STAPLER

Simple and Swift Bioinformatics Pipeline Manager

Manual for version 18.04.19

Jaakko Tyrmi

jaakko.tyrmi@gmail.com

Documentation contents

1 Introduction

The purpose of this software is to allow you to create and manage bioinformatics pipelines in a quick and easy way. STAPLER may be the right workflow manager tool for any of the following scenarios:

1. You wish to use a bioinformatics program for large number of input files.
2. You wish to use several bioinformatics programs in sequence for your input files.
3. You want to quickly test different alternative workflows
4. You wish to run your analysis on supercomputer or cluster using one of the supported resource managers (LSF, SGE, SLURM or TORQUE).
5. You wish to parallelize the workflow.

STAPLER has a built-in support for many commonly used bioinformatics tools used in analyzing whole genome sequencing data. This native support means that the parameters being used with each tool are validated before run to make sure that all necessary variables have been defined appropriately to avoid unwanted errors during runtime. The vast number of bioinformatics software means that not all programs and versions (let alone your own custom-made programs) can be natively supported. Therefore, STAPLER also provides a simple syntax for incorporating almost any non-supported program to bioinformatics pipeline.

The documentation for STAPLER is divided into four places:

1. This manual for general description of the tool.
2. Contents of the *EXAMPLE_DIRECTORY* for examples of various workflows.
3. Built in help function of STAPLER for usage details for each bioinformatics tool. Type on command line:

```
$ python STAPLER.py --help
```

4. YouTube tutorial videos: https://www.youtube.com/channel/UCN3YLYAC1UxA_cUDxlAyEpw

2 Installation

2.1 *Requirements*

The only requirement for running STAPLER is Python v. 2.7.0 or later Python 2 version. Notice that Python 3 is not compatible. STAPLER has been tested to work on Linux and MacOS operating systems.

Notice that no third-party software are distributed with STAPLER, so the bioinformatics programs and the optional supercluster resource manager must be preinstalled on your platform or you must install them yourself. If resource manager is not available, jobs can be parallelized to run as a UNIX background processes.

STAPLER does not require superuser rights or modifications to environment variables etc. during installation (assuming that appropriate python version is installed already). Therefore, installation is possible even when you are just a client of a third-party computer cluster or supercomputer.

2.2 *Editing configuration file*

STAPLER workflows can contain two kinds of steps: steps using the supported third-party software with built-in STAPLER support and custom steps you define in command line while creating the workflow file.

If you intend to take advantage of STAPLER built-in tools you will need to modify the **config.txt** found in the STAPLER directory. The purpose of this file is to tell STAPLER how to run the supported software on the particular platform you are using., i.e. what is the path to each program executable file or is the software available in PATH, do you need to load module to use the program etc. You may check if commands are properly configured by using STAPLER built-in config.txt validation feature:

```
 $ python STAPLER.py --VALIDATE_CONFIG [options]
```

The config.txt file contains one row for each supported tool. Each row has four columns, which have the following purposes:

The **cmd_name** column: Contains the name of the tool (must match the internal representation of the tool in STAPLER, do not edit unless you are changing the Python source code as well).

The **execute** column: Tells STAPLER the path to the executable of the supported software on your platform. For example, let us assume you have bwa installed in a directory /my/dir/bwa_installation, in which you have the executable bwa. Therefore to execute bwa-mem you would normally write `$ /my/dir/bwa_installation/bwa mem`, appended with the necessary options. Therefore, the execute column should contain the string: */my/dir/bwa_installation/bwa mem*, as STAPLER will then later append the necessary parameters when generating workflows. However, if the tool is set as a PATH variable, it i.e. can be run from anywhere for example just by typing *bwa mem*, one can just use the PATH variable in the execute column: *bwa mem*. The config.txt that you can download from github is pre-configured with assumption all tools can be found from PATH. If the keyword *none* is used here, STAPLER assumes that this tool is not available at the current platform.

The **load_module** column: In many supercomputing environments different tools are incorporated into specific modules that must be loaded before the tool is usable. If loading a module is necessary to execute a command, the load command can be added to the "load_module" column. If there are several modules that need to be loaded, you may list them by using "!+" as delimit. If no module needs to be loaded, use the "none" keyword. By using this feature, you do not need to load the modules manually before submitting your job or add them to your input files manually as this is done automatically for you.

The **unload_module** column: Sometimes conflicts between different modules may occur, for instance to load a module another module is assumed to be loaded. When using a sequence of applications with varying module requirements, as in the case of STAPLER, such conflicts may arise if a module loaded for previous application does not allow for loading a module required by the next application. Therefore it is recommended to use the "unload_module" for defining the commands to unload the modules that were loaded in the "load_module" to return to the default environment of your platform. Use the "!+" as delimit if you need to unload several modules. Depending on the environment the module environment may be reset to default with "module reset" command.

3 Quick start examples

3.1 *Installation*

1. Download STAPLER and configure it for your platform by editing the *config.txt* file
2. Make sure that Python 2.7.0 (or later) and all the necessary third-party software you intend to use are properly installed
3. Check if the config.txt file is configured properly by typing

```
$ python STAPLER.py --VALIDATE_CONFIG
```

3.2 *Make use of STAPLER built-in help*

1. Check the list of available parameters and the usage help for each supported third-party bioinformatics tools

```
$ python STAPLER.py --help
$ python STAPLER.p --help <supported_software_name>
```

3.3 *Create a simple workflow*

1. Create your first workflow by creating a new *staplefile* (see the *EXAMPLE_DIRECTORY* for examples on how to do this)
2. Create a simple shell script of this workflow by typing
```
$ python STAPLER.py path/to/my_staplefile.txt
```
3. STAPLER then provides you with a path to the shell script to run the job. Do it by typing
```
$ source path/to/my_output_shell_script.sh
```

3.4 *Create a parallelized workflow*

1. Create a *staplefile* as you did above.
2. Create a parallelized shell script of this workflow by typing
```
$ python STAPLER.py --UNIX path/to/my_staplefile.txt
```
3. Run the job by typing
```
$ source path/to/my_output_shell_script.sh
```

3.5 *Create a parallelized workflow for a workload manager*

1. Create a *staplefile* as you did above, but this time include workload manager parameters (see the *EXAMPLE_DIRECTORY*)
2. Create an array job of this workflow (for SLURM workload manger in this example) by typing
```
$ python STAPLER.py path/to/my_staplefile.txt --SLURM
```
3. Submit the job to SLURM resource manager by typing
```
$ sbatch path/to/STAPLER_SBATCHABLE.sh
```

3.6 *Control the parallelization details*

1. Create a *staplefile* for the run.
2. Create a UNIX background job or workload manager job but include --MAX_JOB_COUNT parameter to set the maximum number processes to create. Also choose the parallelization priority mode with --PRIORITY parameter.
```
python STAPLER.py path/to/my_staplefile.txt --SLURM --MAX_JOB_COUNT 8
--PRIORITY s
```

3. Submit the job to SLURM resource manager by typing
```
$ sbatch path/to/STAPLER_SBATCHABLE.sh
```

3.7 *Check if the run has been successful*

1. Check if the run has been successful by typing
```
$ python STAPLER.py --VALIDATE_RUN path/to/my_staplefile.txt
```

3.8 *Fix and restart a failed run*

1. If the run validation feature has reported that the run has not been successfully finished, examine why this happened. Did the run use too much memory, was an incorrect parameter used, …?
2. If necessary, apply fixes to the original *staplefile* you used to initialize the failed workflow (edit parameters, allocate more memory to the resource manager etc.).
3. Create a new job using the fixed *staplefile*, but now include --FIX_RUN parameter to the command line to indicate that only failed commands need to be rerun (parallelization features are available as usual):
```
$ python STAPLER.py –-FIX_RUN path/to/my_staplefile.txt
```
4. Run the job by typing
```
$ source path/to/STAPLER_SBATCHABLE.sh
```

3.9 *Remove a workflow*

1. Remove the run by using the --REMOVE parameter and providing a path to the *staplefile* defining the workflow you wish to remove. This will remove the output directories and all files within those, but the starting point directory, or any files within, will not be removed.
```
$ python STAPLER.py --REMOVE path/to/my_staplefile.txt
```

3.10 *Compress your workflow steps to save space*

1. After running a job as in examples above, it is possible to compress all the steps of the workflow with the --COMPRESS parameter (parallelization features are available as usual). Decompression of compressed workflow is performed similarly, but --DECOMPRESS option is used instead.
```
$ python STAPLER.py path/to/my_staplefile.txt –-COMPRESS
```
2. Run the compression job by executing the newly created job.
```
$ source path/to/STAPLER_SBATCHABLE.sh
```

Read more detailed instructions for each of these steps below, from example files and from built in help:

```
$ python STAPLER.py --help
```

4 Creating a workflow

Hint: See the contents of "EXAMPLE_DIRECTORY" to better understand this part of the manual! The example data contains part of the raw data used in publication: Mattila et al. (2017) Genome-Wide Analysis of Colonization History and Concomitant Selection in *Arabidopsis lyrata*. Mol. Biol. Evol. 34(10):2665–2677.

This section provides an overview of STAPLER input and output. When STAPLER has been successfully installed you need three things to create a workflow:

1. **Staplefile** containing run information, including the run name, input data and a list of software that you wish to apply on your input data (e.g. *STAPLEFILE_SAMTOOLS_RUN.txt*)
2. **Starting point directory** containing **input data files** for your workflow (e.g. */EXAMPLE_DIRECTORY/ RAW_DATA/*)
3. **Project directory** to which stapler will create all directories and files of the workflow (e.g. *EXAMPLE_DIRECTORY/SAMTOOLS_RUN/*)

When you have created the workflow by running STAPLER with the staplefile, STAPLER will generate the following directories into the project directory:

1. **Workflow script directory** will contain all necessary scripts to execute the workflow.
2. **Output directories** will contain the output files of each workflow step output files after the workflow is being executed.

4.1 *Staplefile: modeling a workflow*

4.1.1 Required fields

The general concept and usage of staplefile is presented here, for more details see the staplefile examples in the EXAMPLE_DIR. Let us open for instance the STAPLEFILE_SAMTOOLS_RUN.txt.

In short, a staplefile defines the the input data for workflow and the sequence of tools (and their parameters) that you wish to use.

The purpose of the staplefile is to provide an easy and fast way to automate long workflows without complex manually created shell scripts and even more importantly, to allow you to parallelize workflows easily. They also allow you to quickly create different versions of the workflow by tuning parameters or adding/skipping certain tools.

A staplerfile should have the following contents:

1. First line must contain the string
   STAPLEFILE
2. **A mandatory line** defining job name must be found. The job name is used in the output directory and file names. E.g.
   JOB NAME: SAMTOOLS_RUN
3. **A mandatory line** defining the starting point for a workflow. This directory should contain all the input files for the current workflow. E.g.
   STARTING POINT DIR: /path/to/EXAMPLE_DIRECTORY/RAW_DATA

5. **A mandatory line** defining the project directory for the current run. STAPLER will create all output directories within this directory. E.g.
   PROJECT DIR: /path/to/EXAMPLE_DIRECTORY/SAMTOOLS_RUN
6. **An optional section** defining parameters for possible resource manager written between lines
   RESOURCE MANAGER:
   and
   RESOURCE MANAGER END:

   . STAPLER will automatically define a name for the run based on the JOB NAME line, so it should not be included here. STAPLER will automatically parallelize the workflow by creating an array job / multiple job run feature of the preferred resource manager, so the parallelization paremeters defining the array job / multiple job should not be defined by user. The often necessary parameters to define include run time limit, the name of the partition / queue to submit the job to, number of nodes and cores to use and the maximum amount of RAM to use. The STAPLEFILE_SAMTOOLS_RUN.txt file shows an example how to configure SLURM resource manager parameters:
   RESOURCE MANAGER:
   #!/bin/bash -l
   #SBATCH --time=1:00:00
   #SBATCH --partition=serial
   #SBATCH --ntasks=1
   #SBATCH --nodes=1
   #SBATCH --mem-per-cpu=4000
   RESOURCE MANAGER END:
7. **A mandatory section** listing the actual sequence of tools to use in the workflow written in between the lines
   COMMANDS:
   and
   COMMANDS END:

4.1.2 Incorporating software with built-in support

In the command section two types of commands can be used: tool with built-in support in STAPLER and custom tools. In the case of tools with built-in support, it is important to check how to use each tool with STAPLER. This information can be found using the built-in help of STAPLER. To list all supported tools, type:

```
$ python STAPLER.py --help
```

and to see how to use a specific supported tool, type:

```
$ python STAPLER.py --help <tool_name>
```

By default, STAPLER will automatically infer parameters that define input and output file paths. These should not therefore be defined by the user. Tools specific help also shows mandatory parameters, which must be incorporated to the command line. In some occasions parameters are required that are not part of final command line, but give STAPLER important information, for instance how paired-end fastq file pairs are named. These STAPLER-specific commands can be recognized from the use of exclamation mark, e.g. --!read_format parameter is required by the bwa mem implementation of STAPLER. Help function also tells what optional parameters can be used with the command. STAPLER will print a warning message if you are using a parameter which is not within mandatory or optional arguments.

4.1.3 Incorporating custom software

STAPLER also includes **custom commands** to enable the use of your own scripts or any other programs that are not supported by STAPLER. These commands must start with the keyword CUSTOM. Each CUSTOM command should indicate where the input and output files paths should be entered. Placeholder for input file path is defined by writing $INPUT and output file path placeholder with $OUTPUT string. Input and output file types can be defined by including a file extension to these keywords, e.g. $INPUT.fastq and $OUTPUT.sam. If a file extension is omitted from $INPUT, STAPLER will try to read in all files found in the input directory. If file extension is omitted from $OUTPUT, the file extension of the input file will be given to the output file.

When using paired end data files, one should use $PAIR1 and $PAIR2 strings as a placeholder for input files instead of the $INPUT string. Using the pair strings requires the use of identifiers to define how paired-end files pairs are named within curly braces, e.g. $PAIR1{_R1} and $PAIR2{_R2}. This this example each first pair file is expected to contain the string _R1 and each second pair file is expected to contain the string _R2.

If the command takes multiple input data files, which are not to be paired, one can define the $INPUT string multiple times, but in this case each input string must be given an index within curly braces:

$INPUT{1} $INPUT{2} $INPUT{3} …

In the case of defining multiple input files, it is a good idea to explicitly define which input file is used as the basename of output file(s). This can be done by adding curly braces to the output string containing the input string to use in the naming. For instance string $OUTPUT{$PAIR1} would use the first pair as a basename for this output file.

For example, if STAPLER would not support the bwa mem tool, we could look at the bwa mem manual for advice how to define the command line:

```
bwa mem ref.fa read1.fq read2.fq > aln-pe.sam
```

We would then replace the input and output paths accordingly for our SAMTOOLS_RUN example:

```
CUSTOM path/to/bwa mem /path/to/EXAMPLE_DIRECTORY/REFERENCE_GENOME
/a_lyrata_frigida.fasta $READ1.fastq $READ2.fastq >
$OUTPUT{$READ1.fastq}
```

If you need to load modules to run the program, you may run commands at the same row by delimiting them with two ampersand characters (&&), e.g.

```
CUSTOM module load mymodule && path/to/bwa mem
/path/to/EXAMPLE_DIRECTORY/REFERENCE_GENOME /a_lyrata_frigida.fasta
$READ1.fastq $READ2.fastq > $OUTPUT{$READ1.fastq}
```

4.2 *Generate workflow scripts*

When STAPLEFILE has been created, STAPLER can be run by typing:

```
$ python STAPLER.py /path/to/STAPLEFILE.txt [options]
```

You can check what options are available from the STAPLER built-in help:

```
$ python STAPLER.py --help
```

When STAPLER has finished creating the run scripts, you can check the project directory to see the directories created for the current workflow. **Output directories** have been created to contain output files for each workflow step. Notice that not all tools create an output directory. For instance tools used for indexing sequence, alignment and vcf files will usually create the index within the input directory. Notice that STAPLER does not execute the workflow scripts, so the output directories will remain empty until you run the workflow scripts.

The **workflow script directory** contains all the scripts necessary to run the workflow and a log file created by STAPLER for debugging. This directory will contain one or more workflow files, which you can now submit to the resource manager if you are using a super cluster or a supercomputer. If you are running the scripts on your own computer you can use the "source" to start the run.

Notice that if your workflow has been split to multiple parts, you must wait for the first part to finish before the next one can be run.

4.3 *File naming within workflows explained*

Internally STAPLER assigns an **id** for each input file. The id is a file name stripped of file path and suffix (i.e. the basename of the file). For instance, the base name of *EXAMPLE_DIRECTORY/C98_P2_R1.fastq* would be *C98_P2_R1*. In cases where several input files is read by a command and a single output file is written, STAPLER will infer the basename of the input files, assigns the first basename as the output file basename and adds the appropriate file extension. For instance mapping files *C98_P2_R1.fastq* and *C98_P2_R2.fastq* with bwa mem will create an output with name *C98_P2_R1.sam*.

4.4 *Parallelization features overview*

STAPLER can parallelize the workflows either by making them run as UNIX background tasks or by creating an array job / multiple job run for a resource manager often found in supercomputing environments (see an up-to-date list of supported resource managers by using the STAPLER built-in help).

Bioinformatics workflows often include steps where multiple files are merged. Let's consider a workflow where we have sequenced 4 samples with Illumina paired-end sequencing, so we have 8 FASTQ files as a starting point (fig 1). The workflow consists of FASTQ file trimming, alignment to reference, SAM to BAM conversion and processing, variant call and finally variant filtering. In trimming the FASTQ files STAPLER may create 8 separate processes – one for each FASTQ file. Only 4 processes can be created for the alignment step (and the same applies for the subsequent SAM/BAM file processing) as two FASTQ files are used as input for each process. In the final step, consisting of variant call and VCF file filtering only single process may be created as all BAM files are taken as an input and a single VCF file is created.

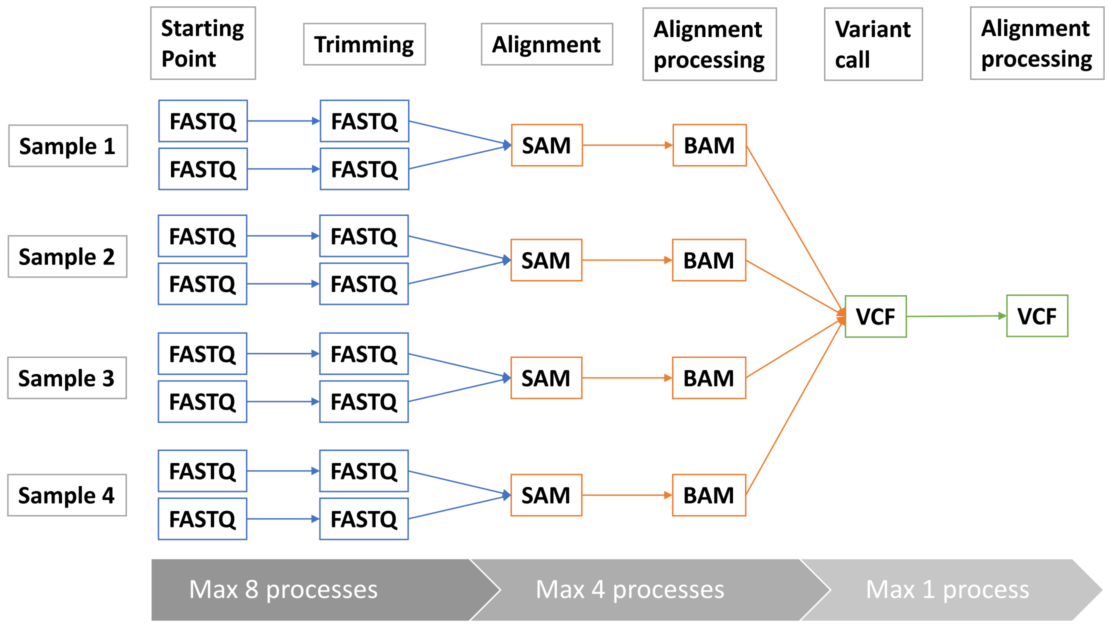

Figure 1. Example workflow from paired-end read trimming to variant call for 4 samples.

If you choose to generate this workflow as a UNIX background job, STAPLER creates a single continuous workflow and ensures that an optimal number of processes is always created. However, this is not the case if the workflow is parallelized by using a resource manager due to the nature of how array jobs / multiple job runs work. In addition, resource managers require that each submitted job defines the maximum run time and the number of CPUs and the amount of RAM to allocate. As these parameters often vary quite a bit during the workflow and users are usually billed based on the resource usage, it is not sensible to run the whole workflow using the same parameters. Therefore STAPLER provides two parallelization priority modes two choose from: **continuous mode** and **split mode**.

In the split mode STAPLER will split the workflow in cases where multiple jobs would need to coalesce. In the case of the above workflow example this would create three-part workflow consisting of 1) FASTQ trimming 2) alignment and alignment processing 3) variant call and variant filtering. The advantage of this mode is speed, as the maximum number of simultaneous jobs are always used. The drawback is that each workflow must be manually submitted when the previous workflow has finished. STAPLER does not currently provide any watchdog functionality which would do this automatically, as such very long running tasks are often not allowed on login nodes of supercomputing environments.

In the continuous mode STAPLER prioritizes the continuous running of the workflow so that only a single workflow is created, but jobs that will eventually coalesce are run within the same process. The advantage of this mode is that user does not need to manually submit several jobs to finish the whole workflow. The drawback is that any workflow involving a coalescence event will not spawn the optimal number of processes and will therefore take longer time to finish. In the case of the above workflow example, a single continuous workflow would result, but only one process would be created, as all jobs will coalesce into one in the variant call stage. It is possible to add the special **SPLIT** command into staplefile to create a manual split point. For instance, a split command could be added before variant call step to split the workflow into two parts. The first part would then include the trimming, alignment, and alignment processing steps divided to four processes, and the second part would include variant call and variant filtering steps within single process.

It is possible to **set an upper limit for the number of processes** STAPLER will create by including the --MAX_JOB_COUNT parameter to STAPLER command line. By default, STAPLER will generate as many processes as each workflow allows, which often depends on the number of input files (as explained above). Most supercomputing environments impose a limit for the size of array jobs / multiple job runs, which cannot be exceeded. When executing workflows as UNIX background tasks the number of computing cores and the amount of available RAM on the current platform will limit the number of simultaneous tasks one can run. Notice though, that many programs themselves can take advantage of multiple cores. Therefore, if your workflow includes a step where you have set a program to use four cores and STAPLER creates four processes for the workflow to be run in parallel, 4 * 4 = 16 cores will be used simultaneously.

5 Troubleshooting:

Getting odd errors or crashes? First thing to check is that you are using an up to date Python 2 version, meaning 2.7.0 or later! Also notice that Python 3 does not suffice.

Getting error message saying that a parameter -X is not supported by a tool, although it is clearly stated in the original manual of the tool that -X is supported? The STAPLER may not support all features of all software or the syntax may differ (e.g. sometimes only the long or short version of an option is supported: --parameter_X instead of -X). Check how a tool works before using it with STAPLER: *python STAPLER.py --help <tool_name>*

STAPLER does some basic checks on user input. If a problem occurs, it (hopefully) prints out an error message giving a good enough description of the problem for you to solve. Often more detailed information on the error may be found from the log file, so check it out!

Sometimes user errors might get through the validation, in which case the program may crash without giving out any usable info on what is wrong. In these cases it is a good idea to check if the log file would contain useful information. At least you can find out the command line that caused the script to crash and try to find out if there is something wrong with it.

Bugs and improvement suggestions can be reported to jaakko.tyrmi@gmail.com.