput from kilobytes to terabytes per day. Experiments that employ NGS lead in general to challenges in reproducibility due to a lack of standards, exceedingly large dataset sizes, and increasingly complex computational tools. The usage of multiple data sources and computational tools in these studies further complicate reproducibility.

To simplify the design and execution of biomedical workflows by end users, especially those that use multiple software tools and data resources, a number of scientific workflow systems have been developed over the past decade. Scientific workflows correspond to series of structured activities and computations that arise in scientific problem solving. They involve the invocation of a number and variety of analysis tools. Therefore, one of the main purposes of these systems is to overcome the problem of accessibility of computation tools and multiple data sources to the end users without programming expertise. Examples include Taverna [22], Kepler [18], Galaxy [10], Conveyor [17], Pegasus [8], Gene Pattern [23], Tavaxy [1], and Swift [26]. Such workflow systems have an abstract representation of a workflow in the form of a directed acyclic graph (DAG), where nodes represent tasks to be executed and edges represent either data flow or execution dependencies between different tasks. Thus, the workflow system maps the edges and nodes in the graph to real data and software components. The workflow engine is responsible for executing the software components either locally on the user machine or remotely, for instance using cloud services. Some of these scientific workflow systems may use high performance computing facilities, if available, for processing large volumes of data concurrently. But most of these scientific workflow systems cannot be easily installed and configured, being most of the times only available to users with access to some kind of specialized IT support.

We propose here a framework architecture and an implementation that relies on: a flow-based executable language for the specification of pipelines, repositories for tools, a virtual environment assembler, and a standalone execution en-

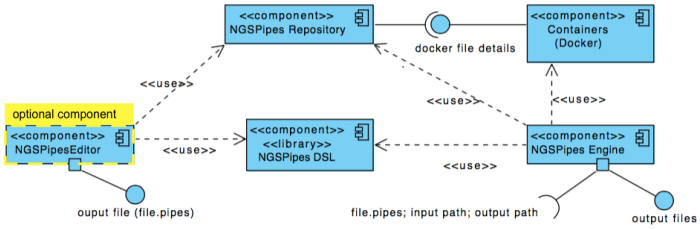

Fig. 1. Component diagram that describes the architecture of the overall NGSPipes framework.

gine. We developed also an user-friendly editor prototype for specifying pipelines as a proof of concept.

These components allow us to have an ubiquitous open system aiming to meet the above requirements. Namely, one of our main contributions is a pipeline specification language, with a clear separation between the language and the execution engine. This language is suitable for end users with or without programming expertise, and without compromising the expressive power for describing pipelines (or more generally data flow processing within a directed acyclic graph model). Moreover, by being system independent, we believe that the proposed language will allow pipelines to be transparently exported and reused within different systems in use.

The proposed framework aims also for the decoupling of concrete data and tools from workflows/pipelines specification. This is particularly important if we take into account data privacy and tools licensing, essential issues for the scientific and industry communities. The architecture of the proposed framework was designed to support the execution of pipelines without users need to configure, install and manage tools, servers and complex workflow management systems. Moreover, given a pipeline to execute (described through the specification language), all the execution environment is automatically setup and the pipeline is executed.

The remaining paper is organized in three main parts: framework architecture and implementation description, an illustrative case study, and discussion. The framework and related prototypes are open source and readily available online.

2 Implementation

Let us introduce the *NGSPipes* framework as proof of concept. As shown in Figure 1, the framework architecture comprises three main components:

- *The specification language*, a domain specific language (DSL) for describing pipelines with the just enough expressive power.
- *Repositories of tools* that contain the description of each tool available for integrating within pipelines. We note that new tools can be easily added and new repositories can be made available independently.

– *The execution engine* which given a pipeline to execute (described using above language), automatically sets up all the execution environment and executes the pipeline.

All these components are independent, easily extensible and reusable, allowing a seamless integration of new tools for data analysis and processing. Note that decoupling pipeline and tools specifications from data sources and real tools leads to a more flexible framework. We can for instance run the pipeline using different, but compatible, versions of the same tool. As we will discuss later, we would only need to change the repository of tools being referenced by the pipeline specification.

2.1 Tools Repository

Each *repository of tools* contains all the information related to a set of available tools for constructing pipelines (see Figure 1). Such information includes details such as what is necessary to install and/or execute a given tool, and where we can fetch it. Thus, the pipeline definition does not need to include these details and, on the other hand, we are able to automatically assemble the execution environment.

For each available *tool*, the repository should include a *tool descriptor* and at least a *tool configurator*. The *tool descriptor* is the entity responsible for supplying all the information on how to run a given tool, such as available commands and arguments, processor options and memory requirements. This information should be described according to the specification documentation [7] and discussed below. Let us take as an example the tool Velvet [27]. It includes the commands `velvetg` and `velveth`, and a fragment of its descriptor is shown in Figure 2. A descriptor must include at least: the tool *name*, the tool *version*, memory requirements (*requiredMemory*), *setup* scripts to be executed on execution environment setup, and available *commands*. Each command is described following a similar approach: the command *name*, the (real) *command* to be executed, the *priority* of the command, *arguments* and *outputs* generated, and the argument composer (*argumentComposer*) for specifying how to link arguments to values.

A *tool configurator* includes the information needed to define the execution context for a given tool: the *name* of the file where the execution context is defined, the name of the execution context (the *builder*), the *setup* scripts that must be executed for assembling the execution context, and the *uri* that identifies and allows to fetch the tool. Figure 3 shows an example where the tool is provided by a docker image and, thus, it is necessary to install docker in the execution context [20].

2.2 Specification Language

The *specification language* is a DSL for describing pipelines. It contains primitive building blocks with the enough expressiveness to define data processing

```
"name" : "Velvet",
"version" : "0.7.01",
"setup" : [ "make" ],
"requiredMemory": 12288,
"commands" :[
     { "name" : "velveth",
       "command" : "velveth",
       "priority" : 2,
       "arguments" : [
         { "name" : "output_directory",
           "outputType" : "outputDir",
           "isRequired" : "true",
           ...
         },
       ...
  }
  ...
```

Fig. 2. Partial descriptor for Velvet tool.

```
{ "name" : "DockerConfig",
  "builder" : "Docker",
  "uri" : "ngspipes/velvet0.7",
  "setup" : [
    "wget -qO- https://get.docker.com/ | sh"
 ]
}
```

Fig. 3. An example of a configurator for the Velvet tool.

pipelines, namely when data processing can be modelled as a directed acyclic graph. Figure 4 depicts partially the syntax of this language, given by a grammar. The full syntax of the language can be found in DSL documentation [5], using an EBNF notation alike. The primitives of the language are *Pipeline*, *tool*, *command*, *argument* and *chain*. Since a *Pipeline* implies the execution of one or more tools, its specification must reference the tools repository that is being used.

The reference to the repository found in the specification of a pipeline must identify not only where to find the repository, but also the type of repository: local or remote, like Github. As shown in Figures 4 and 5, the first line in a pipeline specification specifies the type of repository (`Github`) and where the repository can be found. Each tool used in a pipeline is then specified by providing its name, its configurator and the list of tool commands that will be executed within this pipeline. For instance, in the pipeline of Figure 5 the second tool is the Velvet tool, and DockerConfig is the chosen configurator. This information together with the repository information specifies the environment for executing Velvet commands. Note that the same command for a given tool may be executed several times and with different parameters, being listed more than once. Note also that the commands within different tools may be interleaved.

As mentioned before, each command in the pipeline appears in the context of a tool. For executing each command, it is necessary to identify its name, which is unique in the tool context, and to set the arguments for each required

pipeline ::= **Pipeline** *repositoryType repositoryLocation* { (*tool*) + };
tool ::= **tool** *toolName configurationName* { (*command*)+ };
command ::= **command** *commandName* { (*argument* — *chain*) + };
argument ::= **argument** *argumentName* argumentValue;
chain ::= **chain** *argumentName* ((*toolName*)? *commandName*)? *outputName*;

Fig. 4. Partial grammar for the specification language using EBNF notation.

```
Pipeline "Github" "https://github.com/ngspipes/Repository" {
  tool "Trimmomatic" "DockerConfig" {
    command "trimmomatic" {
      argument "mode" "SE"
      argument "quality" "-phred33"
      argument "inputFile" "ERR406040.fastq"
      argument "outputFile" "ERR406040.filtered.fastq"
      argument "fastaWithAdaptersEtc" "adapters/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"
    }
  }
}
```

Fig. 5. Example of a pipeline specification. See Section 3 for details concerning this pipeline and related case study.

parameters. For instance, in the pipeline of Figure 5, the argument `file_format` for command `velveth` has as argument "-fastq", *i.e.*, the input file for this command must be in FASTQ format.

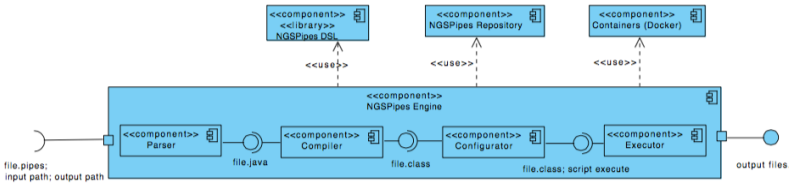

Fig. 6. Components diagram of the excution engine.

The specification language also includes the *chain* primitive for linking outputs into inputs. With this primitive we can define as an argument of a command an output file of other command. This primitive is used to specify execution flows. The output from each command may be files named internally by the command or named through command arguments. In both situations it is common that other commands use these output files for keep processing the pipeline. For instance, in command `blastx`, the argument `-query` receives as value the file "contigs_fa", which is an output of the command `velvetg` of tool `velvet`. The *chain* primitive has a simplified version, which can be used when the output is from the previous command in the pipeline specification. In this case, we only specify the name of the output file to chain with the given argument. As an example, we can see in Figure 5 the argument `filename` of `velveth` command chained with the output file, named as "outputFile", of command `trimmomatic`.

2.3 Engine

The *engine* is responsible for: the analysis of the pipeline description and transformation to an executable format, the setup of tools used in the pipeline description, and for the execution of the tools in an isolated context.

The pipeline description is transformed to an executable format. Because the pipeline can be specified outside the editor, language consistency checks must be applied. For the setup of the execution environment, the engine relies on information collected from the repository of tools referred in the pipeline specification. The orchestration and planning of execution is delegated to the language library. It checks the correct execution of the pipeline steps and outputs the relevant information to the user.

Figure 6 shows the main components of the *engine* and their interaction. The engine receives the pipeline description, the input path and the output path. Internally, the *engine* is divided in four components. The parser transforms a pipeline (described in the language presented in Section 2.2) to a representation in the Java language. Any grammatical errors are detected in this phase. The second component is the *compiler*, which will produce an executable pipeline with the correct invocation sequence. Note that no tools are embedded in this executable pipeline. They will be dynamically downloaded and executed only by the *executor*. The third component is the *configurator*, which is responsible for the configuration of the *executor* and for booting the pipeline execution phase.

The configuration data consists of the computational resources that will be available during execution (i.e. amount of memory and number of CPU cores) as well as input and output paths. Part of the configuration data is obtained automatically by looking at the executable pipeline and the *repository of tools*. By looking at the pipeline, the *configurator* determines which tools where used and, by looking at the *repository*, it determines the memory required to run each tool. The amount of memory set by the *configurator* will be the highest value among all the included tools.

The fourth component of the *engine* is the *executor* and it relies on two layers of virtualization. The first layer is a system-level type of virtual machine (VM). The current framework implementation relies on a widely used hypervisor to run this VM – the VirtualBox system, as described in engine documentation [6]. This allows the engine to be installed on any type of main stream operating system (e.g. OSX, Windows, Unix). Inside the virtual machine, a Linux-based operating system is ready to be executed. On top of this, the *engine* uses a lightweight virtualization technology to ensure proper installation, keep up-to-date, and run each command of the pipeline. Currently, the framework uses Docker containers technology [20]. Other solutions can be integrated in the future because both the pipeline language and repositories of tools are not compromised with this technology.

Before the actual steps of the pipeline are executed, the *engine* ensures that the VM is ready to use the container technology by checking if necessary packages are available. Once completed, the pipeline is executed, downloading and running the correct tool/command. The download part is done only in the first execution. After that, tools remain installed to favour speed and reproducibility. Because each command is executed in a separated container, the *executor* must ensure that the input files, located in the user environment, are made available to each tool.

The *engine* is available in two versions: a command line application and a graphical user interface (GUI) application. Both versions are functionally equivalent and are packed as a regular Java application. The execution of a given pipeline can be parametrized, including the input and output directories as well as hardware resources made available to the pipeline execution. The amount of memory can be limited, which overrides the value determined automatically by the *Configurator*. However, doing so can result in an execution error if insufficiency memory is specified for the data to be processed. To change the amount of memory is essential to know how much data is going to be processed. For instance, although it is recommended having 12 GBytes of physical memory for using the Velvet tool, in our case study we will only need 4 GBytes of memory since for bacterial strain sequencing we can use less memory in general.

3 Case study

We consider a standard pipeline used on epidemiological surveillance using NGS data. The aim is to characterize bacterial strains through allelic profiles [19].

When sequencing a bacterial strain by paired end methods with desired depth coverage of 100x (in average each position in the genome will be covered by 100 reads), the output from the sequencer will be two FASTQ files containing the reads. Each read typically will have 90-250 nucleotides length, using Illumina technology. The first data processing step is to trim the reads for removing the adapters used in the sequencing process and any tags used to identify the experiment in a run.

From clean reads two approaches can be followed: *de novo* assembly or mapping to a reference genome.

In *de novo* assembly, software such as Velvet [27] or SPAdes [3] is used to obtain a draft genome composed of contigs, longer DNA sequences resulting from assembling multiple reads. Annotation software such as Prokka [24] can then take contigs as input and determine the gene content and annotate it against multiple databases. Alternatively, the draft genome can be compared to databases of gene alleles for multiple loci using BLAST [2]. Given BLAST results we can create an allelic profile characterizing the strain [19].

In mapping approaches, a reference genome is chosen and the reads are directly mapped against it using read mapping software such as BWA [15] or Bowtie2 [14]. The output is a file containing the relative position of each read in the reference genome. That file is then processed to determine the positions that have single nucleotide polymorphisms (SNPs) when compared to the reference genome [16]. The resulting SNPs are then analysed to determine if they might be the result of recombination events [4], and filtered out if they are to be used in phylogenetic analysis. Several allelic or SNP profiles for different strains resulting from both approaches can then be compared to determine their phylogenetic relationships using different methods [12,9].

The pipeline in Figure 5 follows the *de novo* assembly approach and relies on BLAST for comparing the draft genome to a database of gene alleles. We relied on data from NCBI Sequence Read Archive for testing and evaluating the framework, namely data on *Streptococcus pneumoniae*. See the use case documented within engine documentation [6] for more details. Figure 7 shows execution times when running this pipeline. The systems used to run the pipeline differ in hardware and operating system, as presented in Figure 7.a). To evaluate the engine performance we execute it assigning 2 cores (parameter `-cpus 2`) and 4 GBytes of RAM (parameter `-mem 4`). The size of the FASTQ input file is 814 MBytes. The pipeline (see Figure 5) and initial data was used a first time, which we call *cold execution*. During this run, the engine automatically installs the necessary tools and keeps them installed for the second and following executions, which we call *warm execution*. Figure 7.b) depicts the results for these two scenarios and the four systems described in Figure 7.a). Depending on the system, the pipeline takes between 38 and 42 minutes to execute. We also note that keeping the tools installed for new executions is a good option since the speedup of a warm execution varies between 7% (system C) and 23% (system B).

System	OS	RAM	Disk type
A	Windows 10	8 GB	SSD
B	Windows 10	16 GB	HDD
C	OS X Yosemite	8 GB	SSD
D	Slackware 14.0	256 GB	HDD

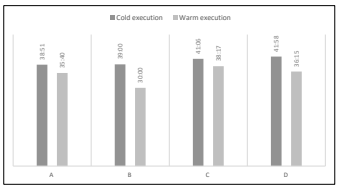

Fig. 7. (a) Operating system and hardware of four different experimental setups (b) Performance of cold and warm execution

4 Discussion

Although NGS data have being shared in recent years, we cannot yet talk about open science. Even if tools are available, analyses pipelines are often not detailed and clearly defined. This includes tools parametrization. Hence, it is almost impossible to reproduce published results, or to use exactly the same approach with different data, even with all data available.

NGSPipes framework is based on three principles. The first principle is to completely avoid servers and services configuration. The second one is to automatically get and configure only required tools. The third is to precisely describe pipelines. These three principles allow us to address almost all points raised in introduction. NGSPipes framework makes use of resources and environment isolation for making tools available, avoiding servers and services manual configuration.

This approach is well known in IT industry and is already being adapted for life sciences [21]. Such ecosystem is of crucial importance for NGSPipes framework and is being also adopted by traditional platforms, e.g., Galaxy [10] and Nextflow. This is an important step since we can share resources among many different systems and platforms. Note in particular that NGSPipes framework is agnostic with respect to the job performed by each tool. A tool can invoke remote Web services, fetch remote data, or even make use of cloud computing resources either directly or through other workflow systems. The main aim and novelty of NGSPipes framework is to provide a decoupled architecture for automatically setup and run pipelines without requiring users to deal with low level details of computer systems.

Still tools alone are not of much use, analyses pipelines must be made available, precisely defined, and platform independent. In this paper we propose a simplified specification language and support library for this purpose, based on a clear and straightforward syntax. Note that both language and library are completely independent from the execution engine. In particular it can be reused by any other platform. More general specification languages exist for specifying pipelines and workflows, being BPMN 2.0 the most well known. BPMN 2.0 is however too much richer and possibly introduces another layer of complexity for life sciences practitioners. Still we believe that the language proposed in this paper would benefit from being mapped and aligned with a subset of BPMN 2.0.

Although in present version NGSPipes framework can be easily used and integrated in cloud services since it relies on common cloud technologies, some issues remain. As raised in introduction, deployment on cloud should be transparent, in fact we would say that executing a pipeline should be as easy as downloading a file from Web. There is however some work to be done as cloud technologies mature and become commodity. Task parallelization and distribution is another issue. The heterogeneous nature of NGS data and analyses jobs, relying on different tools, lead to rather different computational workloads. In this context both task scheduling and resources provision planning should be aware of workload patterns.

References

1. Abouelhoda, M., Issa, S.A., Ghanem, M.: Tavaxy: Integrating taverna and galaxy workflows with cloud computing support. BMC bioinformatics 13(1), 1 (2012)
2. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of molecular biology 215(3), 403–410 (1990)
3. Bankevich, A., Nurk, S., Antipov, D., Gurevich, A.A., Dvorkin, M., Kulikov, A.S., Lesin, V.M., Nikolenko, S.I., Pham, S., Prjibelski, A.D., et al.: Spades: a new genome assembly algorithm and its applications to single-cell sequencing. Journal of Computational Biology 19(5), 455–477 (2012)
4. Croucher, N.J., Page, A.J., Connor, T.R., Delaney, A.J., Keane, J.A., Bentley, S.D., Parkhill, J., Harris, S.R.: Rapid phylogenetic analysis of large samples of recombinant bacterial whole genome sequences using gubbins. Nucleic acids research p. gku1196 (2014)
5. Dantas, B., Fleitas, C., Francisco, A.P., Simão, J., Vaz, C.: Ngspipes dsl documentation. https://github.com/ngspipes/dsl/wiki, accessed on June 2016.
6. Dantas, B., Fleitas, C., Francisco, A.P., Simão, J., Vaz, C.: Ngspipes engine documentation. https://github.com/ngspipes/engine/wiki, accessed on June 2016.
7. Dantas, B., Fleitas, C., Francisco, A.P., Simão, J., Vaz, C.: Ngspipes tools documentation. https://github.com/ngspipes/tools/wiki, accessed on June 2016.
8. Deelman, E., Singh, G., Su, M.H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., et al.: Pegasus: A framework for mapping complex scientific workflows onto distributed systems. Scientific Programming 13(3), 219–237 (2005)
9. Francisco, A.P., Bugalho, M., Ramirez, M., Carriço, J.A.: Global optimal eburst analysis of multilocus typing data using a graphic matroid approach. BMC bioinformatics 10(1), 1 (2009)
10. Giardine, B., Riemer, C., Hardison, R.C., Burhans, R., Elnitski, L., Shah, P., Zhang, Y., Blankenberg, D., Albert, I., Taylor, J., et al.: Galaxy: a platform for interactive large-scale genome analysis. Genome research 15(10), 1451–1455 (2005)
11. Goodman, A., Pepe, A., Blocker, A.W., Borgman, C.L., Cranmer, K., Crosas, M., Di Stefano, R., Gil, Y., Groth, P., Hedstrom, M., et al.: Ten simple rules for the care and feeding of scientific data. PLoS Comput Biol 10(4), e1003542 (2014)
12. Huson, D.H.: Splitstree: analyzing and visualizing evolutionary data. Bioinformatics 14(1), 68–73 (1998)
13. Koboldt, D.C., Ding, L., Mardis, E.R., Wilson, R.K.: Challenges of sequencing human genomes. Briefings in bioinformatics p. bbq016 (2010)

14. Langmead, B., Salzberg, S.L.: Fast gapped-read alignment with bowtie 2. Nature methods 9(4), 357–359 (2012)
15. Li, H., Durbin, R.: Fast and accurate long-read alignment with burrows–wheeler transform. Bioinformatics 26(5), 589–595 (2010)
16. Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R., et al.: The sequence alignment/map format and samtools. Bioinformatics 25(16), 2078–2079 (2009)
17. Linke, B., Giegerich, R., Goesmann, A.: Conveyor: a workflow engine for bioinformatic analyses. Bioinformatics 27(7), 903–911 (2011)
18. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the kepler system. Concurrency and Computation: Practice and Experience 18(10), 1039–1065 (2006)
19. Maiden, M.C., van Rensburg, M.J.J., Bray, J.E., Earle, S.G., Ford, S.A., Jolley, K.A., McCarthy, N.D.: Mlst revisited: the gene-by-gene approach to bacterial genomics. Nature Reviews Microbiology 11(10), 728–736 (2013)
20. Matthias, K., Kane, S.P.: Docker: Up & running. shipping reliable containers in production. O'Reilly Media (2015)
21. Moreews, F., Sallou, O., Ménager, H., Monjeaud, C., Blanchet, C., Collin, O., et al.: Bioshadock: a community driven bioinformatics shared docker-based tools registry. F1000Research 4 (2015)
22. Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., et al.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics 20(17), 3045–3054 (2004)
23. Reich, M., Liefeld, T., Gould, J., Lerner, J., Tamayo, P., Mesirov, J.P.: Genepattern 2.0. Nature genetics 38(5), 500–501 (2006)
24. Seemann, T.: Prokka: rapid prokaryotic genome annotation. Bioinformatics p. btu153 (2014)
25. Voelkerding, K.V., Dames, S.A., Durtschi, J.D.: Next-generation sequencing: from basic research to diagnostics. Clinical chemistry 55(4), 641–658 (2009)
26. Wozniak, J.M., Wilde, M., Foster, I.T.: Language features for scalable distributed-memory dataflow computing. In: Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on. pp. 50–53. IEEE (2014)
27. Zerbino, D.R., Birney, E.: Velvet: algorithms for de novo short read assembly using de bruijn graphs. Genome research 18(5), 821–829 (2008)