STAPLER

Simple and Swift Bioinformatics Pipeline Maker

Manual for version 17.03.15

Jaakko Tyrmi

jaakko.tyrmi@gmail.com

1 Introduction

The intended purpose of this software is to allow you to create bioinformatics pipelines in a quick and easy way. STAPLER may be the right pipeline tool for any (or any combination) 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 are using SLURM resource manager for parallelization of bioinformatics tasks.

As often is the case with pipeline software, the most obvious limitation of STAPLER is that it supports only selected software, and the support may be limited to certain subset of tools and versions. However, it is possible to circumvent this problem to a degree by taking advantage of the built in flexibility of STAPLER.

The documentation for STAPLER is divided into three places:

1. This manual for general description of the tool.
2. Contents of the *example_dir* for details of various input/output files.
3. Built in help function of STAPLER for usage details for each bioinformatics tool (type on command line: *python STAPLER.py --help*).

2 Requirements

The only requirement for running STAPLER is Python v. 2.7.0 or later Python 2 version. You also need to install all the supported programs that you intend to use, since none of these are included in STAPLER. STAPLER has been tested on Linux operating system but it should also work on other Unix-based or Unix-like operating systems (e.g. OS X), although this has not been tested. SLURM resource manager must also be installed, if you wish to use the parallelization features of STAPLER.

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 easy even when you are just a client of a third party computer cluster or supercomputer.

3 How to use STAPLER

3.1 Quick steps

1. Download STAPLER and configure it for your platform by editing the *installation_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. Create your first workflow by creating a new *staplefile* by editing the example template provided
4. Create a simple shell script of this workflow by typing
   python STAPLER.py path/to/my_staplefile.txt
5. Run this job by typing
   source path/to/my_output_shell_script.sh

OR alternatively:

4. Create an array job of this workflow by typing
   python STAPLER.py path/to/my_staplefile.txt --SLURM
5. Submit this job to SLURM resource manager by typing
   sbatch path/to/STAPLER_SBATCHABLE.sh
6. Check if the run has been successful by typing
   python STAPLER.py path/to/my_staplefile.txt --CHECK
7. Compress your workflow results to save space by typing
   python STAPLER.py path/to/my_staplefile.txt –COMPRESS
   sbatch path/to/COMPRESS_SBATCHABLE.sh

Read more detailed instructions for each of these steps below, from example files and from built in help (*python STAPLER.py --help*).

3.2 Installing

Before you can start using STAPLER you will need to modify the "installation_config.txt". The purpose of this file is to tell STAPLER how to run the supported software on the particular platform you are using. This 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 **prefix** column: Tells STAPLER the path to the executable of the supported software on your platform. For instance, if the jar file for picard's CompareSAMs tool is found in "/appl/bio/picard/picard-tools-1.113/CompareSAMs.jar" you should add this path to your installation_config.txt. As some tools require another program to run the executable (for instance, CompareSAMs.jar requires java), you should include this program call on this column also. Therefore the whole value in this column in this case would be: "java -jar /appl/bio/picard/picard-tools-1.113/CompareSAMs.jar". If the tool is included in your PATH, you should use the keyword "none" here.

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 for 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.

3.3 Creating a pipeline

Hint: See the contents of "example_dir" to better understand this part of the manual!

When STAPLER is successfully installed you need three things to run it:

1. **Project directory** for you project files/directories (e.g. the *example_dir*), containing
2. **Input directory** containing **input data files** for your workflow (e.g. *my_raw_data*)
3. A text document ("**staplefile**" hereafter) containing a list of software that you wish to apply on your input data (e.g. *staplefile_example.txt*)

3.3.1 The project directory

The project directory is any directory that contains an **input directory** for STAPLER. Output directories are created by STAPLER to this directory. After creating a couple workflows to this directory with STAPLER there will be quite a few directories in it, so it may be wise to have a consistent scheme in naming the staplefiles and the actual jobs.

3.3.2 The input directory

The input directory containing input files has to be created manually by the user into the project directory. The input directory can contain, say, fastq files you wish to filter before mapping them to reference genome.

3.3.3 The staplefile: modeling a workflow

The general concept and usage of staplefile is presented here, for more details open the *staplefile_example.txt*.

In short, a staplefile defines the sequence of tools (and their parameters) that you wish to use in a workflow. In the case of example file, a list of read quality control software with suitable parameters. 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.

STAPLER is run on command line with a path to a **staplefile** as a parameter (see examples further below). STAPLER will first create an output directory for the shell script/SLURM batch script that is created as a result. This directory has the following format

.../STAPLER_<my_job_name>_BATCH_SCRIPTS. This directory will contain either

1. shell script that the user can then run (e.g. "source <file_name>") or
2. SLURM batch script ("sbatch <file_name>”) if the --SLURM option was included in the command line for STAPLER.

If the --SLURM option is used the BATCH_SCRIPTS directory will contain several files, but only the file called *STAPLER_SBATCHABLE_<timestamp>* can be input for sbatch command. Parallelization of workflows with SLURM is discussed in more depth later in this manual.

STAPLER reads the sequence of tools listed by user in the staplefile one by one. For each listed tool a new directory is created into the **project directory** (e.g. *.../example_dir/STAPLER_ExampleJob01_0_fastx_quality_stats* etc.). At this point the directories are empty, as the shell script/SLURM batch script has not yet been run. When user runs the output script of STAPLER, the output files will appear to these directories. This set of directories are referred as the **directory stack**. First directory of the stack is the input directory explicitly defined by the user in the staplefile (e.g. *my_raw_data*). This directory is used as the input directory for the first tool listed in a staplefile (e.g. *my_raw_data* contains input for fastx_quality_stats tool). This directory is then assumed to be the input for the second command and so forth.

However, if at some point the previous directory does not input files in a proper format, STAPLER will then go back in the directory stack trying to find the latest directory which contains files in correct file format for the tool in question.

When listing commands to staplefile it is important keep in mind that STAPLER may use different command line syntax than what is shown the original tool's manual. Also, STAPLER may not support all options or features of all the supported tools. Therefore, when adding a new tool to staplefile one should always check how to use the particular tool in question by checking the internal documentation of STAPLER: *python STAPLER.py --help <tool_name>*

3.3.4 Workflow and branching logic further explained

For example, let us consider the simple workflow in *example_dir*:

In this example, the user has some fastq files in *my_raw_data* directory. Therefore my_raw_data is the **input directory** and it's parent directory, *example_dir*, is the **project directory**. The user has created a **staplefile** *staplefile_example.txt* that defines a workflow for first examining the initial quality of this data and then uses tools of fastx-toolkit to filter the data (Fig 1.).

As the user types the command
(*python STAPLER.py .../example_dir/staplefile_example.txt –SLURM*)

STAPLER will:

1. Create a shell script directory containing shell scripts and possible error files created by the software tools (<job_name>_BATCH_SCRIPTS)
2. Create output directories for each user defined software tool (STAPLER_<job_name>_<tool_name>)
3. Deduce the input directory for each software listed by the user (For instance in Fig 1, fastx_trimmer can not take boxplots created by fastq_quality_boxplot_graph as an input. STAPLER will then look for usable files (.fastq) from previous directories of the **directory stack**. In this case, suitable input files were found from my_raw_data directory.)

This behavior allows for user workflow to include some simple brancing events. Spawning of multiple alternative workflow branches is not supported.

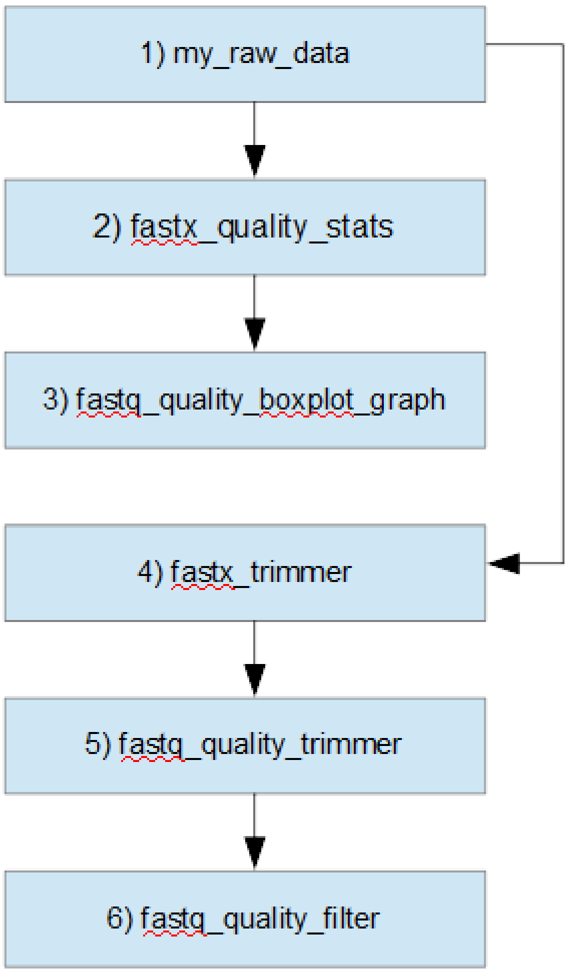

Figure 1. A workflow of MasterFile_example.txt

3.3.5 File naming explained

Internally STAPLER assigns a **basename** for each input file. The basename is a file name stripped of file path and suffix. This basename is then used as a starting point when STAPLER creates the absolute path for the file. For instance, the base name of *example_dir/my_raw_data/sample_01_R1.fastq* would be *sample_01_R1*. In cases where several input files are taken and one output file is produced the output file name will be one of the input file base names with appropriate path and suffix added. For instance mapping files *.../sample_01_R1.fastq* and *.../sample_01_R2.fastq* to a reference genome may output a file called *.../ sample_01_R1.sam*. This behavior is used internally by STAPLER to keep track of input files to make sure that when parallelizing work flows each input file is handled in the same thread.

3.4 Running STAPLER (with examples)

The STAPLER is run from command line. After STAPLER is run, go to the output directory to run the shell scripts/SLURM batch script. Take the following into account when running STAPLER:

- You must be in the directory that contains STAPLER.py when running STAPLER, otherwise an error message about missing installation_config.txt file is shown!
- Before running a shell script (created without --SLURM parameter) you may want to add a shebang line to the shell script pointing to the command line interpreter you wish to use to process the shell script. E.g. *#!/bin/bash* if you wish to use bash.
- If you run the SLURM batch script by using its absolute path, the SLURM stdout and stderr files are created into your current directory! Therefore it is recommended to be in the output directory when running the SLURM batch script.
- Often it is most convenient to open two terminal windows, with the first terminal window showing the STAPLER installation directory and the second window showing the **project directory** that you are working with. staplefile path is first copied from the second terminal window to the first window to run STAPLER. STAPLER will then print the path to the newly created batch script directory, which can then be opened in the second window to run/submit the job.

Print general help of STAPLER:
python STAPLER.py --help

Print specific help of a STAPLER tool:
python STAPLER.py --help <tool_name>

Run STAPLER with a staplefile:
python STAPLER.py <path/to/staplefile.txt>

Run STAPLER with a staplefile with parallelization:
python STAPLER.py <path/to/staplefile.txt> --SLURM

Check SLURM generated .out and .err files for any errors/problems after job has been submitted and finished:
python STAPLER.py <path/to/staplefile.txt> --CHECK

Compress the result files of a run (creates a simple shell script to run):
python STAPLER.py <path/to/staplefile.txt> --COMPRESS

Decompress the result files of a run with parallelization (creates a SLURM input file):
python STAPLER.py <path/to/staplefile.txt> --DECOMPRESS --SLURM <core_count>

4 Parallelizing a workflow with SLURM

4.1 Theory of parallelization

In order to understand how STAPLER parallelizes workflows it is necessary to understand the basics of parallelization. For this, please refer to user documentation of the super computing provider you are a client of. In the documentation, you should find information on how to interact with the popular resource manager SLURM (which must be installed in order to use parallelization features of STAPLER). STAPLER enables parallelization by taking advantage of the "job array" feature of SLURM. It may be a good idea to familiarize yourself with array jobs by reading the cluster/supercomputer specific documentation or the following SLURM documentation:

http://slurm.schedmd.com/job_array.html

4.2 Parallelization in STAPLER

The "--SLURM" parameter can be used in command line to create a SLURM **batch script file**, which can then be submitted with sbatch command to SLURM resource manager. This file can be regognized from the format *STAPLER_SBATCHABLE_<timestamp>.sh.* The batch script file defines necessary parameters for an **array job** for SLURM. In addition to the batch script file the output directory contains other shell scripts. By default each of these shell scripts contain all the commands concerning a single input file (notice that this default behavior can be changed by using project files, see the end of this chapter). Therefore you will get one such shell script file for every input file you have, listing all the commands you defined in your staplefile. These files should not be run manually, as they are called by the SLURM batch script file when you submit it. Each of these shell script files are run as a new process on the supercomputer. Therefore, fifty input files results in single batch script file and fifty shell scripts. Command *sbatch STAPLER_SBATCHABLE_<timestamp>.sh* will then result in fifty processes running on the supercomputer.

There is an important caveat to keep in mind: By default, this program creates an array job in a way that all input files have their own shell script, in which each command you have listed in the staplefile is applied to this input file. When you submit the "slurm shell script file", these separate shell scripts are then executed in parallel. This works fine as long all the commands you have specified require the same number of input files, but if this number changes, things will not work as expected. For instance, let us examine the following workflow:

You have paired end Illumina raw reads from two sequenced individuals:

- sample_1_R1.fastq & sample_1_R2.fastq
- sample_2_R1.fastq & sample_2_R2.fastq

Using the FASTX-tookit quality_trimmer command in staplefile for these input files would create four array jobs, one for each file. Now imagine that the next step would be to use MosaikBuildFastq command, that takes both reads of single sample as an input.

At this point a problem arises: When the fastest of the four processes has finished executing the quality_trimmer command, it will then start MosaikBuildFastq. However, the pair of this read file will not be ready, since FASTX-toolkit quality_trimmer is still working on it! This would cause an error and the rest of the specified commands would be skipped. In practice you have two options for such workflows:

1. Use a project file (see PROJECT.txt for an example) to define sets of files that should be kept in

the same thread of execution (i.e. shell script). In this particular case you should have two defined sets: sample_1 and sample_2. This would result in two instead of four processes, hence solving our synchronization issue. Now all files originating from the same individuals will be processed in the same process! Notice that when using a project file fewer number of processes are created, which leads to increased run time.

2. Do not mix commands requiring different number of input files into one run, but split the workflow into separate steps instead.

5 Example workflows

1) SNP calling pipeline

The STAPLER supports many useful scripts for working with next generation data: fastx_toolkit, parts of BWA and Mosaik aligners, many picard-toolkit features, GATK SNP caller, many custom scripts etc. These tools can be used in sequence to create an alignment pipeline if one has raw reads from next generation sequencing.

The *staplefile_example.txt* is an example of such pipeline. Notice that you will need to modify it to include correct parameters and file paths to work on your platform!

2) Using STAPLER for non-supported software

STAPLER contains a tool called "ANY_TOOL", which may be useful for using tools not supported by STAPLER. See the tool specific help in STAPLER for more details. Lets pretend for a moment that STAPLER would not support fastq_quality_filter of fastx_toolkit. We could use this tool with ANY_TOOL with the following command line:

ANY_TOOL --!TOOL_NAME path/fastq_quality_filter --!INPUT_TYPE .fastq -q 20 -p 50 -Q 33

Next, lets use ANY_TOOL for imitating fastq_quality_boxplot_graph

ANY_TOOL --!TOOL_NAME path/ fastq_quality_boxplot_graph.sh --!INPUT_TYPE fastq --!OUTPUT_TYPE png -q 20 -p 50 -Q 33

Notice that you will have to replace ANY_TOOL in the output file to the actual tool name before running the output file!

6 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 installation_config.txt file is missing? You should be located in the same directory as STAPLER.py when running it, otherwise this error is shown.

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 etc. can be reported to jaakko.tyrmi@gmail.com.