Benefits of the Snakemake Workflow Management Software in Comparison to Traditional Programming

Josh Loecker, Patrick Ewing
North Central Agricultural Research Laboratory, USDA
Brookings, SD

Abstract

Tools surrounding bioinformatics have increased data acquisition and accuracy significantly, especially with near-real time results using nanopore DNA sequencing. With large amounts of data, reproducibility is of high importance, and long workflows can become convoluted. Snakemake, built on the Common Workflow Language and Python, aims to alleviate this with readable formatting, reproducibility, and portability for any machine. Using 97 fastq files, the usability of these three traits were compared between a Bash and Snakemake workflow using a range of one to twelve threads. In every test, Snakemake was faster than Bash. At its fastest, Snakemake was 27% faster than Bash. Reproducibility of both workflows was verified using an MD5 hash of results. The hashes differed between the workflows; this may be a result of executing the workflows in two different terminal environments. Despite this, it is a valid method of validating reproducibility between tests within individual workflows.

Outside speed tests, Snakemake offers quality of life features that allow it to pull ahead from Bash. Containerization of workflows using Conda is one example of this. The ability to require specific versions of software within a workflow boosts reproducibility. Additionally, portability is increased because the container can be deployed almost anywhere, and the required software can be downloaded on an as-needed basis. With readability comes maintainability. Snakemake will almost always pull ahead of Bash in this regard with its simple `input`, `output`, and `shell` fields.

The field of Bioinformatics is moving very quickly, and it can be difficult for traditional Bash scripts to keep up in certain aspects. While Bash is paramount in the execution of some software, more powerful tools like Snakemake are required to handle the execution of an entire, complex workflow.

Introduction

The first successful DNA sequence was the determination of twelve base pairs of a bacteriophage lambda by Wu in 1968 (Wu & Kaiser, 1968). Following this, Gilbert and Maxam were able to sequence twenty-four bases in two years, a rate of one base per month (Shendure et al., 2017). Current technology has risen leaps and bounds above this. Nanopore sequencing can yield upwards of 250 gigabytes of data in two days using devices that fit in the palm of a hand (Nanopore, 2021; Shendure et al., 2017). Attempting to manage gigabytes of DNA sequences gives rise to the idea of "Big Data", which can be understood – in basic terms – as a method used to analyze tremendous amounts of information in non-traditional methods. The first challenge is data acquisition, which has been overcome with "high throughput" sequencing, of which nanopore is an emerging technology (Labrinidis & Jagadish, 2012). This massive increase in ability to collect data is why bioinformatics has been considered to be one of the most data intensive fields, especially in terms of storage and computation requirements (Khan et al., 2019; Stephens et al., 2015). This shows the second challenge: processing these data, within a reasonable time frame, using tools that are not overly complex.

The challenge with processing data introduces reproducibility, scalability, and transparency of analysis. Reproducibility is the idea that a workflow should generate the same exact results, no matter where it is run, as long as the same settings are used. Scalability is the capacity to be able to increase resource usage based on resources available. If a cluster supercomputer is available, then all resources given should be used to their maximum potential. Finally, transparency of analysis should show the analysis methods in simple terms that are easy to understand.

Of these three, reproducibility is the most important. Despite this, *Science* magazine has reviewed dozens of scientific articles not providing an adequate model for reproducibility, which is a major concern when attempting to repeat an experiment or verify results (McNutt, 2014). Reproducibility is also a concern when extremely large datasets are being analyzed, which is where scalability of workflows are useful. The ability to scale resource usage with the size of a dataset as needed ensures the workflow is running at its most optimal abilities. Reproducibility and scalability are useful with large datasets because computation can be time consuming and repeating experiments is costly. Traditional scripts rely on linear approaches; an input path is stated as a command line argument, and the target script iterates over the file(s). Because of this, simple workflows can be made with ease; longer workflows, however, quickly become disorienting due to the potential lack of structure, ultimately reducing workflow transparency. In addition, attempting to customize parameters is more difficult without custom functions to read from a configuration file or parse parameters from the command line. This is because most parameters are defined in-line, next to the command when executed. If a change of commands is required, the specific line must be sought out and modified. As a result, specific parameters can become "lost" in a workflow, reducing transparency. Development on a local machine works well because it allows for faster build times and allows installation of any required software. However, it often lacks the computational power required for larger datasets. While pushing the project to a remote server is an option, installation of software requires administrative access – something not offered to users. The developer must work within the confines of their user space. The software used on a local machine may not be present on a remote one. While similar software may exist, if it is not the same between locations reproducibility cannot be verified.

These specific downfalls – reproducibility, command line parameters, and installation on a remote server – can be solved with the combination of the Common Workflow Language (CWL) and Conda (Khan et al., 2019; Koester, 2021). The Common Workflow Language is a framework that allows workflows to be "powerful, easy to use, portable, and support reproducibility" (Amstutz et al., 2016). Snakemake is built on the CWL and inherits these values. Conda is a package manager that allows software to be installed by non-administrative users in an environment isolated from the host operating system (Grüning et al., 2018). Snakemake emerges from a mix of the CWL and Conda (Koester, 2021). The downfall of traditional scripts, as described above, is where Snakemake shows its strength. Reproducibility and portability are foundations of Snakemake; built upon the Common Workflow Language, certification of results is easily done (Mölder et al., 2021). Additionally, Snakemake attempts to ease the second challenge, described above, with its human readable, Python-friendly formatting (Koester, 2021).

Steps within the Snakemake workflow are defined via rules. On average, each rule has an input and an output section, much like traditional scripts. Transparency benefits from the explicit input/output sections. Snakemake also parses all input independent of the workflow – no user defined functions are required to handle command line parameters. This is important because it ensures all parameters are handled in the same way, every time, boosting reproducibility. On top of this, extra time is not required to build the code infrastructure required to handle command line parameters. While Snakemake is installed through Conda, it also offers Conda support within itself. This means each rule can be executed within a specific version of a Conda package, promoting reproducibility. Nearly every remote server allows users to install required software through Conda, as it reduces security risks. Exporting the Conda environment file allows deployment on a new computer within minutes, which greatly helps with portability between machines. A comparison of the Snakemake Workflow using two versions of this "Conda Directive" are done to show speed comparisons between them. A more detailed explanation of this is described under the Materials & Methods section "Snakemake's Conda Directive".

The purpose of this work is to show that workflow management is required to improve reproducibility, portability, transparency, and show speedup improvements. Reproducibility and transparency be established with MD5 hashes and ease of multithreading. Portability will be verified by downloading prerequisite software on-demand. Speedup effects will be seen through increasing the number of threads available to the workflow by one, until the workflow is utilizing the maximum number of threads available. I expect these tests to be much more easily completed within Snakemake than in Bash due to Snakemake's infrastructure being built on these ideas. The results will be displayed through a series of speed tests between Bash and Snakemake.

Materials & Methods

Four total commands will be running within each of the workflows. These commands, and their descriptions, are listed in Table 1. These commands were executed on a 2019 MacBook Pro with a 6-core Intel i7 CPU and 16 GB of RAM. Samtools, Burrows-Wheeler Alignment, Snakemake, and Python 3.7 were used in the execution of this workflow (Danecek et al., 2021; Li & Durbin, 2009; Mölder et al., 2021; Rossum, 2018). The software was installed in a Conda environment, and the original workflow was adapted to a Snakemake workflow (Anaconda Software Distribution, 2021; Loecker, 2021/2021; *The Unix Shell: Automating a Workflow*, 2017). These fastq files were aligned against the Zymogen Community Reference Database (*ZymoBIOMICS Microbial Community DNA Standard*, n.d.). The general workflow procedure can be found in Exhibit 1.

Table 1: A list of commands to execute in the workflow

Command	Function
`bwa aln`	Align an input file to a reference database
`bwa samse`	Convert the output from `bwa aln` to a `sam` file
`samtools view`	Convert the `sam` file to a `bam` file. Much easier for computer manipulation, but near-impossible for human-readability
`samtools sort`	Sort the `sam` file with respect to their position in the genome. Allows for much easier human-readability

Exhibit 1: Commands to execute to run each workflow

```
git clone github.com/JoshLoecker/CapstoneProject
conda env create -f CapstoneProject/snakemake/environment.yaml -n capstone
conda activate capstone
```

Bash Execution

```
cd CapstoneProject/bash
bash run_alignment.sh ../testing_data/testing.fastq.gz
```

Snakemake Execution

```
cd CapstoneProject/snakemake
snakemake -j all
```

Bash Script Procedure
A modified Bash script was used to align 97 fastq files to the Zymogen Community reference database (*The Unix Shell: Automating a Workflow*, 2017). This script (from here known as the "Bash Script") takes an input file location as a command line parameter and performs the following steps.

1. Create a `results` subdirectory
2. Create `sai`, `sam`, `bam`, and `bam_sorted` subdirectories under the `results` directory
3. Execute `bwa aln` to align the incoming fastq file with the reference database.
4. Convert the aligned file into a machine-readable format with `bwa samse`.
5. Format the machine-readable into a human-readable format with `samtools view`
6. Sort reads based on their position in the genome with `samtools sort`

Snakemake Procedure
The same Bash Script was translated to a Snakefile. The same 97 fastq files were aligned to the same Zymogen Community reference database (*ZymoBIOMICS Microbial Community DNA Standard*, n.d.). Snakemake performs the following steps.

1. Look in the input directory for all files with the `.fastq.gz` extension
2. Use the requested input for `rule all` as the base rule and determine which rule(s) can produce the request input as its own output
3. Work backwards until the input of a rule cannot be matched to an output
4. Execute the rules in the following order, on each input file:
    a. Run `bwa aln` to align the incoming file with the reference database.
    b. Convert the aligned file into a machine-readable format with `bwa samse`.
    c. Format the machine-readable into a human-readable format with `samtools view`
    d. Sort reads based on their position in the genome with `samtools sort`

Analysis Procedure
A secondary Bash script was created for the Snakemake Procedure and Bash Script Procedure. This allowed for timing the runtime of scripts. As extreme precision was not required, the command `date +%s` was used. This shows the current Epoch time. Calculating the difference in Epoch between the end of the workflow and the start of the workflow, the total run time to the nearest seconds can be determined. These results were written to a file. To determine the speedup effects of parallelization, this secondary Bash script started each workflow a total of twelve times,

adding an additional thread at each iteration. Twelve total threads were utilized because this was the maximum number of threads available on the laptop used in testing. Reproducibility was tested by generating an MD5 hash of the contents of each output file for each command executed. Each hash was saved in a respective file. These files were hashed again, to produce four total hashes corresponding to the four commands ran.

Snakemake's Conda Directive

The Conda Directive allows for containerization of rules within their own Conda environment. This is useful because environments can be downloaded at runtime, allowing workflows to be downloaded on-demand. Two options are available for this directive are: 1) Download the environment at runtime and activate it for each input file or 2) Just activate the Conda environment for each input file. The Conda environment must be downloaded for the first run when using this directive. Subsequent executions will use the already-downloaded environment files. In the former scenario, when *N* files are used as input Snakemake downloads the `bam` and `samtools` environments. Then the environment containing `bam` activates *N* individual times, and the environment containing `samtools` activates *N* separate times. For the latter scenario, only the activation of environments is necessary, not downloading the environment. These Conda Directives are not available within Bash, and as such were only used to evaluate Snakemake.

Safety Considerations

There are no known immediate safety considerations with this research. Long term computer usage can cause carpel tunnel and eye fatigue.

Results

Speed Comparisons
A comparison of Bash and Snakemake workflows require them to produce identical output. A simple workflow to align DNA sequences to a reference databased was used (*The Unix Shell: Automating a Workflow*, 2017). This Bash script was translated to a Snakemake workflow, and 97 input files were used in analysis.

Table 2 shows the raw data containing the runtime for each workflow. A Graphic representation of this can be seen in Figure 1. The largest speed decrease was found when going from just one to two threads. At one thread, Snakemake and Bash took 630 seconds and 676 seconds, respectively. When using two threads, Snakemake dove to a 325 second runtime, and Bash took 414 seconds. This makes sense, as the computing power was doubled. Time differences between the two workflows can be seen in the "Difference" column.

Table 2: Bash and Snakemake workflow runtimes

Threads	Snakemake	Bash	Difference
-	Time (s)		
1	606	676	70
2	325	414	89
3	228	308	80
4	184	248	64
5	166	200	34
6	144	181	37
7	147	181	34
8	124	159	35
9	127	141	14
10	122	136	14
11	119	127	8
12	117	133	16

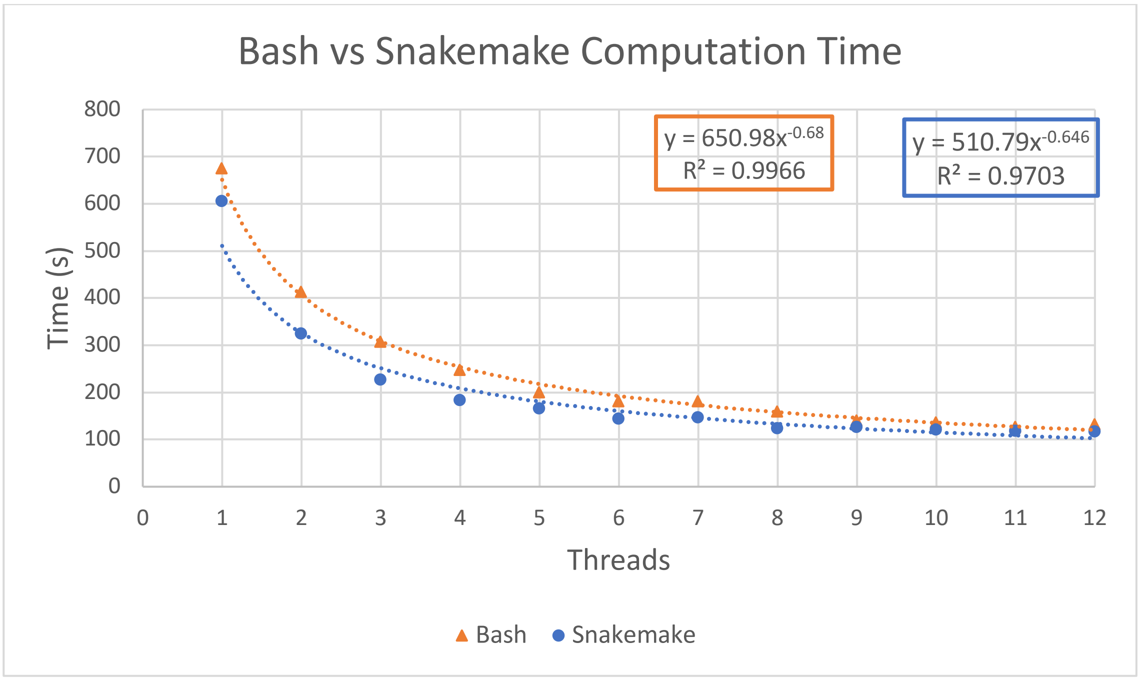

Figure 1: Computation time for Bash and Snakemake workflows

Snakemake also allows workflow rules to be contained in separate Conda environments. This means Snakemake is the only required Conda package to be installed at runtime; other required packages will be downloaded on an as-needed basis. Table 3 shows the runtime differences between two methods of running a workflow using this method. The "Download Environment" column states the length of time required to download the required Conda environment(s), install it, and run the workflow. The "Preinstalled Environment" column lists the length of time required to activate the environment and run the workflow. As expected, the Traditional workflow was the fastest in every scenario. At its fastest, it was 41% faster than the Downloaded workflow, and 28.5% faster than the Pre-Downloaded workflow. At the slowest times, the Traditional workflow was 21.1% faster than the Downloaded workflow and just 6.8% faster than the Pre-Downloaded workflow.

Reproducibility Analysis

To determine if reproducibility is possible within Snakemake and Bash, each workflow was ran using an increasing number of threads, from 1 to 12. An MD5 hash was generated from each of the files generated, resulting in 96 hashes. A second MD5 hash was created from these 96 hashes. This resulted twelve total hashes to compare, as opposed to 1,152 hashes split evenly among 12 files. These results can be seen in Table 4. Snakemake and Bash have the same resulting hash for `bwa aln`, but none of the other commands.

Table 3: Snakemake runtimes using Conda directives

Threads	Traditional	Downloaded	Pre-Downloaded
-	Time (s)		
1	606	809	772
2	325	413	393
3	228	310	268
4	184	232	210
5	166	201	181
6	144	186	185
7	147	178	157
8	124	175	155
9	127	167	146
10	122	157	140
11	119	151	135
12	117	147	133

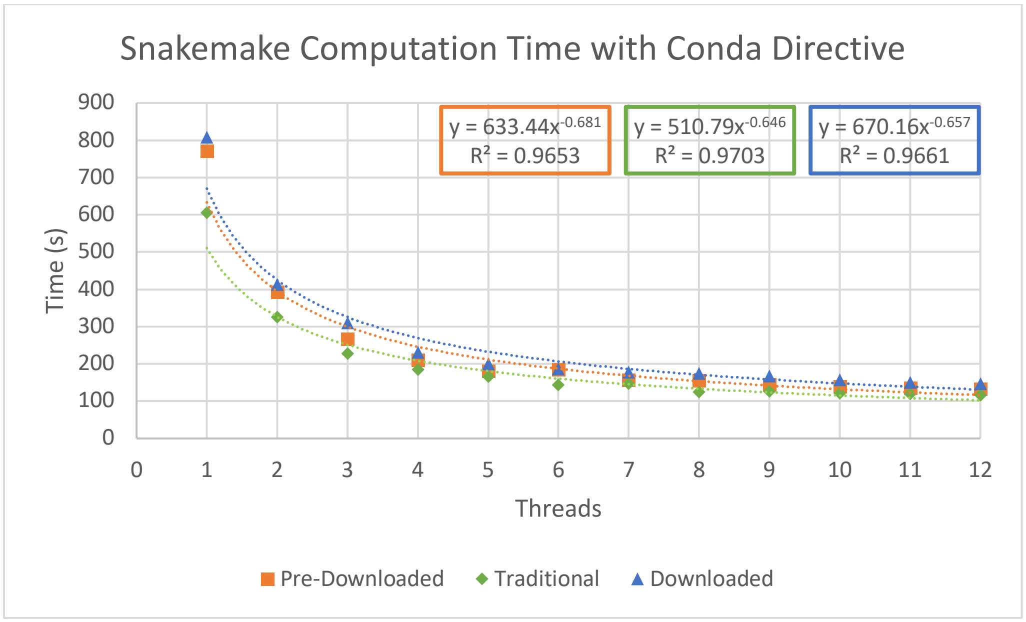

Figure 2: Graph of Snakemake workflow runtimes using Conda directive

Table 4: MD5 hash of results from Snakemake and Bash

Command	Hashes Match?
`bwa aln`	Yes
`bwa samse`	No
`samtools view`	No
`samtools sort`	No

Discussion

Table 3 shows the time elapsed for Bash and Snakemake to perform their routine using N number of threads. As expected, time elapsed decreases with an increase in thread count. What was not expected, however, was that Snakemake would have faster runtimes than bash in every case. While both Python and Bash are both interpreted languages, Snakemake has a slight overhead by translating from its own scripting language into the Python language. In theory, Bash is faster than Snakemake when given an unlimited number of threads, which can be seen with the slightly more-negative exponential value from Figure 1. However, large-scale computing is expensive, and it is unreasonable to assume infinite resources are available. When using the line of best fit to extrapolate on an ever-increasing number of threads, Bash will become just two-hundredths of a second faster than Snakemake when the workflows have access to 275 threads, with this specific dataset. This estimation may be inaccurate because it does not take into account the I/O time that is notorious in writing large amounts of data to traditional hard drives (Hsu & Smith, 2004). Again, assuming this many resources are available is unreasonable, and it is safe to say for this workflow, Snakemake was faster than Bash in every scenario.

It was difficult to ensure Bash was utilizing all threads made available to it. Appendix C shows an example snippet of the code used to enable multithreading within Bash. A downfall of Bash multithreading in this format is that output to STDOUT generally occurs after the main thread has completed its tasks, which means it can be challenging to truly know when all threads have finished processing. The terminal will allow keyboard input, despite other processes potentially printing data to the screen. This is just one example of where Snakemake has the upper hand. It is much easier to define *N* number of threads to be used for the job;

Snakemake will use these threads, and the main thread will not allow keyboard input until all jobs are complete.

To improve reproducibility and portability, Snakemake offers the ability to compartmentalize each rule in the workflow into specific Conda environments. This allows fine-tuning available packages, package versions, and ensures conflicts do not exist with software installed on the operating system (Koester, 2021). These processes add additional computation time to the workflow, although it is negligible. In theory, just ten hundredths of a second is required for a conda environment to be activated on the hardware used in this experiment. Adding slight overhead to activate the environment within Snakemake, it is safe to say thirteen hundredths of a second will be used to activate one Conda environment. Expanding this to the 96 input files used, with two environments to activate, approximately 24.96 additional seconds will be required just in the activation of environments. This performance difference can be seen in Table 3 and Figure 2. The key difference between these two workflows is that in the "Downloaded" plot, the Conda environment was downloaded before the workflow began. However, after downloading the Conda environment, it can be used in a near-identical fashion as the Traditional workflow. Figure 2 shows the differences between using this version of the cached workflow (or "Pre-Downloaded") and the "Traditional" workflow. Again, the Traditional workflow is faster because the Conda environments in the Pre-Downloaded Workflow must be activated for every file being processed, for each rule being ran. This amount of time is not significant enough to not use the compartmentalization features of Conda. Due to the uncertainty of download speeds, attempting to extrapolate how long it takes to download a Conda environment is not reliable.

Snakemake was always faster than – or as fast as – Bash. Snakemake's speed was especially seen when using all packages in a single Conda environment. At Snakemake's fastest, it was almost 1.5 minutes faster than Bash. Even at Snakemake's slowest runtime, it was 8 seconds faster than Bash. These results assume "Traditional" workflow uses.

Outside raw performance, other benefits exist within Snakemake that push the advantage far in its favor. Reproducibility is a requirement in research, and as previously stated, is a foundation of Snakemake (Koester, 2021; McNutt, 2014). To compare the differences in output from Snakemake and Bash, and MD5 algorithm is used. This function produces a 128-bit string of text that is distinct from any other string that could possibly be generated (Rivest & Dusse, 1992, p. 5). While the MD5 hashes generated from each workflow's results were consistent within the workflow, only the `bwa aln` command was identical between the workflows. This means the results between Snakemake and Bash were not identical, and therefore not reproducible, in any command except for the `bwa aln` command. The reason why this was the only hash that matched is unknown. One possible explanation is that Snakemake was ran in the terminal of an IDE while Bash was ran in a traditional terminal window.

While containers were not used in this test, they have a very common use-case. Singularity containers allow custom software to be installed on an operating system where the user does not have root privileges. When ran, the software will be encapsulated in its own miniature sandbox, where it can only reach resources specified at runtime. Much like using Conda environments, this does add a slight amount of overhead. However, Singularity allows for reproducibility and portability because the installed software has a specific version, and the runtime environment is consistent no matter where it is executed (Le & Paz, 2017). Snakemake's ability to use Singularity containers causes it to pull further ahead of Bash, especially when transparency is taken into account. While a Bash script can be executed within a Singularity container, Snakemake has a simple `container:` command that is easy to read, and ensures that individuals referencing the workflow understand exactly how commands are being executed. This option is not available in Bash, and it may be problematic to execute a Bash workflow with the same parameters as previous executions.

Outside reproducibility, maintainability is almost always a requirement for software development; transparency is a prerequisite for maintainability. When comparing Snakemake and Bash, it can be almost guaranteed that Snakemake pulls ahead in transparency as well. On average, Snakemake has three fields: `input`, `output`, and `shell`. When used in a Snakefile (Snakemake's equivalent to an executable file), it is quite easy to understand where files are coming from, where they will end up, and the command being ran on the file. An example of this can be seen in

Appendix **A**. Intuitively, the input for this rule is `my_input.txt` and the output is `my_output.txt`. The `shell` command will simply copy data from the input into the output, but any terminal command can be entered here. Bash has a very similar layout when using simple workflows, such as the one just mentioned. Appendix B shows a Bash script that will perform the same function as the Snakefile in

Appendix **A**. Input and output are specified, and the input is copied to the output. While the Bash script initially appears simple, it becomes more convoluted when multiple steps are in a workflow, which is often required.

Conclusion

While data are important, the basis of scientific discovery lies in the interpretation of these data. With an ever-increasing amount of data being gathered through nanopore sequencing, the ability to process these data requires smarter analysis to ensure verification and reproducibility. This idea started with the Common Workflow Language; Snakemake used this as a foundation and created a workflow management system that places reproducibility at its core. Without Snakemake, an unreasonably large amount of time would be spent managing, processing, and storing data to just obtain results – not in the interpretation of these results.

Bash is vital in the execution of command line programs, but it has fallen behind in its overall usefulness as a workflow manager. Snakemake offers a multitude of features – including executing Bash commands – that allow it to be more than just a scripting language. Snakemake's ease of parsing command line parameters, deployment to remote servers, and specific package versioning through Conda sets it far ahead of traditional workflow scripts.

Ultimately, data acquisition in bioinformatics is accelerating. Traditional scripts are falling further behind in their ability to handle this extensive amount of information being generated nearly in real-time. As a result, data analysis in bioinformatics is nearly impossible without the use of workflow management, including Snakemake, to ensure reproducibility and portability.

References

Amstutz, P., Crusoe, M. R., Tijanić, N., Chapman, B., Chilton, J., Heuer, M., & et al. (2016). *Common Workflow Language, v1.0* [Data set]. figshare. https://doi.org/10.6084/m9.figshare.3115156.v2

Anaconda Software Distribution. (2021). *Conda* (4.10.1) [Computer software]. Anaconda. docs.conda.io/

Danecek, P., Bonfield, J. K., Liddle, J., Marshall, J., Ohan, V., Pollard, M. O., Whitwham, A., Keane, T., McCarthy, S. A., Davies, R. M., & Li, H. (2021). Twelve years of SAMtools and BCFtools. *GigaScience*, *10*(2). https://doi.org/10.1093/gigascience/giab008

Grüning, B., Dale, R., Sjödin, A., Chapman, B. A., Rowe, J., Tomkins-Tinch, C. H., Valieris, R., & Köster, J. (2018). Bioconda: Sustainable and comprehensive software distribution for the life sciences. *Nature Methods*, *15*(7), 475–476. https://doi.org/10.1038/s41592-018-0046-7

Hsu, W. W., & Smith, A. J. (2004). The performance impact of I/O optimizations and disk improvements. *IBM Journal of Research and Development*, *48*(2), 255–289. https://doi.org/10.1147/rd.482.0255

Khan, F. Z., Soiland-Reyes, S., Sinnott, R. O., Lonie, A., Goble, C., & Crusoe, M. R. (2019). Sharing interoperable workflow provenance: A review of best practices and their practical application in CWLProv. *GigaScience*, *8*(giz095). https://doi.org/10.1093/gigascience/giz095

Koester, J. (2021, March 3). *Snakemake.* https://snakemake.readthedocs.io

Labrinidis, A., & Jagadish, H. V. (2012). Challenges and opportunities with big data. *Proceedings of the VLDB Endowment*, *5*(12), 2032–2033. https://doi.org/10.14778/2367502.2367572

Le, E., & Paz, D. (2017). Performance Analysis of Applications using Singularity Container on SDSC Comet. *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, 1–4. https://doi.org/10.1145/3093338.3106737

Li, H., & Durbin, R. (2009). Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics (Oxford, England)*, *25*(14), 1754–1760. https://doi.org/10.1093/bioinformatics/btp324

Loecker, J. (2021). *JoshLoecker/CapstoneProject* [Shell, Python]. https://github.com/JoshLoecker/CapstoneProject (Original work published 2021)

McNutt, M. (2014). Reproducibility. *Science*, *343*(6168), 229–229. https://doi.org/10.1126/science.1250475

Mölder, F., Jablonski, K. P., Letcher, B., Hall, M. B., Tomkins-Tinch, C. H., Sochat, V., Forster, J., Lee, S., Twardziok, S. O., Kanitz, A., Wilm, A., Holtgrewe, M., Rahmann, S., Nahnsen, S., & Köster, J. (2021). Sustainable data analysis with Snakemake. *F1000Research*, *10*, 33. https://doi.org/10.12688/f1000research.29032.1

Nanopore, O. (2021). *MinION*. Oxford Nanopore Technologies. http://nanoporetech.com/products/minion

Rivest, R., & Dusse, S. (1992). *The MD5 message-digest algorithm*. MIT Laboratory for Computer Science Cambridge.

Rossum, G. (2018). *Python* (3.7) [Python]. Python Software Foundation. https://python.org/

Shendure, J., Balasubramanian, S., Church, G. M., Gilbert, W., Rogers, J., Schloss, J. A., & Waterston, R. H. (2017). DNA sequencing at 40: Past, present and future. *Nature*, *550*(7676), 345–353. https://doi.org/10.1038/nature24286

Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., Iyer, R., Schatz, M. C., Sinha, S., & Robinson, G. E. (2015). Big Data: Astronomical or Genomical? *PLOS Biology*, *13*(7), e1002195. https://doi.org/10.1371/journal.pbio.1002195

The Unix Shell: Automating a workflow. (2017, June 11). https://thejacksonlaboratory.github.io/introduction-to-hpc/08-workflow/

Wu, R., & Kaiser, A. D. (1968). Structure and base sequence in the cohesive ends of bacteriophage lambda DNA. *Journal of Molecular Biology*, *35*(3), 523–537. https://doi.org/10.1016/S0022-2836(68)80012-9

ZymoBIOMICS Microbial Community DNA Standard. (n.d.). ZYMO RESEARCH. Retrieved April 22, 2021, from https://www.zymoresearch.com/products/zymobiomics-microbial-community-dna-standard

Appendix A

A very simple Snakefile with `input`, `output`, and `shell` directives

```
rule example:
    input: "my_input.txt"
    output: "my_output.txt"
    shell: "cat {input} >
{output}"
```

Appendix B

A simple Bash file moving input to output

```
input="my_input.txt"
output="my_output.txt"
cat "$input" > "$output"
```

Appendix C

An example code snippet showing the process used to enable Bash multithreading

```
core_count=2
(
for file in "[INPUT LOCATION]/"*;
do

    if (( i % core_count == 0 ));
then
        wait
    fi
    ((i++))
    start=$(date +%s)
    echo "Thread: $i"
    end=$(date +%s)
    total_time=$((end-start))
done
)
```