OVarFlow

Release 2.0

Jochen Bathke

Dec 25, 2021

OVARFLOW USAGE

OVarFlow is an **open source workflow for variant discovery** of SNVs (single nucleotide variants) and indels (insertions and deletions). With today's high-throughput sequencing technologies and continuously declining sequencing costs, variant discovery in whole-genome resequencing data is not only more affordable but also more demanded than ever. Hence the need for easy and reliable variant calling emerges in a broader audience. Consequently OVarFlow was created with the **three major goals** of:

- **automation**,
- **documentation** and
- **reproducibility**.

To achieve those goals OVarFlow is build upon several technologies that are proven and widely used in bioinformatics, being Snakemake as a workflow management system, Conda as an environment manager and software repository and GATK as a variant discovery toolkit.

CHAPTER

ONE

TARGET AUDIENCE

Variant calling is no novel task. Especially GATK not only provides the tools for variant discovery but also its well known Best Practices Workflows. A downside of those guidelines is their focus on human sequencing data, being the probably best studied model organism. With less well studied organisms workflows often have to diverge significantly. OVarFlow steps into this gap and provides **variant discovery also for non-model organisms** in the fields of:

- **biological basic research**,
- **animal breeding** and
- **plant breeding**.

For the latter, only haploid organisms have been tested. Tetraploid organisms might require adaptation of the workflow (especially GATK HaplotypeCaller).

CHAPTER

TWO

PREMISES TO USE OVARFLOW

To be able to use OVarFlow some requirements must be fulfilled in the first place. Obviously whole-genome resequencing data of the respective organism must be given. Like GATK Best Practices Workflows OVarFlow is designed to be used with **Illumina short read sequencing data**. Furthermore a **reference genome sequence** must be given and also a **reference annotation**, in case that functional annotation is desired.

To be able to analyze those data, some more technical requirements must also be considered. First of all access to a **Linux based computing infrastructure** of sufficient size must be given. What is sufficient of course depends upon the size of your data set. Some hints are given within the "Resource requirements" and "Hardware recommendations" sections.

Finally on the side of **human resources** prior knowledge of the **Unix/Linux command line** with some proficiency is required. Apart from this no prior knowledge (albeit helpful) is expected in any of the used technologies (Snakemake, Conda or GATK). This documentation tries to be as comprehensive as possible, introducing the technologies as needed and linking to further resources to get you started. For users with prior experience in the mentioned technologies the "Quick reference for OVarFlow" might already be everything you need to get started. Those users might be able to **setup variant calling in 30 minutes**. The rest is handled by OVarFlow.

CHAPTER

THREE

MOTIVATION BEHIND OVARFLOW

Some of the motivation behind the creation of OVarFlow should already be obvious from the previous paragraphs. But still some might wonder why all the hassle when GATK Best Practices are not only a detailed description, but are also commonly referred to in method sections of various papers (e.g. PMID 29395925, PMID 24824529, PMID 30952207). In the end many method sections mentioning GATK are rather superficial (e.g. PMID: 31246983, PMID: 31900978), sometimes not even mentioning the names of GATK subtools used in the analysis. This was also noted by the initiators of GATK, therefore writing:

> *6. What is not GATK Best Practices?*
>
> *Lots of workflows that people call GATK Best Practices diverge significantly from our recommendations. [. . .] However, any workflow that has been significantly adapted or customized, whether for performance reasons or to fit a use case that we do not explicitly cover, should not be called "GATK Best Practices", which is a term that carries specific meaning.*

Source: About the GATK Best Practices (date of accession: May 7th 2020).

Another problem is, that GATK Best Practices evolve over time, ultimately rendering global references to them (like http://www.broadinstitute.org/gatk/guide/best-practices) useless. Thereby reproducibility of the exact data evaluation workflow is lost. Irreproducible research even lead to the coining of the phrase replication crisis which is an ongoing problem in science. A problem that even major science publishers like nature (Special: Challenges in irreproducible research - 2018) are more and more aware off.

Therefore the main motivation behind OVarFlow is to achieve exact documentation and reproducibility of data evaluation. It is the kind of openness that science should offer!

OVarFlow achieves this goal by four key points:

- the OVarFlow Snakefile and workflow itself,
- the documentation of Conda environments in a yml file,
- documentation of the analyzed dataset in a CSV file and
- documentation of non-default workflow settings in a yml file.

This results in a maximum of documentation and reproducibility of the data analysis and in addition eases writing of any methods section, by providing those four files. Also users of OVarFlow are encouraged not only to use OVarFlow but also to adopt it to their specific needs and then to republish their modified workflow.

With that being said, good luck with your variant discovery project and the hope that the following documentation will turn out to be useful in your work!

3.1 Capabilities of OVarFlow

The complexity of variant calling with all its distinct data evaluation steps can be a daunting task. OVarFlow tries to wrap as much of this complexity as possible, thereby automating this intricate process to a maximum degree. Especially the usage of GATK with its hundreds of single tools is challenging to novice users. But not only the amount of tools is challenging also their individual usage with some very subtle obstacles, e.g.:

- High peak loads caused by the Java garbage collection in dependence on the number of available cores of the CPU.
- Considerable extended computation times depending on the given instruction set of the CPU.

All those complexities have been taken into consideration and were incorporated into OVarFlow. But not only the intrinsic complexity of variant calling and GATK is encapsulated by OVarFlow. Furthermore OVarFlow was extended to include features that GATK does not possess directly.

3.1.1 Some highlights of OVarFlow

Massive parallelization Not only a high degree of parallelization, but also the ability to fine-tune the desired degree of parallelization. Parallelization of GATK HaplotypeCaller version 3 was abandoned with the newer GATK 4 version, only leaving Apache Spark as an option. With OVarFlow GATK 4 HaplotypeCaller can operated in parallel on various genomic intervals thereby accelerating the most time consuming step of variant discovery.

Inclusion of already available variant calls Previously generated variant calls (vcf files) can easily be incorporated into new data evaluations. This allows for easy incorporation of new individuals into running studies, without the need to recalculate all samples.

Exclusion of small genomic contigs Many genomes contain small contigs of e.g. 1000 bp or even less. Often those tiny contigs are of no further interest. OVarFlow lets the user decide whether to include those tiny contigs into the analysis or not. Furthermore the threshold of contig sizes to exclude can be chosen by the user.

Functional variant annotation Also functional annotation of the detected variant is automatized by incorporating the annotation program SnpEff into the workflow.

Easy application installation Variant calling depends upon a fast software set. By the use of Conda environments, installation of all needed applications is basically scaled down to a single command. Alternatively a single, pre-built Docker container already bundles all the required software packages.

3.1.2 The two phases of OVarFlow

OVarFlow is a variant calling workflow, that posesses two separate phases.

The basic variant calling workflow First workflow is mandatory. It is designed to be as basic as possible and a the briefest way to deliver annotated variants. Therefore minimum prior knowledge is required. Only a reference genome and annotation is required.

The extended BQSR workflow The second workflow is optional and builds upon the basic variant calling workflow. It uses previously called variants to perform base quality score recalibration (BQSR) and further improve the variant calling results.

3.1.3 The primary goal of OVarFlow

Finally the main goal of OVarFlow is documentation and reproducibility of variant calling, which is achieved by three components:

- OVarFlow as a workflow itself.
- Easy documentation of the used program versions via Conda environments and yml files.
- A CSV file to document the respective variant calling and all the input data used in it.

3.2 Quick reference for OVarFlow

The extensive documentation of OVarFlow might seem daunting, illustrating the complexity of variant calling. Besides the inherent complexity of the task, the documentation tries to be as comprehensive as possible to assist novice users. On the other hand advanced users that already have a working Conda environment can set up the variant calling workflow in probably less than half an hour. A task that might take days to weeks is then automated by OVarFlow. This quick reference is for those advanced users that want to quickly setup a new project.

1. Create a project directory (`project_dir`):

```
mkdir -p /path/to/project_dir/
```

2. Create a Conda environment (`conda_env`) for your project (or use one that already available for variant calling) and activate this environment:

```
conda create --prefix /path/to/project_dir/conda_env
conda env update --prefix /path/to/project_dir/conda_env \
                 --file OVarFlow_dependencies_mini.yml
conda activate /path/to/project_dir/conda_env
```

3. You need to create a directory structure and put some files from OVarFlow's GitLab repository into place:

```
/path/to/project_dir/
/path/to/project_dir/conda_env/
/path/to/project_dir/variant_calling/
/path/to/project_dir/variant_calling/FASTQ_INPUT_DIR/
/path/to/project_dir/variant_calling/REFERENCE_INPUT_DIR/
/path/to/project_dir/variant_calling/OLD_GVCF_FILES/
/path/to/project_dir/variant_calling/Snakefile
/path/to/project_dir/variant_calling/scripts/average_coverage.awk
/path/to/project_dir/variant_calling/scripts/createIntervalLists.py
/path/to/project_dir/variant_calling/samples_and_read_groups.csv
/path/to/project_dir/variant_calling/config.yaml # optionally
```

   Some of the files can be created through the OVarFlow Snakefile, to avoid typos:

```
cd /path/to/project_dir/variant_calling/
snakemake -np
```

4. Place your reference and sequencing files into the appropriate directories.
5. The configuration file `samples_and_read_groups.csv` has to be adopted for your specific project. Modify that file accordingly. It will also serve as a reference for your settings.

6. An additional optional configuration file `config.yaml` allows for fine-tuning of Java resource usage and defining the degree of parallelization of the data evaluation.
7. It is optional but advisable to test whether the annotation can be processed by `snpEff` at first, preventing late stage failure.

```
snakemake -p --cores <number_of_desired_threads> create_snpEff_db
```

8. You can start the variant calling now:

```
cd /path/to/project_dir/variant_calling/
snakemake -p --cores <number_of_desired_threads>
```

That's already everything to start your variant calling. Depending of the size of your data set and available computing resources, OVarFlow will take care of the rest of the process that might take even weeks, while you can continue working on other projects.

Finally you might want to document the exact software versions, that were used in the data evaluation. Just extract that information from your Conda environment:

```
conda activate /path/to/project_dir/conda_env
conda env export > conda_environment.yml
```

3.2.1 Adding the BQSR workflow

The above workflow will already result in a set of annotated variants that can be sufficient for further analysis. To further refine the called variants, the GATK team recommends to perform base quality score recalibration (BQSR). Therefore BQSR was implemented in a second workflow, that can optionally be run in succession of the first workflow, to further improve the called variants through BQSR.

1. The BQSR workflow has to be run within the same directory where the previous workflow was executed. So `cd` into the project directory first:

```
cd /path/to/project_dir/
```

2. Two files have to be copied from the GitLab repository:

```
/path/to/project_dir/variant_calling/SnakefileBQSR
/path/to/project_dir/variant_calling/configBQSR.yaml (optionally)
```

3. The conda environment that was previously used has to be activated again:

```
conda activate /path/to/project_dir/conda_env
```

4. The input data is automatically detected from the file structure generated in the previous workflow. This includes the following directories and files, that still have to be present:

```
03_mark_duplicates/<file_names>.bam
11_filtered_removed_VCF/variants_filtered.vcf.gz
processed_reference/<file_name>.fa.gz
snpEffDB/<directory_name>/<genes.gff, sequences.fa.gz, snpEffectPredictor.bin>
```

   A configuration file like previously `samples_and_read_groups.csv` is therefore neither needed nor used.

5. Fine-tuning of the workflows performance is enabled through the configuration file `configBQSR.yaml`. This file is mainly about Java heap size and garbage collection threads, that can be optimized for a given computing environment.
6. The BQSR workflow can now be started like this:

```
cd /path/to/project_dir/variant_calling/
snakemake -p --cores <number_of_desired_threads> -s SnakefileBQSR
```

3.2.2 Warning

Not every version of Snakemake works with OVarFlow. The workflow makes use of so called *checkpoints*. Due to a bug that was introduced in Snakemake versions higher than 5.26.1 checkpoints don't work anymore. This bug was fixed in Snakemake 5.31.0. Therefore explicit software version were defined in `OVarFlow_dependencies_mini.yml`. In cases were it is desired, the most current software version can be obtained using the file `OVarFlow_dependencies_mini_unversioned.yml`.

3.3 A Primer into the technologies

For novice users, with only a basic understanding of bioinformatics, this section is supposed to serve as a very brief introduction into the technologies that are used in OVarFlow. In this sense the following paragraphs are more of a technical reference. As OVarFlow makes heavy use of the technologies outlined below, this will help novice users with the terminology used in further sections.

3.3.1 Python 3

OVarFlow makes is build using the Python 3 programming language. Python is a scripting language, meaning that the source code of the respective program is interpreted by a special run-time environment which doesn't need to be compiled first. To be able to execute a Python program the Python interpreter (usually CPython - the reference implementation of the interpreter) as well as its accompanying modules have to be installed.

3.3.2 Snakemake

Snakemake is a workflow management system (WFMS). WFMS allow for the creation of defined sequences of tasks a computer can execute. Practically this means the automatization of the successive or even parallel execution of various programs. Thereby the main usage of Snakemake is the creation of automatic, reproducible data analysis workflows.

Snakemake itself is written in the Python programming language and the Python syntax, with some specific extensions, is used to write workflows in Snakemake. Therefore a basic understanding of Python is required to create a workflow using Snakemake. The file containing the workflow is called a Snakefile.

3.3.3 Conda & Bioconda

Many bioinformatics tasks revolve about the well orchestrated execution of a plethora of command-line tools. WFMS like Snakemake can automate such processes. On the other hand the individual software has to be obtained and installed on the executing system. Package management system simplify this task tremendously, often removing the need to compile software packages from source code.

Conda is an open source package management system, that find broad application in data sciences. Beyond that it is also an environment management system, that allows for the independent installation of several versions of a software, without causing dependency conflicts. Even though some bioinformatics tools can be found via Conda it is not specialized in bioinformatics use cases. This cap is bridged by Bioconda, which is a so called channel for Conda, targeted a bioinformatics tools. Basically the amount of available tools is increased by adding the Bioconda channel to the Conda package manager.

3.3.4 GATK & GATK Best Practices

GATK is the commonly used abbreviation for the Genome Analysis Toolkit. This collection of command-line tools is focused on the identification of genomics variants in high throughput sequencing data. It bundles more than 200 individual command-line tools. GATK is actively developed at the Broad Institute with the current major version being GATK 4 (in 2020). As opposed to previous versions GATK 4 is open-source and freely available under a BSD 3-clause license.

Some common use cases of GATK are described within the so-called "GATK Best Practices". Those descriptions try to give an overview of tasks and workflows that are widespread in variant calling. As the tools are further developed, the "GATK Best Practices" are also subject to modifications.

3.3.5 Docker & Singularity

Like OVarFlow many modern software products are composed of many individual pieces, whose well orchestrated interaction is mandatory for the final product to work. Deployment of such complex software applications can be complex. OS-level virtualization, also referred to as container virtualization, tries to simplify software deployment by bundling all individual components of an application in a single container. Container virtualization finds wide usage among software developers and server administrators but is less focused on end users. Most container virtualization techniques have been designed around Linux and are supposed to be used with this operating system (still ports to Windows and macOS do exist).

Docker is probably the most well known container technology. Despite its wide use, it has some considerable drawbacks, when used in multiuser computer environments. Most problematic is the fact, that Docker containers need to be executed with system administrator privileges. This renders Docker unusable in many multiuser computer environments. This drawback of Docker is circumvented by Singularity. Singularity, just like Docker, packages individual software components into a single container. Furthermore it can make use of Docker containers, but does not require system administrator privileges. Therefore Singularity finds broad use in academic high performance computing, where multiple users need to access a single system.

3.4 Setup & preparations

OVarFlow has been designed to be used in two alternative ways:

- directly, executing a so-called Snakefile containing the workflow or
- using a pre-built Docker-Container containing the Snakefile as well as all executables and their dependencies.

Both solutions have their own strengths and weaknesses. Direct usage of the Snakefile will give you more control over the workflow. This includes the opportunity to easily update the individual programs used in the workflow. But changes are not limited to the programs used, you might also incorporate your own personal changes to the Snakefile, thereby altering the data evaluation procedure. Of course those options require at least a basic understanding of the Python 3 programming language and the Snakemake syntax. Docker on the other side will hide some of the complexity of OVarFlow, including installation of additional programs, but it will also limit the end user to the program versions bundled within the docker container. Ultimately the Docker container only encapsulates the Snakefile and applications, which are utilized in OVarFlow. To sum things up:

- Snakemake & Conda will allow for more control and easy updating of applications.
- Docker on the other hand needs fewer configuration but gives the user less control.

Still both options are designed to be used under a Linux based operating system and have not been tested on other platforms. Anyway variant calling is a computationally demanding task which consumer hardware is badly suited for. Therefore high performance computing (HPC), which is vastly dominated by Linux, is required.

The following paragraphs are directed to novice users, with no prior experience in the usage of Conda & Snakemake or Docker. The descriptions are intended to create a basic setup and refer to broader documentation of the respective software.

3.4.1 Setting up a Conda environment

Conda is a package and environment manager. It allows for the installation of various software comparable to an appstore. Also different versions of a single software can be installed that are totally independent of one another.

Different distributions of Conda are available, namely Anaconda and Miniconda. The basic functionality of both distributions is identical, but Anaconda is meant to provide a full grown application suit for data science using Python and R. In doing so Anaconda will install a plethora of software that is commonly used in the field. Most of this software is not needed for the usage of OVarFlow. Hence the installation of Miniconda is recommended. This minimum installer for Conda still allows for the manual installation of every software that comes bundled with Anaconda in case it should be needed at a later time.

- Download the Python 3 installer for Linux in the 64-bit version. 32-bit computers would be overwhelmed with variant calling anyway.
- Verification is optional but highly recommended (`sha256sum Miniconda3-latest-Linux-x86_64.sh`).
- A detailed description of the installation is available, but essentially comes down to a single command-line:

```
bash Miniconda3-latest-Linux-x86_64.sh
```

  The installer will prompt some questions. Novice users can accept the defaults.

- After closing and reopening the shell, the Conda command should now be available. This can easily be tested by running the command `conda help`.

Now that Conda is installed, additional software resources - channels in Conda terminology - have to be made available.

- List the currently available channels via `conda info`.
- Conda-forge and Bioconda need to be added:

```
conda config --add channels defaults
conda config --add channels bioconda
conda config --add channels conda-forge
```

- That the available channels have indeed been altered can be verified again by `conda info`.

Your Conda installation is now ready to be used with OVarFlow. It will enable you to obtain all of the software that is used by OVarFlow. Installation of software dependencies and further usage with OVarFlow is covered in the Conda & Snakemake usage section.

3.4.2 Setting up Docker or Singularity

Alternatively to the above Conda usage container virtualization can be employed. This technology has the advantage of bundling an application and its dependencies. In this case no Conda installation is required, as all indispensable software components are included in the container. On the other hand the software for container virtualization itself has to be present on the system. Also a certain understanding of container technologies is mandatory to be used efficiently. Docker and Singularity are two widely used, compatible container technologies.

Docker

Docker provides a comprehensive documentation but the docker Docker curriculum might be better suited for novice users. Dockers biggest drawback is probably its need for root access to the respective computer. If that's a hindrance Singularity might be an alternative.

The company behind Docker provides `.deb` and `.rpm` packages for various Linux distributions. As Docker is written in the Go programming languages, statically link binaries are available as well.

The Docker installation can easily be tested:

```
sudo docker run hello-world
```

By adding your user to the group docker the need to include `sudo` with every docker command is circumvented.

However usage of Docker is far from self explanatory and a basic understanding of OS-level virtualization with the concept of images and containers should be given. Briefly, images are the blueprint of a container. The image itself is immutable and contains all the code of an application. A container is a running instance of the image. The application is then executed from the container. When used without caution a new container is created every time Docker is started. (For programmers: its a bit like the concept of class and object.)

An overview of the basic docker commands is available:

```
docker --help
```

The most basic docker usage shall be shown with the example image `godlovedc/lolcow`. This image can be obtained via:

```
docker pull godlovedc/lolcow
```

This image should now be listed in the locally available images:

```
docker images
```

You can create and run a new container from the image:

```
docker run godlovedc/lolcow
```

All containers available on the system can be listed via:

```
docker ps -a
```

Also a second container can be created from the image, by executing `docker run godlovedc/lolcow` a second time. Now `docker ps -a` will list two containers that where created from the image `godlovedc/lolcow`.

A given container can also be used again. Its name can be obtained first via `docker ps -a`.

```
docker start -i <container_code_name>
```

Singularity

Singularity offers an equally comprehensive documentation. Especially the quick start section is worth having a look at. A detailed description of the installation process as well as an introduction into the usage of the `singularity` command is given.

OVarFlow does not provide a dedicated Singularity image. But Docker images can be used with Singularity as well. An usage example of the `lolcow` image is also included:

```
singularity pull docker://godlovdc/lolcow
```

Further details can be found in the linked documentation.

Finally it should be noted, that the links provided point at the documentation of the version 3.5, which is current at the time of writing. By changing the version number in the provided links you can also obtain documentation for different versions of Singularity.

3.5 Conda & Snakemake usage

Setting up a Conda environment and manually executing the Snakefile is the recommended way to use OVarFlow. This method will allow you to have the maximum control of the whole process. Also, if desired, you can determine exactly which versions of the respective programs shall be used.

3.5.1 What is needed

Before you can begin with variant calling using OVarFlow, some prerequisites have to be fulfilled. Especially certain files have to be given. Most of the time externally supplied files:

- a reference genome in fasta format (the higher the quality the better) and
- a reference annotation in gff format (same as above).

Most of the time provided by you:

- Illumina sequencing files in fastq format of various individuals and
- (optionally) GVCF files of previous variant calling workflows on the same reference genome.

The optional incorporation of previously created GVCF files allows you to include variants you determined previously, saving you from recomputation of those variants.

Furthermore you will need to provide:

- a specific directory structure containing the previous mentioned files,
- a CSV file for configuration,
- the Snakefile and its Python scripts and
- a Conda environment containing the needed applications.

This section will teach you how to set everything up. It is assumed that you have a working Conda installation. If not, have a look at the "Setup & Preparations" section. In the end the following directory structure will be created:

```
/path/to/project_dir/
/path/to/project_dir/conda_env/
/path/to/project_dir/variant_calling/
/path/to/project_dir/variant_calling/FASTQ_INPUT_DIR/
/path/to/project_dir/variant_calling/REFERENCE_INPUT_DIR/
/path/to/project_dir/variant_calling/OLD_GVCF_FILES/
/path/to/project_dir/variant_calling/Snakefile
/path/to/project_dir/variant_calling/scripts/average_coverage.awk
/path/to/project_dir/variant_calling/scripts/createIntervalLists.py
/path/to/project_dir/variant_calling/samples_and_read_groups.csv
/path/to/project_dir/variant_calling/config.yaml
```

3.5.2 Creating a Conda environment

Most of the time Conda environments reside in the home directory of the respective user, which is fine on single user systems. In larger computational environments it is advisable to create project specific Conda environments, that reside by your data.

```
conda create --prefix /path/to/project_dir/conda_env
```

The above command will create a so far empty Conda environment that is named `conda_env` and resides under the specified path. The next step is to install all applications required by OVarFlow. For this you can use a yml file, listing all direct dependencies of OVarFlow.

```
conda env update --prefix /path/to/project_dir/conda_env \
                 --file OVarFlow_dependencies_mini.yml
```

Now you can activate the Conda environment, so the installed applications are available in your `$PATH` variable.

```
conda activate /path/to/project_dir/conda_env
```

If the command was successful your prompt will change to show the path of the Conda environment at the beginning of the prompt in parentheses.

3.5.3 Preparing OVarFlow

With the Conda environment active, you now have access to all needed applications. Lets create a directory for the actual variant calling:

```
mkdir /path/to/project_dir/variant_calling
```

Now four files from OVarFlow's GitLab repository have to be placed within this directory as follows:

```
/path/to/project_dir/variant_calling/Snakefile
/path/to/project_dir/variant_calling/scripts/createIntervalLists.py
/path/to/project_dir/variant_calling/scripts/average_coverage.awk
/path/to/project_dir/variant_calling/samples_and_read_groups.csv
/path/to/project_dir/variant_calling/config.yaml
```

The Python script (createIntervalLists.py) must be executable (file permission `x`):

```
cd /path/to/project_dir/variant_calling/scripts/
chmod ug+x createIntervalLists.py
```

Finally three directories have to be created, that will harbor the reference genome and annotation, the fastq Illumina sequencing files and GVCF files from previous variant callings. Those three directories can either be created manually or the OVarFlow Snakefile will create them. If OVarFlow creates those directories any spelling mistakes are prevented:

```
cd /path/to/project_dir/variant_calling/
snakemake -np
```

The script will prompt about the creation of those directories and exit itself. Now three additional directories have been created:

```
/path/to/project_dir/variant_calling/FASTQ_INPUT_DIR/
/path/to/project_dir/variant_calling/REFERENCE_INPUT_DIR/
/path/to/project_dir/variant_calling/OLD_GVCF_FILES/
```

The respective files have to be placed into the corresponding directory. It has to be mentioned that OVarFlow has only been tested with **a single pair of forward and reverse fastq files per individual**. So in case your reads are derived from various lanes or sequencing runs you should combine all forward and reverse reads in single R1 (forward) and R2 (reverse) files.

3.5.4 The CSV configuration file

The last mandatory step before the actual variant calling is the preparation of a configuration file:

```
/path/to/project_dir/variant_calling/samples_and_read_groups.csv
```

A template of this file is also available within OVarFlow's GitLab repository. Besides its purpose as configuration file it also serves as documentation of the data evaluation. This CSV (colon separated values) file has to list the files that were placed in the previously created directories. Furthermore it contains the read group information for each pair of sequencing files. Also short contigs can be excluded from the reference genome. Lower quality genomes usually are more fragmented and can contain thousands of small contigs besides some larger ones. In cases when you're not interested in the small contigs you might exclude those contigs, thereby accelerating data evaluation. An excerpt of the CSV file might look like this:

Reference Sequence:	cow_ref.fa.gz		
Reference Annotation:	cow_ref.gff.gz		
Min sequence length:	800		
old gvcf to include:	previous_1.gvcf.gz	previous_2.gvcf.gz	
forward reads	**reverse reads**	**ID**	**more read group data**
cow.purple_R1.fastq.gz	cow.purple_R2.fastq.gz	id_1	values ...

Static values of the CSV file are set in bold face. The other fields have to be put in by the user of OVarFlow. Of course if you don't have any GVCF files from previous data evaluations those fields are empty, still its heading will have to be present. If you don't want to exclude any short sequences you can set the value to 1, meaning that every contig would at least have to contain 1 base. Due to space limitations not every read group data is listed here.

To fill in your own values into the CSV file you might use a text editor of your liking (UTF-8 encoding and Unix line breaks should be supported). Also you might use a spreadsheet calculation program. LibreOffice Calc has been tested for this purpose and works just fine. This offers the advantage of the more human readable formatting.

What are read groups?

The previous section mentioned read groups. This section is for those people that are not familiar with this term. The GATK website offers a detailed description. Basically a read group is the set of reads, that is produced in a sequencing experiment. In practical use of OVarFlow it's best to think of it as the forward and reverse reads of a single individual.

The *read group data* now are some meta data of the sequencing experiment. Those data will be used in the mapping of the Illumina reads and become incorporated within the bam files. Also those information will be used as a heading in the final vcf file to identify the columns of the respective individuals. Therefore an especially meaningful and unique name has to be chosen for the *SM - unique sample name* field. To list all of the required read group data fields:

- *forward reads*: those files have to end with `_R1.fastq.gz`
- *reverse reads*: those files have to end with `_R2.fastq.gz`
- *ID*: a unique ID of your liking (e.g. ID_sampleName)
- *PL - platform technology*: OVarFlow has been tested and designed to be used with Illumina reads, so the value is Illumina
- *CN - sequencing center*: an ID for the sequencing center that generated the reads
- *LB - library name*: a unique ID of your liking (e.g. lib_sampleName)
- *SM - uniq sample name*: choose a short, unique name for the sample

3.5.5 The YAML configuration file

Optionally further fine-tuning of the workflow is permitted through a final configuration file:

```
1 /path/to/project_dir/variant_calling/config.yaml
```

Many applications used within the workflow are based upon Java. Resource usage of the Java virtual machine (JVM) is strongly influenced by the underlying hardware. Unfortunately default values for Java heap size and the number of parallel garbage collection threads are not always set to optimal values. While reasonable default values are already defined within the Snakefile, the yaml file allows for modifications in case that a certain data set requires unique settings.

Also the degree of parallelization of the workflow can be modified. The default settings will adjust `bwa` to use six threads for each mapping and HaplotypeCaller to operate on four intervals in parallel. Depending on the structure of the reference genome the number of intervals that are evaluated in parallel cannot be guaranteed, as a given genome is only split between full contigs. Individual contigs won't be split.

Generally this file doesn't need to be present, but it enables adjustments if special circumstances should cause a demand to do so.

3.5.6 Starting the workflow

Now that your Conda environment is active and all files are in place, it's time to start the actual variant calling workflow of OVarFlow. First of all change into the `varaint_calling` directory were the Snakefile resides:

```
cd /path/to/project_dir/variant_calling/
```

OVarFlow also allows for the functional annotation of the detected variants. This is done within the last step of the workflow deploying `snpEff` as a tool to do so. As this is the last step of a long process it is especially annoying if this step fails. From personal experience `snpEff` is not able to make use of every genome annotation. Gff annotations available from the RefSeq have proven to be reliable. Still it is recommended to test if `snpEff` can make use of the provided reference annotation. If it fails, it is better to fail early. First perform a dry run (`snakemake -np`) and then the actual creation of the snpEff database:

```
snakemake -np create_snpEff_db
snakemake -p create_snpEff_db
```

Actual creation of the database might take a considerable amount of time. As OVarFlow is based upon Snakemake it will detect if a database is already available and won't recompute it during the workflow. In case anything failed have a look at the log file:

```
less -SN /path/to/project_dir/variant_calling/logs/snpEffDB/Huhn_2Mio/report_stdout.log
less -SN /path/to/project_dir/variant_calling/logs/snpEffDB/Huhn_2Mio/report_stderr.log
```

Now that the annotation has shown to be usable a dry run of the complete workflow is advisable. This won't perform any actual work, but will prompt all the steps that have to be executed during the variant calling workflow:

```
snakemake -np
```

Finally the actual variant calling workflow can be started. Depending on your given hardware resources you will probably want to parallelize the whole process. This can easily be done by providing a command line option to Snakemake. The `--cores <number>` switch will advice Snakemake to use the given number of threads/cores (alternatively, the number of jobs can be specified `--cores <number>`). This allows OVarFlow to operate on several files or several genomic intervals in parallel, thereby accelerating the computation and shortening the required time.

```
snakemake -p --cores <number>
```

The rest is automatically handled by OVarFlow. Depending on the provided computational resources and number as well as size of sequencing files computation might take several days or even some weeks. During the process several new directories will be created, storing intermediate and final results:

```
00_FastQC
01_mapping
02_sort_gatk
03_mark_duplicates
04_haplotypeCaller
05_gathered_samples
06_combined_calls
07_genotypeGVCFs
08_split_SNPs_Indels
09_hard_filtering
10_merged_filtered_VCF
11_filtered_removed_VCF
12_annotated_variants
```

(continues on next page)

(continued from previous page)

```
benchmarks
interval_lists
logs
processed_reference
snpEffDB
```

3.5.7 Alternative targets

In the above usage, the default Snakemake workflow target rule is applied. To be even more versatile, some alternative target rules are available that execute only a subset of the entire workflow. This may save some computation time for unnecessary calculations. However, it is always possible to rerun the entire workflow, as Snakemake will recognize results that have already been calculated and resume the workflow from there. The following alternative target rules are available:

noSnpEff

In case that no functional annotation of the detected variants is desired, an alternative target is available, called `noSnpEff`. By specifying this as the last option of the workflow invocation, everything will be executed as before, except for the functional annotation of the variants.

```
snakemake -p --cores <number> noSnpEff
```

Without executing SnpEff, there is also no need to specify a reference annotation within the CSV configuration file. Still, the respective line has to be present in the file but no option has to be stated, e.g.:

```
Reference Sequence:,SampleSeq.fa
Reference Annotation:,
...
```

variantsPerSample

The target `variantsPerSample` is not available in OVarFlow 2.

This alternative target rule executes the workflow up to the variant calling of each individual samples. Thereby, every directory including `05_gathered_samples` will be created.

```
snakemake -p --cores <number> variantsPerSample
```

dedubBAM

Finally, the workflow can also be utilized to perform just the mapping, including sorting and marking of duplicated reads. In this process, every directory including `03_mark_duplicates` will be created. Also, statistics of the average coverage will be calculated.

```
snakemake -p --cores <number> dedubBAM
```

3.6 Configuration & adaptation

Every variant calling task and every computer system has its own specifics. Therefore, variant calling via OVarFlow can be adjusted by two configuration files. The design decision for two different files was made consciously, as both serve different purposes. One file provides the sample data, the other configures the workflow and its resource usage. Prototypes of both files are to be found within the OVarFlow repository.

3.6.1 The CSV file

The file `samples_and_read_groups.csv` is **mandatory**. As the name implies it provides all data that are related to the respective sample. In doing so, this file is also interesting for the biological side or cooperation partner, respectively. Therefore, the comma separated values (CSV) format was chosen, as it can easily be displayed and edited by spreadsheet applications.

The prototype of the file looks as follows:

```
1  Reference Sequence:,Huhn_2Mio.fna.gz
2  Reference Annotation:,Huhn_2Mio.gff.gz
3
4  Min sequence length:,2000
5
6  old gvcf to include:,fake_1.gvcf.gz,fake_2.gvcf.gz
7
8  forward reads,reverse reads,ID,PL - plattform technology,CN - sequencing center,LB -␣
   ↪library name,SM - uniq sample name
9  GGA081_R1.fastq.gz,GGA081_R2.fastq.gz,id_GGA081,illumina,UBern,lib_GGA081,GGA081
10 GGA112_R1.fastq.gz,GGA112_R2.fastq.gz,id_GGA112,illumina,UBern,lib_GGA112,GGA112
```

With this file lines 1 to 9 have to be present. The number of lines starting from line 10 is arbitrary. It only depends on the number of samples that shall be processed. But don't include empty lines. Overall the file is used to provide the following information and has to be modified accordingly:

Reference sequence

Position: **line 1**, **field 2**
Allowed endings: **.fa.gz**, **.fna.gz**, **.fasta.gz**

Reference annotation

Position: **line 2**, **field 2**
Allowed endings: **.gff.gz**
Other formats might work, but were not tested.

Min sequence length

Position: **line 4**, **field 2**
Short contigs within the reference genome can be excluded from the analysis. Here the cutoff value for the minimum sequence length is defined. This value can be set to 1 to include any contig, no matter of its length. A value has to be given in any case.

Old gvcf files

Position: **line 7**, **field 2 to n**
Allowed endings: **.gvcf.gz**

OVarFlow can pickup gvcf files containing the variants of single individuals. **Caution:** Those variants have to be called on the same reference genome as used in the current analysis! Additional gvcf files have to be listed in succession, separated by commas. The fields 2 to n may be left blank, if no given variants shall be included.

Sample information

Position: **line 9 to n**, **fields 1 to 7**

Here the sample information of the given analysis has to be listed. This includes the name of the fastq.gz files and read group data. All fields are required. There is a **mandatory naming scheme** for the filename suffix: `_R1.fastq.gz` for forward reads and `_R2.fastq.gz` for reverse reads. At least line 9 (a single sample) has to be present, with an arbitrary number of succeeding lines. No empty lines shall be present.

3.6.2 The yaml file

The file `config.yaml` is **fully optional**. Most of the time OVarFlow will work without this file. Still it provides the ability to change some internal settings of OVarFlow. OVarFlow has been highly optimized (see section: Benchmarking & Optimizations). However, not every possible analysis can be foreseen. There might be combinations of genomes and sequencing data that require different settings for the Java virtual machine. Such modifications of OVarFlow are possible through this configuration file.

Considering the purpose of this file it is obvious that it is intended for the technical or bioinformatics user of the analysis. Therefore, such information were separated from the more biological data of the CSV file.

The prototype of the file looks as follows:

```
# yaml file listing optionally available configuration
# options for OVarFlow
# no option nor the yaml file itself must be present
# here the default options of OVarFlow are listed

heapSize:
    SortSam         : 10
    MarkDuplicates  : 2
    HaplotypeCaller : 2
    GatherIntervals : 2
    GATKdefault     : 12

ParallelGCThreads:
    SortSam         : 2
    MarkDuplicates  : 2
    HaplotypeCaller : 2
    GatherVcfs      : 2
    CombineGVCFs    : 2
    GATKdefault     : 4

Miscellaneous:
    BwaThreads      : 6
    BwaGbMemory     : 4
    GatkHCintervals : 4
    HCnpHMMthreads  : 4
    GATKtmpDir      : "./GATK_tmp_dir/"
    MaxFileHandles  : 300
    MemoryOverhead  : 1

```

(continues on next page)

(continued from previous page)

```
Debugging:
    CSV             : False
    YAML            : False
```

The above template also documents the default settings that are hard-coded within the OVarFlows Snakefile. Therefore, no changes would be applied using this file. Of course, all the numeric values can be changed. The purpose of the individual blocks are as follows:

heapSize *Range of accepted values:* **1 - 40**

Here the Java heap size is defined, that will be used by the individual GATK applications. The values are provided in Gb. A value of e.g. 4 will be equivalent to setting the Java option `-Xmx4G`. The given names are equivalent to the respective GATK application. Only GatherIntervals is equivalent to GatherVcfs (in earlier versions CombineGVCFs) in cases were the HaplotypeCaller was acting on individual intervals and not the whole genome. All remaining GATK applications, that are not executed in parallel, will use a default value of 12 Gb.

The default values were chosen to minimize memory footprint while still being quite generic. Of course, not every possible combination of a reference genome and sequencing data can be tested. In case that the provided amount of memory for a specific analysis is not sufficient Java will throw a `java.lang.OutOfMemoryError` error. Under these circumstances increase the heap size of the affected application for the specific analysis.

ParallelGCThreads *Range of accepted values:* **1 - 20**

The number of ParallelGCThreads has been optimized to be resource efficient while still allowing for quick data evaluation. Currently it is rather unlikely that any changes are required. Still, just to be on the save side, changes are possible. Finally changes to future GATK tools cannot be foreseen, that might require a modification. Changes are equivalent to modifying the JVM option `-XX:ParallelGCThreads=<n>`.

Miscellaneous *Range of accepted values:*

bwa threads: **1 - 40**
bwa memory: **1 - 20** (gigabyte)
HaplotypeCaller intervals: **1 - 100**
HaplotypeCaller native pair hmm threads: **1 - 6**
tmp directory: **/tmp[/subdirectory]** or **./<name>[/subdirectory]**
File handles MarkDuplicates: **10 - 40000**
Java memory overhead **0 - 10** (gigabyte)

Settings of the quantity of bwa mem thread and HaplotypeCaller intervals influence the parallelization of the respective application. Of course, reasonable values depend on the given hardware resources, which are ultimately limiting. No perfect generic value can be set, that is suitable for every user. The setting for bwa mem adjusts the number of parallel mapping threads. The number of HaplotypeCaller intervals splits the reference genome into intervals that are processed in parallel.

For the mapping tasks that use bwa mem, a resource request of 4 gb memory should be fine. However, some combinations of sequencing data (fastq) and reference genome (fa) might require more than 4 gb of memory. In particular in cluster usage, where resource requests are strictly controlled, exceeding these requests might result in application termination. Thus, in such a situation, memory limits have to be increased.

The HaplotpyeCaller possesses an option to adjust the number of so-called *native pairHMM threads*. The higher this number, the higher the CPU usage of the HaplotypeCaller (also see benchmarking section). To execute more HaplotypeCallers in parallel a value of `1` should be chosen. In this case each individual HaplotypeCaller will run a little bit slower. Otherwise `4` is the optimal value (which is the default).

The directory where GATK stores temporary data can be configured. Allowed values are `/tmp` and subdirectories therein as well as any directory within the current working directory (`./<name>`). Directory names have to consist of alphanumeric characters, `.`, `_`, `-` and `/`.

GATK MarkDuplicates tends to open an enormous amount of files. The number of open file descriptors can be beyond the limits of some systems (see `ulimit -Hn` and `ulimit -Sn`). A fixed value of 300 file descriptors is the default of OVarFlow but can be adjusted to fit custom needs.

The workflow has been designed so that memory usage can be scheduled via the `--resources mem_gb=<n>` command line option, just like CPU usage can be schedule through the `--cores <n>` directive. For Java applications, memory scheduling takes into account the respective heap size plus an additional overhead for non-heap memory. This overhead is set to 1 gigabyte by default. This setting is particularly useful when memory resources are requested in cluster usage.

Debugging *Range of accepted values:* **True** or **False** (no quotation marks)

This option enables the debugging output, showing the settings provided by the csv (sample file) and the yaml file (configuration file). Actually any value that translates to a boolean Python value is possible, but `True` is the only reasonable choice.

Not only is the usage of the yaml file optional, also not every setting is required. If default settings for most applications shall be preserved those settings don't need to be provided. The following example would also be a valid `config.yaml` file:

```
heapSize:
    GatherIntervals : 5

Miscellaneous:
    BwaThreads      : 8
    GatkHCintervals : 8
```

3.6.3 Memory recommendations

As noted above, some combinations of given reference genome and sequencing data may require different settings in the `config.yaml` file. Genome size, genome fragmentation, and sequencing depth affect memory requirements. Most thorough testing has been performed using a chicken genome (GRCg6a) and sequencing data up to 34-fold average coverage. Based on those experiences, default values were selected that were also found to be appropriate for various mammalian datasets. However, some mammalian data might require more memory, in particular if the risk of a failed workflow is to be avoided at all costs (thanks to Snakemake, a failed workflow can be restarted at any time). There are two situations where low memory requests can cause the workflow to fail:

- A `java.lang.OutOfMemoryError` if the heap size was chosen to low. MarkDuplicates and the HaplotypeCaller might be affected by this.
- An `oom-kill event` can occur in cluster usage where resource request are strictly controlled. An error message like the following is indicative for such a situation: *Some of your processes may have been killed by the cgroup out-of-memory handler.* Such an error happens when the "Java memory overhead" or if memory request for bwa is insufficient.

A minimal configuration, which should be fairly safe, looks like this:

```
heapSize:
    MarkDuplicates  : 4
    HaplotypeCaller : 3

Miscellaneous:
    BwaGbMemory     : 8
```

When choosing such values, keep in mind that unnecessarily high values block resources, especially when using compute clusters. To learn about reasonable values for the given use case, one can perform an initial run with save settings

and observe the actual consumed resident set size based on the recorded benchmarks in the `benchmarks` directory. A sorted list of the consumed memory, for instance during mapping, can be obtained with the following shell commands:

```
cd benchmarks/01_mapping
for f in *bm
do
    tail -1 $f | cut -f3
done | sort -n | less
```

The largest values can then be used as an indicator of the upper bounds of the memory requirements of the given data set.

3.7 Advanced usage topics

The previous section was a broad introduction into the general usage of OVarFlow. The commands were basically executed on a single computer or server and program execution was supposed to proceed flawlessly. However that's the ideal use case. This section is dedicated to the execution of OVarFlow in a more advanced computer environment and gives advice in case of failure. Furthermore advice is given to achieve reproducibility even on different computational hardware.

3.7.1 Cluster usage - SGE

Variant calling is computationally demanding. Common desktop computers are by no means suitable for this task. So that at least high end desktop (HEDT) CPUs are required, offering two dozens of cores or even more. Of course the more resource are made available the quicker the computation can be finished. Therefore usage of a compute cluster should also be considered.

Obviously this section is dedicated to the cluster usage of OVarFlow. As OVarFlow is based upon Snakemake, all options privided by Snakemake for cloud computing and cluster usage are available as well. OVarFlow has successfully been used with **Sun Grid Engine (SGE)** (Son of Grid Engine is its successor). The following examples have to be adopted for the use with other cluster management software.

On a single large cluster node

One issue with GATK is, that some of its applications cause rather short load peaks, while using much more moderate resource during the majority of their runtime. When using a compute cluster, a general rule of thumb is to request the maximum amount of resources, that is to be expected during runtime. The request of maximum resource requirements will prevent a slowdown of the cluster node, but also waste the resources that are not used most of the time.

One idea to handle this contradiction is to reserve all resources of one cluster node to be used by OVarFlow. Peak loads are more or less ignored within this approach. OVarFlow is then configured to utilize all available resources most of the time. So if there is a slowdown within the used cluster node, due to peak loads, only OVarFlow will be affected, thereby not standing in the way of other users.

This approach has successfully been used on an SGE based cluster. If you're using a different cluster management software you'll have to adopt the steps accordingly. Complex cluster jobs are best submitted using a submit shell script. At the beginning of such a script the resource requests can be configured. If chosen correctly this will result in the exclusive reservation of the cluster node for OVarFlow.

This procedure is only recommended if reasonably capable cluster nodes are available. In a test a single cluster node provided:

- 40 threads (Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz)

- 256 GB of memory

By requesting 40 cores (<parallel_environment_name> has to be replaced with your local configuration), the node was exclusively reserved. Also the hostname= directive will let you choose a certain target host. The use of wildcards allows for the selection of a range of certain cluster nodes. An example of a whole submit script is listed below:

```
#!/bin/bash

#$ -q queue_name.q
#$ -cwd
#$ -V
#$ -o output.txt
#$ -e error.txt
#$ -pe <parallel_environment_name> 40
#$ -l virtual_free=6000M
#$ -l hostname=<node-prefix*>

# write the name of the execution host to a file
hostname > TIME_STAMP
# write down the starting time
echo "start at:" >> TIME_STAMP
date >> TIME_STAMP

# the Conda environment has to be activated
# in older Conda versions -V was needed using the following command
#. activate /path/to/project_dir/conda_env
# now you use
conda activate /path/to/project_dir/conda_env

snakemake -p --cores 40

# write down the end time
echo "end time:" >> TIME_STAMP
date >> TIME_STAMP
```

With SGE the actual submission of the job to the cluster is then achieved through a single command:

```
qsub <submit_script_name>
```

On an entire cluster

In case that potential load peaks caused by GATK are of no concern, an entire Cluster for OVarFlow is possible as well. Thereby unleashing all of the resources of the entire cluster to OVarFlow. This is probably the most efficient way of using OVarFlow.

The ability for cluster execution is build into Snakemake. But precaution has to be taken, that the Conda environment is available for the cluster jobs. One option is to export the current environment with the -V option of the qsub command:

```
conda activate /path/to/project_dir/conda_env
cd /path/to/project_dir/variant_calling
snakemake -p --cores 20 --cluster 'qsub -V -cwd -b y -pe <parallel_environment> {threads}
↪'
```

The above command will automatically use the number of {threads} for each command as defined in the workflow, while using a maximum of 20 threads in parallel for all currently submitted jobs. Certainly the

<parallel_environment> argument must be replaced with the respective name of your local cluster environment. One drawback is, that the cluster management software will write four log files for each submitted job into the current directory. To tidy things up, it is advisable to create a log directory and include this into the cluster submission command:

```
conda activate /path/to/project_dir/conda_env
cd /path/to/project_dir/variant_calling
mkdir logs_cluster
snakemake -p --cores 20 --cluster 'qsub -V -cwd -o logs_cluster -e logs_cluster -b y -pe
↪<parallel_environment> {threads}'
```

One thing to consider is, that the above command will reveal all of the users environment variables publicly (`qstat --cores <job_number>` under the entry `env_list`).

3.7.2 Cluster usage - Slurm

Slurm is an alternative cluster managment and job scheduling system. In recent years it gained popularity over SGE, which suffers from maintenance issues. Fortunately, Snakemake and Slurm work with one another just as well as SGE does.

On a single large cluster node

Submitting the entire workflow to a single cluster node is quite simple with slurm. The `srun` command accomplishes this purpose. Slurm's job submission system automatically exports the user's current environment (default option `--export=ALL`). To benefit from this feature, the corresponding Conda environment should be activated first. Afterwards, the workflow can be tested through a dry run:

```
conda activate /path/to/project_dir/conda_env
srun -c 28 --mem 50g snakemake -np --cores 26 --resources mem_gb=48
```

The above command also illustrates how resource requests can be made. Many cluster systems are very strict about resource requests. In this case, a job won't be allowed to use more CPU resource than requested, and if a job consumes more memory than requested, it might even be terminated. The above command performs resource requests in two ways. First, 28 cores and 50 Gb of memory are requested from the cluster management system (the values used are just an example and need to be adjusted to the size of the given dataset). Second, to ensure that the requested resource are not exceeded, the Snakemake scheduler is configured to use slightly less resources. It should be noted that resource requests are neither automatically transfer nor matched between Slurm and Snakemake. Therefore, both must be set manually. Once the dry run has been successfully executed, the actual workflow can be started:

```
srun -c 28 --mem 50g snakemake -p --cores 26 --resources mem_gb=48
```

On an entire cluster

Of course, when dealing with very large datasets, it is not reasonable to restrict oneself to the use of a single cluster node. Snakemake itself is capable of handling such a large scale submission, but on the side of Slurm, the `sbatch` command has to be used this time:

```
conda activate /path/to/project_dir/conda_env
snakemake --default-resources mem_gb=16 -p --cores 200 --cluster 'sbatch -c {threads} --
↪mem {resources.mem_gb}G'
```

Again, the Conda environment is activated first. This time, no dry run is performed, but the workflow is executed directly. Depending on the size of the given dataset and cluster, a reasonable large number of `n` cores should be used. Snakemake automatically schedules resource requests (`{threads}` and `{resources.mem_gb}`) for each individual job. These resource requests are based upon the setting made in the `config.yaml` file. For instance, memory requests are inferred from the configured heap size and Java memory overhead (see configuration section). By this mechanism, reasonable settings are selected for most jobs. Not every rule of the workflow has a predefined memory requirement. For these rules, a default of 16 Gb is requested in the above command.

3.7.3 Trouble shooting

OVarFlow has been designed for easy and automatic execution of the variant calling workflow. As the underlying processes are quite complex and involve a lot of various software tools, runtime errors can not be excluded within those tools, that OVarFlow is ultimately relying on. Furthermore the respective computational environment can be a source of failure as well. Variant calling will involve high performance computing most of the time. This involves a variety of hardware resources with servers dedicated to computation and others dedicated to data storage. Failure in this environment might result in unavailability of data, causing a running calculation of OVarFlow to fail and ultimately leading to termination of the workflow.

The good news is, thanks to its Snakemake basis, OVarFlow can often recover from such situations. Albeit manual intervention might be needed in such circumstances.

Often Snakemake itself can pick up an interrupted workflow. Executing a dry run might give first insights:

```
snakemake -np --rerun-incomplete
```

If this command succeeds the real execution can be performed, of course while specifying a reasonable number of threads (`--cores`) to accelerate calculations. Also it might be interesting to see the reason why a specific command is executed (`--reason`):

```
snakemake -p --cores <number_of_threads> --rerun-incomplete  --reason
```

In case that Snakemake was interrupted previously, it might block re-execution (so-called lock):

```
Error: Directory cannot be locked. Please make sure that no other Snakemake process is␣
↪trying
to create the same files in the following directory:
/path/to/project_dir/variant_calling
If you are sure that no other instances of snakemake are running on this directory, the
remaining lock was likely caused by a kill signal or a power loss. It can be removed␣
↪with the
--unlock argument.
```

In such a situation the lock can be removed and the workflow can be rerun:

```
snakemake --unlock
snakemake -p --cores <number_of_threads> --rerun-incomplete --reason
```

Some times more manual interaction is necessary. First of all it's good to know in which step the error occurred. Therefore the following points could be checked:

- The console log of the commands that were executed by Snakemake.
- The log messages of every single command, written within the `log` directory.
- The files and directories that were already created by the workflow.

- Is every created file complete? Compared with other files of the same type are file sizes reasonable and are the expected index files (e.g. .bam and .bam.bai) present?

If certain files are corrupted those can be removed manually via the `rm` command. In cases where a disaster recovery is not possible, the whole workflow can be started newly by removing the directories that were created (including `logs` and `.snakemake`) and restarting the workflow:

```
rm -rf 00_FastQC [01_...] logs .snakemake
snakemake -p --cores <number_of_threads>
```

3.7.4 Error identification

The ability to restart a failed workflow can be very helpful. But this feature is only useful, if the problem that cause the workflow to fail can be identified. Here, an example of a failed workflow will be shown and also how to identify the causative problem.

The workflow was terminated with the following final messages:

```
...
Finished job 3039.
1111 of 3097 steps (36%) done
[Sat Jul 10 03:56:53 2021]
Finished job 2816.
1112 of 3097 steps (36%) done
[Sat Jul 10 04:11:46 2021]
Finished job 2939.
1113 of 3097 steps (36%) done
[Sat Jul 10 04:20:47 2021]
Finished job 2814.
1114 of 3097 steps (36%) done
Shutting down, this might take some time.
Exiting because a job execution failed. Look above for error message
Complete log: /path/to/project_dir/.snakemake/log/2021-07-08T142952.561603.snakemake.log
(conda: conda_env)user@host:/path/to/project_dir/variant_calling$
```

Snakemake returned to the shell prompt, after an exhaustive series of jobs was finished, of which one had failed. This resulted in the above error message, only the already running jobs were finished, and no further jobs were started. The actual error message of the job that failed is not shown. It can be identified by either scrolling upwards, till the message appears or by viewing the stated log file (which is a copy of the messages shown on the shell). In doing so, a more comprehensive error message is found:

```
...
Error in rule mark_duplicates:
  jobid: 769
  output: 03_mark_duplicates/sample-82.bam, 03_mark_duplicates/sample-82.txt
  log: logs/03_mark_duplicates/sample-82.log (check log file(s) for error message)
  shell:
      export _JAVA_OPTIONS=-Xmx2G
      gatk --java-options -XX:ParallelGCThreads=2 MarkDuplicates -I 02_sort_gatk/sample-
↪82.bam -O 03_mark_duplicates/sample-82.bam -M 03_mark_duplicates/sample-82.txt -MAX_
↪FILE_HANDLES 300 --TMP_DIR ./GATK_tmp_dir/ 2> logs/03_mark_duplicates/sample-82.log
      one of the commands exited with non-zero exit code; note that snakemake uses bash␣
↪strict mode!)
  cluster_jobid: Your job 963665 ("snakejob.mark_duplicates.769.sh") has been submitted
```

(continues on next page)

(continued from previous page)

```
Error executing rule mark_duplicates on cluster (jobid: 769, external: Your job 963665 (
↪"snakejob.mark_duplicates.769.sh") has been submitted, jobscript: /path/to/project_dir/
↪variant_calling/.snakemake/tmp.jz4k0i0y/snakejob.mark_duplicates.769.sh). For error␣
↪details see the cluster log and the log files of the involved rule(s).
...
```

This error message educates us about the exact job and its rule that failed. In this case, a job spawned from the rule `mark_duplicates` failed. That's a step forward, but the actual error message is still hidden in the reported log file `logs/03_mark_duplicates/sample-82.log`:

```
[Fri Jul 09 14:07:25 CEST 2021] picard.sam.markduplicates.MarkDuplicates done. Elapsed␣
↪time: 45.92 minutes.
Runtime.totalMemory()=1908932608
To get help, see http://broadinstitute.github.io/picard/index.html#GettingHelp
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
        at java.util.Arrays.copyOfRange(Arrays.java:3664)
        at java.lang.String.<init>(String.java:207)
        at java.lang.String.substring(String.java:1969)
        at picard.sam.util.ReadNameParser.getLastThreeFields(ReadNameParser.java:146)
        at picard.sam.util.ReadNameParser.addLocationInformation(ReadNameParser.java:83)
        at picard.sam.markduplicates.MarkDuplicates.buildReadEnds(MarkDuplicates.
↪java:661)
        at picard.sam.markduplicates.MarkDuplicates.
↪buildSortedReadEndLists(MarkDuplicates.java:552)
        at picard.sam.markduplicates.MarkDuplicates.doWork(MarkDuplicates.java:257)
        at picard.cmdline.CommandLineProgram.instanceMain(CommandLineProgram.java:301)
        at org.broadinstitute.hellbender.cmdline.PicardCommandLineProgramExecutor.
↪instanceMain(PicardCommandLineProgramExecutor.java:37)
        at org.broadinstitute.hellbender.Main.runCommandLineProgram(Main.java:160)
        at org.broadinstitute.hellbender.Main.mainEntry(Main.java:203)
        at org.broadinstitute.hellbender.Main.main(Main.java:289)
```

The above error message is a Java stack trace caused by the GATK tool MarkDuplicates. This educates us about the root cause of the problem: `Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded`. The `OutOfMemoryError` tells us, that insufficient Java heap space was provided for this specific job. This problem can easily be fix within the workflow, by providing a larger heap space to MarkDuplicates in the configuration file `config.yaml` (also see the section "Configuration & adaptation"). The required entry could look like this:

```
heapSize:
    MarkDuplicates : 4
```

Thereby, the standard heap size would be doubled. After this simple fix, the workflow can be restarted and finished. Generally, the workflow tries to find a balance between resource efficiency and broad applicability. In some special cases, individual GATK applications need to be provided with more resource, which would be wasteful when evaluating smaller datasets.

3.7.5 Reproducibility

Reproducibility has been among the primary scopes of OVarFlow. Three components of OVarFlow serve this purpose:

- the workflow itself (aka the Snakefile),
- the CSV file to document and configure data evaluation,
- the Conda environment with its specific program versions.

The yml (`OVarFlow_dependencies.yml`) file only includes the major dependencies of OVarFlow. To automatically obtain the latest program versions for new variant callings, no specific program versions are denoted here. To achieve perfect reproducibility all installed programs as well as their versions have to be obtained from a given Conda environment. This is a part of the management of Conda environments. Basically this boils down to the creation of a yml file of the given environment, which includes all programs and their version numbers:

```
conda activate /path/to/project_dir/conda_env
conda env export > conda_environment.yml
```

The Conda environment can easily be recreated out of this yml file:

```
conda env create -f conda_environment.yml
```

3.8 Docker & Singularity usage

A different approach to use OVarFlow is its Docker container or image, respectively. The Docker container bundles all the software that is needed to execute the OVarFlow workflow. There is no need to install or download the individual components of OVarFlow or to create a Conda environment.

This simplification of course comes at a cost. First of all Docker requires system administrator privileges. Also the bundled software components can't be updated, as it can be done with a Conda environments. Finally cluster usage (e.g. SGE) is not the scope of the OVarFlow container. Container usage of OVarFlow has been designed to be utilized on a single larger machine. The need for administrator privileges can be circumvented by the us of Singularity, as an alternative container virtualization technology, instead of Docker.

3.8.1 Docker

The Docker images of OVarFlow, to create a new container, are available on Docker Hub. Different versions of the image might be available. Each version has a distinct tag, showing the build date of the image. The Docker command can be used to download the image from Docker Hub and to make it locally available:

```
docker pull ovarflow/release:<tag>
```

Of course `<tag>` has to be replaced with the version you want to download.

After downloading the image make sure, that it's indeed locally available:

```
docker images
```

Now prepare a directory for the workflow, where your sequencing data are made available and where the results of the workflow will be stored. You can name the directory arbitrarily (here `project_dir`). Also three more directories have to be created within this main directory:

```
mkdir  project_dir
mkdir  project_dir/FASTQ_INPUT_DIR
mkdir  project_dir/REFERENCE_INPUT_DIR
mkdir  project_dir/OLD_GVCF_FILES
```

The three uppercase directories can also be created by OVarFlow, but in any case you would need to fill in the respective content manually. Which is also the next step to perform.

project_dir/FASTQ_INPUT_DIR Has to contain your Illumina sequencing files in fastq format. For each individual to be analyzed you have to provide two files, one file (R1) containing the forward reads and one file (R2) the reverse reads. If you have various sequencing files for each individual merge all forward and all reverse reads beforehand.

project_dir/REFERENCE_INPUT_DIR Has to contain two files: a reference genome in fasta format and a reference annotation in gff format. Files that were obtained from the RefSeq have been utilized successfully with OVarFlow.

project_dir/OLD_GVCF_FILES May contain some gvcf files from previous variant callings. This allows for the inclusion of individuals that were already analyzed, thereby the most time consuming steps (including mapping and variant detection with HaplotypeCaller) won't have to be recomputed. Of course those files must have been analyzed with the same reference genome and annotation, as is used in this analysis.

Finally a csv file called `samples_and_read_groups.csv` has to be present in the `project_dir`. This file serves for the configuration of the workflow, telling OVarFlow which files to use. Thereby this csv file also serves documentational purposes. A sample of this file can be obtained from OVarFlows GitLab repository. A detailed description of the file format can be found under *Conda & Snakemake usage => The CSV configuration file*.

Now that everything is prepared you can create and execute a docker container of OVarFlow:

```
docker run -it -v /path/to/project_dir:/input ovarflow/release:<tag>
```

Of course the `<tag>` has to be replaced with the version you're using. The option `-v` will bind mount a volume within the container. Thereby the directory `/path/to/project_dir` is made available under the path `/input` within the running container.

Now the OVarFlow workflow is already running and no further manual interaction should be required.

Resource utilization

The Docker Image has been designed to make high use of the available resources. The number of available CPU cores (or threads to be more precise) is automatically detected. The OVarFlow will then use *available cores - 4* within its Snakemake workflow. For instance if 32 cores are available OVarFlow will use 28 of those cores, with the internal Snakemake command of `snakemake -p --cores 28 --snakefile /snakemake/Snakefile`. Of course OVarFlow allows for the modification resource utilization. In this case an additional option has to be passed to the OVarFlow container forwarding an environment variable:

```
docker run -it -e THREADS='<number>' -v /path/to/project_dir:/input ovarflow/release:
↪<tag>
```

In any case the `<number>` that is passed to OVarFlow should not exceed the number of available threads. It is the users responsibility to take care of this.

Obtaining the yml file

If you need to know about the single software versions that are used within OVarFlow's Docker container, you can also extract that information from the container. To do so you must first open a shell within the container.

```
docker run -it -v /home/ubuntu/project_dir:/input ovarflow/release:<tag> /bin/bash
```

Within the running container make the Conda environment available and extract the version information to a yml file:

```
conda init bash
bash
activate conda OVarFlow
conda env export > /input/conda_env_OVarFlow.yml
exit; exit
```

The above commands perform the following actions: (1) initializes Conda. (2) the changes made in the previous step must be made available within a newly opened `bash` shell. As can be seen from the changes to the prompt (`(base) root@...:/#`) the Conda base environment is now active. (3) activates the OVarFlow Conda environment. The prompt changes again (`(OVarFlow) root@...:/#`). (4) exports the OVarFlow environment into a yml, that will be written to `/path/to/project_dir`, outside of the Docker Container. (5) will log you out of the two opened bash shells.

Final note on Docker

One thing that has to be mentioned is, that every time `docker run` is invoked, a new container is created from the OVarFlow Docker image. To get an overview of the containers that were already created execute `docker ps -a`. It might be reasonable to sometimes delete old containers `docker rm <container_name>`.

3.8.2 Singularity

Singularity allows you to do the same tasks that Docker does, but without the need for administrator privileges. Making Singularity a popular choice in high performance scientific computing. Also usage of Singularity containers is generally a bit different from Docker images and containers. First of all create a sif file (Singularity image format) from the Docker image. The data will be retrieved from Docker Hub:

```
singularity build OVarFlow_<tag>.sif docker://ovarflow/release:<tag>
```

This sif file contains the whole OVarFlow workflow including all software dependencies. Now prepare a `project_dir` as it was done with Docker (see above). The workflow can now be started via:

```
singularity run --bind /path/to/project_dir:/input OVarFlow_<tag>.sif
```

Just like with Docker, executing OVarFlow with Singularity, will autodetect the number of cores (threads) that are available on the respective computer. Again the default setting of the used number of cores is *available cores - 4*. Changing this setting by setting an environment variable called `THREADS` and then running the Singularity container:

```
export THREADS=<desired_number_of_threads>
singularity run --bind /path/to/project_dir:/input OVarFlow_<tag>.sif
```

Manual start of OVarFlow

Singularity also makes the OVarFlow workflow accessible from a command line. Singularity easily allows to run a shell within the container.

```
singularity shell --bind /path/to/project_dir:/input OVarFlow_<tag>.sif
```

This command will bind mount (`--bind`) the project directory within the container under the path `/input`. Also the users home directory is automatically available within the container. The root folder (`/`) of the host operating system will be overlaid by the root of the container. Therefore the bind mount command is needed as no directory outside of the users home will be available otherwise.

In case that there is a warning `bash: warning: setlocale: LC_ALL: cannot change locale (en_US.utf8)`, the message can be ignored. It won't interfere with the workflow.

After opening the shell, you might for instance want to perform a dry run of Snakemake:

```
cd /input
snakemake -np --snakefile /snakemake/Snakefile
```

Or start the actual workflow, like it would be done with the manual installation of OVarFlow:

```
cd /input
snakemake -p --cores <threads> /snakemake/Snakefile
```

Starting the BQSR-workflow only requires a different Snakefile:

```
cd /input
snakemake -np --snakefile /snakemake/SnakefileBQSR
snakemake -p --cores <threads> /snakemake/SnakefileBQSR
```

Obtaining the yml file

The exact software versions, that are being used in the Singularity container, can also be extracted into a yml file. First of all a shell can easily be opened within the Singularity container:

```
singularity shell OVarFlow_<tag>.sif
```

The users home directory will automatically be mounted with the now running Singularity container, and all data from the home directory are thereby accessible. Besides this the whole content of the container is available. Therefore the OVarFlow Conda environment can be activated and exported. The commands are identical to ones used with the Docker container:

```
conda init bash
bash
conda avtivate OVarFlow
conda-env export > /path/to/project_dir/conda_env_OVarFlow.yml
exit; exit
```

3.9 The BQSR workflow

The workflow previously described, will already supply a fully annotated set of variants. To further refine the called variants, the GATK team recommends to perform base quality score recalibration (BQSR). Therefore BQSR was implemented in a second workflow, that can optionally be run in succession to the first workflow, to further improve the called variants through BQSR. The most obvious downside is, that the execution of a second workflow will increase the time till the final results are available, nearly doubling it. Also a second point should at least be considered. The called variants are used to refine the given data set itself. This self-improvement might at least potentially introduce a certain bias. Still this procedure is strongly recommended by the GATK team and should therefore be legit.

3.9.1 Setup & preparations

The BQSR workflow requires some additional applications. Especially *GATK AnalyzeCovariates* heavily relies on `R` and several *R packages*. The lack of those packages will result in error logs under `logs/15_analyze_BQSR/<sample>_AnalyzeCovariates.log` like the following:

```
Stderr: Error in library(gplots) : there is no package called 'gplots'
```

Therefore a specialized Conda environment has to be created for the BQSR workflow. This Conda environment will include all applications used in the normal workflow, plus `R` and the required *R packages*. If the BQSR workflow shall be run anyways, this Conda environment can also be used for the normal workflow. It is created like this (the YAML file is found in the repository):

```
conda env create --prefix /path/to/project_dir/BQSR_env --file BQSR_dependencies_mini.yml
```

Finally the new Conda environment has to be activated:

```
conda activate /path/to/project_dir/BQSR_env
```

Now the actual workflow has to be copied from the repository and placed into the project directory:

```
/path/to/project_dir/variant_calling/SnakefileBQSR
/path/to/project_dir/variant_calling/configBQSR.yaml # optionally
```

3.9.2 Workflow usage

The BQSR workflow builds upon the normal workflow. As a result of this the normal variant calling has to be preformed first and the following files have to be created through this workflow:

```
/path/to/project_dir/variant_calling/03_mark_duplicates/<file_names>.bam
/path/to/project_dir/variant_calling/03_mark_duplicates/<file_names>.bam.bai
/path/to/project_dir/variant_calling/11_filtered_removed_VCF/variants_filtered.vcf.gz
/path/to/project_dir/variant_calling/interval_lists/<interval_xy>.list
/path/to/project_dir/variant_calling/processed_reference/<file_name>.fa.gz
/path/to/project_dir/variant_calling/processed_reference/<file_name>.fa.gz.fai
/path/to/project_dir/variant_calling/processed_reference/<file_name>.fa.gz.gzi
/path/to/project_dir/variant_calling/processed_reference/<file_name>.dict
/path/to/project_dir/variant_calling/snpEffDB/<directory_name>/genes.gff
/path/to/project_dir/variant_calling/snpEffDB/<directory_name>/sequences.fa.gz
/path/to/project_dir/variant_calling/snpEffDB/<directory_name>/snpEffectPredictor.bin
```

An initial dry run can be performed to test for any missing files, spelling mistakes and the like. In any case the Snakefile for the BQSR workflow must explicitly be specified (`-s` option), otherwise the normal workflow might be reexecuted:

```
cd /path/to/project_dir/variant_calling/
snakemake -np -s SnakefileBQSR
```

Finally the BQSR workflow can be executed. To achieve parallelization and thereby shorter runtimes, the number of used threads (`--cores`) can be specified (depends on the given infrastructure):

```
snakemake -n --cores <threads> -s SnakefileBQSR
```

The workflow will create the following directories:

```
13_start_BQSR
14_apply_BQSR
15_analyze_BQSR
16_haplotypeCaller_2
17_gathered_samples_2
18_combined_calls_2
19_genotypeGVCFs_2
20_split_SNPs_Indels_2
21_hard_filtering_2
22_merged_filtered_VCF_2
23_filtered_removed_VCF_2
24_annotated_variants_2
logs/<various_sub_directories>
benchmarks/<various_sub_directories>
```

Optimized workflow execution

The workflow possesses different phases, which can be parallelized to variable degrees. Some rules might even be postponed to phases that cannot be parallelized as much. This helps in optimizing the overall runtime. This can be achieved by the `--prioritize` switch, that assigns highest priority to a given target rule and its direct dependencies:

```
snakemake -n --cores <threads> -s SnakefileBQSR --prioritize genotypeGVCFs_2
```

As above mentioned, the BQSR workflow depends on the previous execution of the normal workflow. It is possible to run both workflows in direct succession. In this case the BQSR Conda environment has to be activated for both workflows. A single command can then be used to execute both workflows in direct succession:

```
snakemake -p --cores <threads>; \
snakemake -p --cores <threads> -s SnakefileBQSR
```

There may not be an unprotected *Enter* between the two commands. It's also possible to write both commands in a single line.

3.9.3 Workflow configuration

The workflow will automatically detect the files, that were generated during the first workflow and further process those. So for the BQSR workflow the `samples_and_read_groups.csv` file is neither needed nor used.

There is one optional configuration file, named `configBQSR.yaml`. A template of this file can be found in the repository. It's intended to configure the internal behavior of the workflow, mainly Java heap size and the number of garbage collection threads. This file is the equivalent to the `config.yaml` used in the first workflow.

```
# yaml file listing optionally available configuration
# options for OVarFlow BQSR
# no option nor the yaml file itself must be present
# here the default options of OVarFlow are listed

ParallelGCThreads:
     BaseRecalibrator  : 2
     ApplyBQSR         : 2
     AnalyzeCovariates : 2
     HaplotypeCaller   : 2
     GatherIntervals   : 2
     CombineGVCFs      : 2
     GATKdefault       : 4

heapSize:
     BaseRecalibrator  : 2
     ApplyBQSR         : 2
     AnalyzeCovariates : 2
     HaplotypeCaller   : 2
     GatherIntervals   : 2
     CombineGVCFs      : 2
     GATKdefault       : 12

Miscellaneous:
     GATKtmpDir        : "GATK_tmp_dir"
     HCnpHMMthreads    : 4
     DebuggingYAML     : False
     DebuggingSAMPLE   : False
```

The above template also documents the default settings used in the BQSR workflow. This workflow uses some additional GATK applications, which can be optimized through this configuration file. The values after the colon can be adjusted as follows:

heapSize *Range of accepted values:* **1 - 40**

> The same general rules apply to this section as mentioned in the *Configuration & adaptation* section under *The yaml file* paragraph, concerning Java heap size.

ParallelGCThreads *Range of accepted values:* **1 - 20**

> The same general rules apply to this section as mentioned in the *Configuration & adaptation* section under *The yaml file* paragraph, concerning Java GC threads.

Miscellaneous *Range of accepted values:*

> *GATKtmpDir:* **/tmp[/subdirectory]** or **./<name>[/subdirectory]**
> *HaplotypeCaller intervals:* **1 - 100**
> *DebuggingYAML:* **False** or **True**

DebuggingSAMPLE: **False** or **True**

In this section the directory used by GATK to store temporary data can be adjusted. The default is to use a directory `GATK_tmp_dir` within the project directory.

The number of *native pair hmm threads* used by GATK HaplotypeCaller can also be adjusted. A value of `1` can increase parallelization, meaning more HaplotypeCaller processes can run in parallel. While a value of `4` will give the quickest execution of the individual HaplotypeCaller process.

A debugging output of the settings made in this YAML file can be enabled, basically echoing the settings made.

Finally a debugging output of the input data that are processed during the analysis can be enabled.

As before the YAML file can be shortened only to those values that shall be changed, see the *Configuration & adaptation* section.

3.9.4 Container usage

As with the "normal workflow", a container is available for the BQSR workflow. This container includes everything that is needed for the entire variant calling: the normal workflow, the BQSR workflow and all the required software. Of course this comes at the cost of a larger container. User that only intend to perform variant calling without BQSR can stick to the smaller container. In any case the usage of a container frees the user from the installation of Conda and creation of a Conda environment. All of this comes with the container.

As before there are plenty of ways to use the container. First of all Docker or Singularity can be used as a container technology. Then there are various ways to execute the workflow with the respective container technology. The following description is focused on Singularity, as it's more straight forward than Docker. Still there are several ways to use Singularity containers. The below list ranges from highly automated to more manual usage. The latter allow for more control of the workflows, if needed.

Automatic start of the workflow

The entire variant calling including BQSR can be started with a single command. But first of all a project directory and a CSV configuration file have to be prepared. Those steps are described in detail under *Conda & Snakemake usage => Preparing OVarFlow / The CSV configuration file*. The workflow is then started via:

```
singularity run --bind /path/to/project_dir/:/input/ OV_BQSR.sif
```

If all preparations were done correctly various log messages will appear starting with:

```
No THREADS variable set. Using default settings.

Using the following number of threads:
  ->_xy_<-
Starting OVarFlow now:

Building DAG of jobs...
...
```

In case that something is still missing or for instance a wrongly named annotation file in the CSV configuration file (here: *SampleSeq.gff* instead of *SampleAnn.gff*), an error message like the following might be shown:

```
Building DAG of jobs...
MissingInputException in line 851 of /snakemake/Snakefile:
Missing input files for rule create_snpEff_db:
```

(continues on next page)

(continued from previous page)

```
REFERENCE_INPUT_DIR/SampleSeq.gff
IndexError in line 12 of /snakemake/SnakefileBQSR:
list index out of range
  File "/snakemake/SnakefileBQSR", line 12, in <module>
```

There is a default value of used threads, equal to the amount of available cores (or threads) minus 4. The setting of an environment variable before starting Singularity enables modification of number of used threads:

```
export THREADS=<desired_number_of_threads>
singularity run --bind /path/to/project_dir/:/input/ OV_BQSR.sif
```

Manual start of the workflows

Also the contents of the container can be made available within a shell. Thereby the snakefiles of the two workflows are directly accessible.

```
user@host:~$ singularity shell --bind /path/to/project_dir/:/input OV_BQSR.sif
Singularity> snakemake -v
5.26.1
Singularity> cd /input
Singularity> snakemake -p --cores <n> -s /snakemake/Snakefile
Singularity> snakemake -p --cores <n> -s /snakemake/SnakefileBQSR
```

3.10 Example & Tutorial

This section is supposed to give a realistic example for the usage of OVarFlow. It will show how to set up and analyze a project. To do so whole genome resequencing data of a chicken (*Gallus gallus*) will serve as sample data. Chicken was chosen for several reasons:

- generally it would be an organism within the scope of OVarFlow
- the genome is of reasonable length with approx. 1 Gb, reducing the analysis time for the example
- a reference genome (GRCg6a) and annotation of reasonable quality are available
- Illumina paired end sequencing data are readily available (e.g. PRJNA291174)

3.10.1 The test data set

A script has been deposited in the OVarFlow repository to automatically download a suitable test data set (`get_chicken_low_coverage.sh`). This script will automatically create the two directories `Fastq` and `Reference` within the working directory and download four Illumina paired end data sets as well as a reference sequence and annotation.

Even with a smaller data set, variant analysis of an entire genome takes several days, which will give a reasonable idea of the duration of variant calling. In case that a rather quick example is desired, a tiny test data set can be created from the full data set. The script `create_mini_data_set.sh` serves this purpose (approx. runtime on a 24 core machine 1 1/4 h). This script has to be executed within the same directory as `get_chicken_low_coverage.sh`. The script requires `samtools`, `bwa` and `bgzip`. All of those programs are available in the OVarFlow Conda Environment.

3.10.2 OVarFlow execution

Now that a sample data set is available you can begin to set up data evaluation with OVarFlow. This example will make use of the full data set, as downloaded with `get_chicken_low_coverage.sh`. Also to keep the installation of all applications very easy, this example will make use of the Singularity container of OVarFlow, but the container will be used in a manual fashion. By opening a shell within the container, the Snakefile can be started manually.

Step 1: Obtaining OVarFlow

First of all a reasonably current version of Singularity has to be installed on the respective system (3.5.1 has been tested, most 3.x versions should do). With Singularity installed the OVarFlow container can be obtained from Docker Hub:

```
singularity build OVarFlow_<tag>.sif docker://ovarflow/release:<tag>
```

The <tag> has to be substituted with the most current version of OVarFlow.

Step 2: Creating a project directory

Create a project directory (`OVarFlow_Chicken`) under a reasonable path on your system and `cd` into this directory:

```
mkdir -p /your/path/OVarFlow_Chicken
cd /your/path/OVarFlow_Chicken
```

Within this project directory you have to create several subdirectories:

```
mkdir FASTQ_INPUT_DIR
mkdir REFERENCE_INPUT_DIR
mkdir OLD_GVCF_FILES
```

The fastq files from the test data set - remember where you downloaded those files - have to be available from the `FASTQ_INPUT_DIR` directory. Therefore those files could be either copied (wastes disk space) or moved to the directory. The creation of links is an alternative, that we will be using here. The same applies for the reference genome and annotation:

```
for fastq in /path/to/test/data/Fastq/*fastq.gz
do
    ln -s "${fastq}" FASTQ_INPUT_DIR/"${fastq##*/}"
done

ln -s /path/to/test/data/Reference/GCF_000002315.6_GRCg6a_genomic.fna.gz REFERENCE_INPUT_
↪DIR/GCF_000002315.6_GRCg6a_genomic.fna.gz
ln -s /path/to/test/data/Reference/GCF_000002315.6_GRCg6a_genomic.gff.gz REFERENCE_INPUT_
↪DIR/GCF_000002315.6_GRCg6a_genomic.gff.gz
```

OVarFlow relies on a naming convention, to recognize paired end sequencing files. Forward reads need the suffix `_R1.fastq.gz` and reverse reads the suffix `_R2.fastq.gz`. SRR fastq files lack this specific suffix. Rename those files accordingly and make sure that the links are not broken:

```
cd FASTQ_INPUT_DIR
mv SRR2131198_1.fastq.gz SRR2131198_R1.fastq.gz
mv SRR2131198_2.fastq.gz SRR2131198_R2.fastq.gz
mv SRR2131199_1.fastq.gz SRR2131199_R1.fastq.gz
mv SRR2131199_2.fastq.gz SRR2131199_R2.fastq.gz
```

(continues on next page)

(continued from previous page)

```
mv SRR2131201_1.fastq.gz SRR2131201_R1.fastq.gz
mv SRR2131201_2.fastq.gz SRR2131201_R2.fastq.gz
mv SRR2131202_1.fastq.gz SRR2131202_R1.fastq.gz
mv SRR2131202_2.fastq.gz SRR2131202_R2.fastq.gz
ls -l
```

Step 3: Adapt the CSV configuration file

With all needed files in place a CSV configuration file has to be created and placed into your data evaluation directory `OVarFlow_Chicken`. An example file to get you started is also available from the OVarFlow repository (`samples_and_read_groups.csv`). Download this file and copy it to your project directory:

```
cp /path/to/file/samples_and_read_groups.csv /your/path/OVarFlow_Chicken/
```

This file has to be edited. You have to enter the fastq files that shall be processed, the reference that shall be used and some read group information. To do this, the CSV file can be opened in a text editor of your liking, or a spread sheet application like LibreOffice Calc. The latter is more convenient for inexperienced users, but make sure to save any changes to the file again in CSV format! For this project the modified file should look like this:

```
Reference Sequence:,GCF_000002315.6_GRCg6a_genomic.fna.gz
Reference Annotation:,GCF_000002315.6_GRCg6a_genomic.gff.gz

Min sequence length:,2000

old gvcf to include:,

forward reads,reverse reads,ID,PL - plattform technology,CN - sequencing center,LB -␣
↪library name,SM - uniq sample name
SRR2131198_R1.fastq.gz,SRR2131198_R2.fastq.gz,id_98,illumina,ENA,lib_98,SRR98
SRR2131199_R1.fastq.gz,SRR2131199_R2.fastq.gz,id_99,illumina,ENA,lib_99,SRR99
SRR2131201_R1.fastq.gz,SRR2131201_R2.fastq.gz,id_201,illumina,ENA,lib_201,SRR201
SRR2131202_R1.fastq.gz,SRR2131202_R2.fastq.gz,id_202,illumina,ENA,lib_202,SRR202
```

The `Min sequence length` has been chosen arbitrarily for this example. In your personal projects choose a value that is reasonable to you. If you don't want to exclude short contigs set the `Min sequence length` to 1.

The read group data are needed by some GATK tools. The `SM - uniq sample name` is probably most important to you. This name will appear within the final annotated results file. Choose a name that identifies the respective individual reasonably well and uniquely. Here a part of the SRR-number was chosen.

Step 4: Activate the OVarFlow Environment

Using the Singularity container saves us from the need to install Conda and setting up a Conda environment. Of course if you lack Singularity you can also manually setup a Conda environment. A shell can easily be opened within the Singularity container. Go to the directory where you issued `singularity build` thereby creating a `.sif` container in that directory (see "Step 1"):

```
singularity shell --bind /path/to/OVarFlow_Chicken:/input OVarFlow_<tag>.sif
bash
cd /input
```

The first command opens a shell within the container and also bind mounts the project directory within the container under the path `/input`. The second command opens another bash shell within the container. By doing so the prompt changes to the Conda base environment (`(base) user@host:~$`). The third command changes into to the project directory. All project files are now available within the container under the path `/input`.

Step 5: Start the OVarFlow workflow

First of all you should perform a dry run of the workflow, to see if every rule will be executed correctly:

```
snakemake -np -s /snakemake/Snakefile
```

If this command succeeds you can start the real data evaluation. The dry run option (`-n`) has to be removed and a reasonable number of threads to archive parallelization has to be given (`--cores <number>`):

```
snakemake -p --cores <number> -s /snakemake/Snakefile
```

Step 6: Lean back

If everything was set up correctly OVarFlow will now take care of the variant calling.

3.10.3 Resource usage

Within the above example a rather small project was processed. Still, when considering the provided input data, it should be possible to get a rough idea of the resource requirements even for larger projects.

The reference genome GRCg6a total length (Mb): 1065.37

The input data

8 fastq files (for & rev): 45 Gb
low coverage: 10 - 15 fold
(30 fold is desirable in real projects)

The finished project directory size of all output data: 217 Gb

01_mapping: 55 Gb
02_sort_gatk: 55 Gb
03_mark_duplicates: 55 Gb
04_haplotypeCaller: 15 Gb
05_gather_samples: 15 Gb
06_combined_calls: 14 Gb
07_genotypeGVCFs: 1022 Mb
08_split_SNPs_Indels: 1035 Mb
09_hard_filtering: 1037 Mb
10_merged_filtered_VCF: 1024 Mb
11_filtered_removed_VCF: 986 Mb
12_annotated_variants: 1334 Mb

Total runtime to compute computations were performed on a 28 core machine with the following command: `snakemake -p --cores 26 -s ./Snakefile`

total runtime: 37 h 36 min

As variant calling on eukaryotic genomes is a computationally demanding task, runtimes of days are the norm and can even extend to weeks for larger projects. The actual time to finish for a real project will depend on several factors:

- the genome size of the organism
- the sequencing depth
- the number of individuals to analyze (e.g. the number of fastq files)
- the degree of variability of the organism (more variants mean longer runtimes)
- the given computing resources

For the above example the following hardware was used:

Architecture:	x86_64
CPU(s):	28
Thread(s) per core:	1
Model name:	Intel Core Processor (Broadwell)
CPU MHz:	2593.906
Hypervisor vendor:	KVM
Virtualization type:	full
Flags:	... avx ... (this is important)
Main memory:	64 Gb

3.11 Resource requirements

Variant calling is a computationally demanding task. There are three main hardware factors, that can create a bottleneck in the data analysis, being (1) the number and speed of central processing units (**CPUs**) or cores to be precise, (2) amount of **main memory** and (3) **disk space**. No single one of those components can be considered individually. When increasing the number of CPUs memory size might become a new bottleneck. Of course the exact hardware requirements depend on the size of your respective project and acceptable waiting time for the calculations to finish. This is influenced by various factors:

- the genome size of the organism under study,
- the frequency of genomic variants,
- the sequencing depth and
- of course the number of individuals that shall be analyzed.

OVarFlow has been designed to be able to adopt to various project sizes and is able to scale with the provided computational resources. When ever possible various tasks are executed in parallel. Especially the GATK HaplotypeCaller, which is probably the biggest single bottleneck, has been optimized for parallelization. The higher the number of intervals, that are specified by the user, the higher the parallelization degree of the HaplotypeCaller.

Of course the total resource requirements depend on the resource usage of the single applications that are used in OVarFlow. Therefore the resource usage of individual applications has been investigated closely, with a focus on CPU and memory usage. The following sections will document the obtained results and applied optimizations.

3.12 Benchmarking & Optimizations

Gallus gallus (chicken) has been used as a test organism. Not only is its reference genome GRCg6a of reasonable quality but it's also of moderate size, with approx. 1.07 Gbp. The exact file versions used are:

- Reference genome: GCF_000002315.6_GRCg6a_genomic.fna.gz
- Reference annotation: GCF_000002315.6_GRCg6a_genomic.gff.gz

Whole genome sequencing (wgs) data were obtained from the European Nucleotide Archive (ENA), which offers direct download of fastq files. The study PRJNA306389 offers wgs data of varying sequencing depth. Two sequencing data sets of different sequencing depths were chosen:

- Run: SRR3041137
  Base count: 38,136,658,250, average coverage after mapping: 34-fold
- Run: SRR3041413
  Base count: 18,799,906,500, average coverage after mapping: 16-fold

All calculations were performed on a virtual computer provided by the German Network for Bioinformatics Infrastructure (de.NBI) (de.NBI cloud location at the Justus-Liebig-University Gießen). The virtual machine offered 28 cores and 64 GB main memory.

Exact CPU specification as provided by `lscpu`:

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              28
On-line CPU(s) list: 0-27
Thread(s) per core:  1
Core(s) per socket:  1
Socket(s):           28
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               61
Model name:          Intel Core Processor (Broadwell)
Stepping:            2
CPU MHz:             2593.906
BogoMIPS:            5187.81
Hypervisor vendor:   KVM
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            4096K
L3 cache:            16384K
NUMA node0 CPU(s):   0-27
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat␣
↪pse36 clflush mmx fxsr sse sse2 syscall nx rdtscp lm constant_tsc rep_good nopl␣
↪xtopology cpuid tsc_known_freq pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic␣
↪movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm␣
↪3dnowprefetch invpcid_single pti fsgsbase bmi1 hle avx2 smep bmi2 erms invpcid rtm␣
↪rdseed adx smap xsaveopt arat
```

The used software version as provided by `gatk --version` was:

```
The Genome Analysis Toolkit (GATK) v4.1.7.0
HTSJDK Version: 2.21.2
Picard Version: 2.21.9
```

The used Java version as provided by `java -version` was:

```
openjdk version "1.8.0_152-release"
OpenJDK Runtime Environment (build 1.8.0_152-release-1056-b12)
OpenJDK 64-Bit Server VM (build 25.152-b12, mixed mode)
```

The used version of bwa as provided by `bwa` was:

```
Program: bwa (alignment via Burrows-Wheeler transformation)
Version: 0.7.17-r1188
```

The used version of samtools as provided by `samtools --version` was:

```
samtools 1.10
Using htslib 1.10.2
Copyright (C) 2019 Genome Research Ltd.
```

Resource usage of a specific process was monitored every 3 seconds via the command:

```
ps -p <pid of process> -o rss,%mem,%cpu | tail -1
```

Further code details can be found within the repository of OVarFlow. No additional demanding computations were performed during the recording of the resource usage.

3.12.1 Java Garbage Collection

The CPU usage of some GATK tools is heavily affected by the Java Garbage Collection (GC). The Java HotSpot VM offers three different garbage collectors. The *parallel collector* is the default on larger hardware (Java 8 documentation), as used in variant calling. As the name implies the parallel collector uses multithreading to accelerate garbage collection. The number of threads used, depends on the available amount of threads of the respective machine. The documentation describes:

> *On a machine with N hardware threads where N is greater than 8, the parallel collector uses a fixed fraction of N as the number of garbage collector threads. The fraction is approximately 5/8 for large values of N. At values of N below 8, the number used is N. On selected platforms, the fraction drops to 5/16.*

This is important, as the actual number of used GC threads can have an enormous impact on the CPU time consumed by some GATK tools. The actual amount of used GC threads can be determined via the command:

```
java -XX:+PrintFlagsFinal | grep ParallelGCThreads
```

On various different machines (core count by *lscpu*) the following values were obtained:

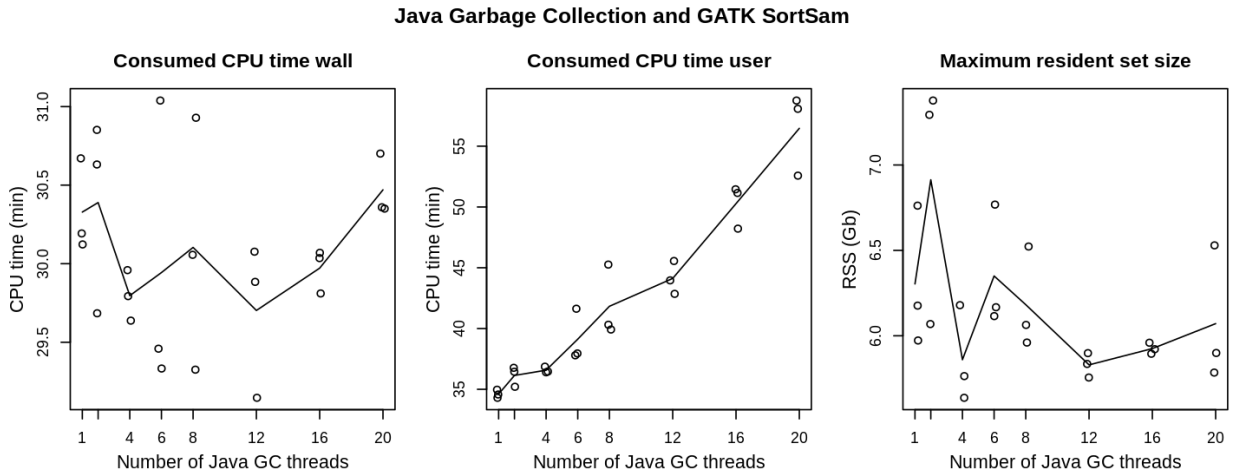
CPU Cores	GC Threads
8	8
28	20
64	43
160	103

To determine the effect of GC on GATK, several GATK commands were executed with different GC settings (1, 2, 4, 6, 8, 12, 16 and 20 GC threads) and their consumed CPU time (wall time and user time) as well as maximum memory consumption (resident set size - RSS) was measured via *GNU time* (1.8, version 1.7 includes a bug resulting in four times to high values for RSS). Each measurement was repeated three times and the resulting mean values were plotted. Depending on the computation times of the respective GATK tool, different data sets were used (SRR3041116, SRR3041413 and SRR30411137). For GATK HaplotypeCaller only an interval was used. This was done to reduce waiting times. Of course for every single analysis the same input data were used. Finally only relative changes within a single command due to Java GC are of interest here, not absolute changes due to different file sizes. The provided commands specify the used data set.

Effect on GATK SortSam

The following command was used to determine the effects of Java GC on GATK SortSam:

```
FILE=SRR3041137
/usr/bin/time -o ${LOG_FILE} --append -v \
gatk --java-options -XX:ParallelGCThreads=${GC} SortSam \
   -I 01_mapping/${FILE}.bam \
   -SO coordinate \
   -O ${TMP_DIR}/${FILE}.bam \
   --TMP_DIR ./GATK_tmp_dir/ 2> ${TMP_DIR}/02_sort_gatk_${FILE}.log
```



When it comes to wall time sorting of bam files is barely influenced by the number of Java GC threads. Considering the multithreaded load on several cores, as is done by the user measurement, the consumed CPU time rises approx. proportional with the number of threads. There is no obvious influence of the Java GC on memory consumption. For GATK SortSam **one or two Java GC threads** give the best performance.

Effect on GATK MarkDuplicates

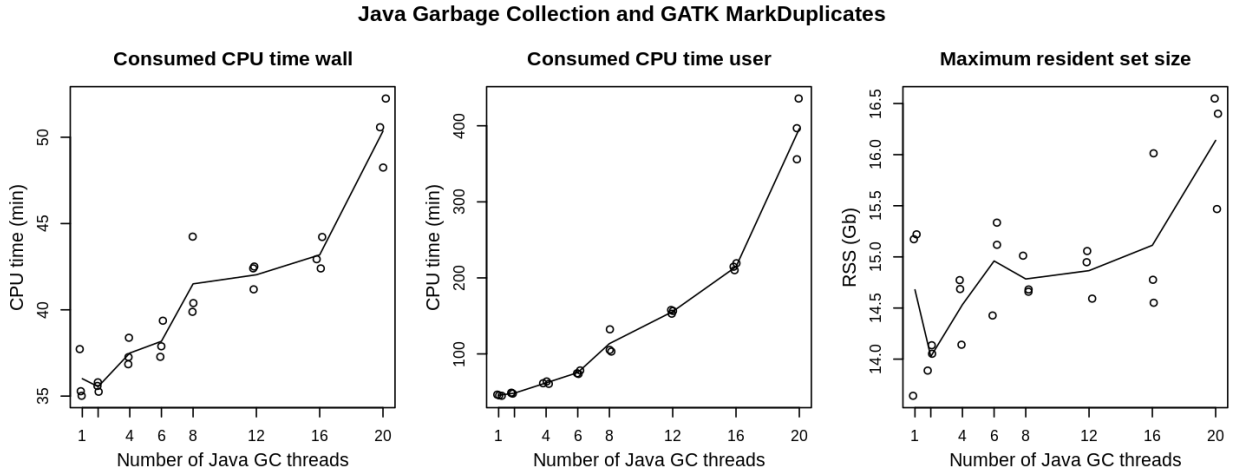
The following command was used to determine the effects of Java GC on GATK MarkDuplicates:

```
FILE=SRR3041413
/usr/bin/time -o ${LOG_FILE} --append -v \
gatk --java-options -XX:ParallelGCThreads=${GC} MarkDuplicates \
   -I 02_sort_gatk/${FILE}.bam \
   -O ${TMP_DIR}/03_mark_duplicates_${FILE}.bam \
```

(continues on next page)

(continued from previous page)

```
   -M ${TMP_DIR}/03_mark_duplicates_${FILE}.txt \
   -MAX_FILE_HANDLES 300 \
   --TMP_DIR ./GATK_tmp_dir/ 2> ${TMP_DIR}/03_mark_duplicates_${FILE}.log
```

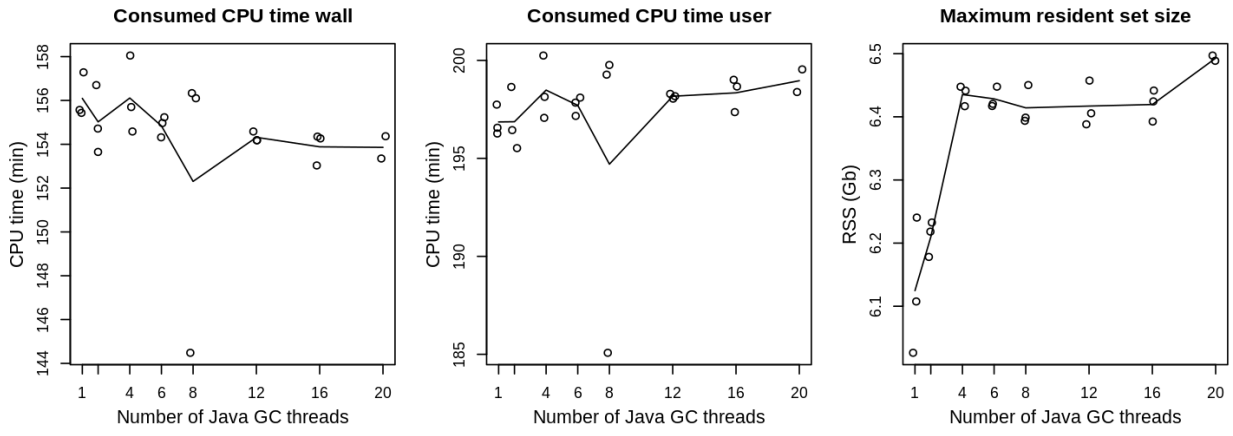


Default settings of 20 GC threads cause the highest CPU loads, both for wall and user time. This is especially important for the total consumed CPU time (user measurement), which is more than **seven times higher** for 20 GC threads as compared to 1 or 2 GC threads. Also memory-wise a preference for lower thread counts might be favorable. Considering all three measurements, the optimum for GATK MarkDuplicates seems to be given with **two Java GC threads**.

Effect on GATK HaplotypeCaller

The following command was used to determine the effects of Java GC on GATK HaplotypeCaller:

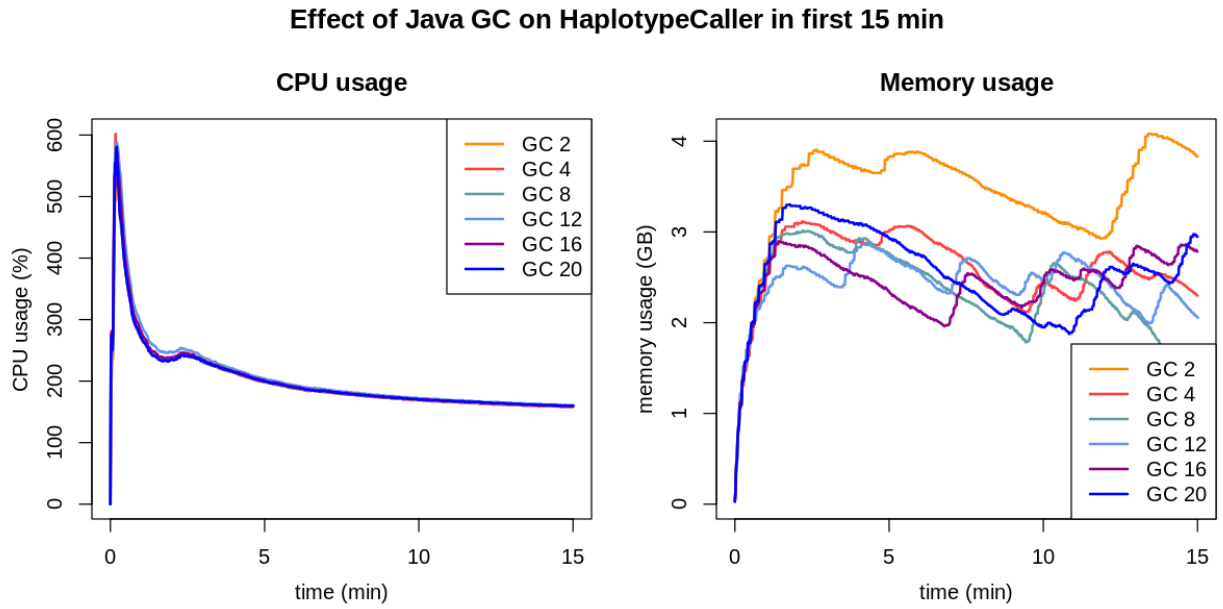
```
FILE=SRR3041413
/usr/bin/time -o ${LOG_FILE} --append -v \
gatk --java-options -XX:ParallelGCThreads=${GC} HaplotypeCaller \
   -ERC GVCF -I 03_mark_duplicates/${FILE}.bam \
   -R processed_reference/GCF_000002315.6_GRCg6a_genomic.fa.gz \
   -O ${TMP_DIR}/${FILE}_tmp.gvcf.gz \
   -L "NC_006093.5" 2> ${TMP_DIR}/${FILE}_tmp.log
```



The amount of consumed CPU time is considerably less dependent on the GC settings than it has been the case for GATK SortSam and MarkDuplicates. The absolute timescale only shows statistical fluctuations. Therefore CPU load of HaplotypeCaller is barely affected by Java GC settings. From the given measurements, maximum memory usage (resident set size) appears to be favourable at **one or two Java GC threads**.

As the HaplotyeCaller is the application with the longest runtimes in OVarFlow, and peak CPU loads of this application were noticed at the beginning of program execution, its CPU and memory usage was investigated more closely. Over a period of 15 min CPU and RSS were measured every second using `ps -p <pid> -o rss,%mem,%cpu` and graphs were plotted for various Java GC settings.

```
FILE=SRR3041137
gatk --java-options -XX:ParallelGCThreads=${GC} HaplotypeCaller \
  -ERC GVCF -I 03_mark_duplicates/${FILE}.bam \
  -R processed_reference/GCF_000002315.6_GRCg6a_genomic.fa.gz \
  -O ${MON_DIR}/${FILE}.gvcf.gz \
  -L "NC_006093.5"  2> ${MON_DIR}/${FILE}.log &
```



Graphs of CPU usage are congruent for all Java GC settings. The peak load at the beginning makes use of six threads (600 % CPU load) and is totally independent of Java GC thread count. Such load peaks were also observed for other GATK tools (see the section concerning file size or sequencing depth, respectively). When it comes to memory, two GC threads caused a higher usage. Still this observation only applies to the first 15 min (see previous graphics).

Effect on GATK GatherVcfs

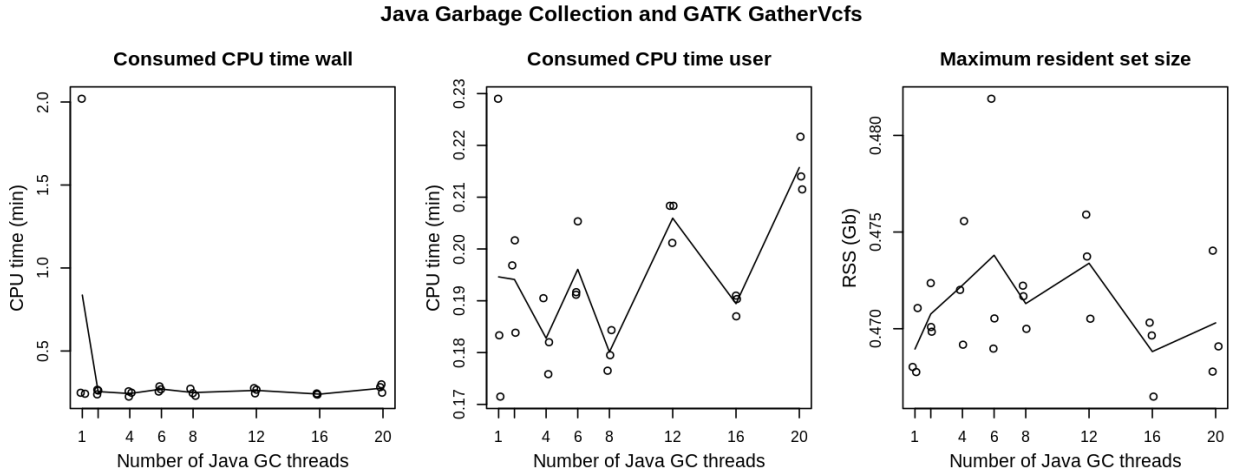
The following command was used to determine the effects of Java GC on GATK GatherVcfs:

```
FILE=SRR3041413
/usr/bin/time -o ${LOG_FILE} --append -v \
gatk --java-options -XX:ParallelGCThreads=${GC} GatherVcfs \
   -O ${DIR}/05_gathered_samples_${FILE}.gvcf.gz \
   -I 04_haplotypeCaller/${FILE}/interval_1.g.vcf.gz \
   -I 04_haplotypeCaller/${FILE}/interval_2.g.vcf.gz \
   -I 04_haplotypeCaller/${FILE}/interval_3.g.vcf.gz \
```

(continues on next page)

(continued from previous page)

```
   -I 04_haplotypeCaller/${FILE}/interval_4.g.vcf.gz \
   --TMP_DIR ./GATK_tmp_dir 2> ${MON_DIR}/05_gathered_samples_${FILE}.log
```

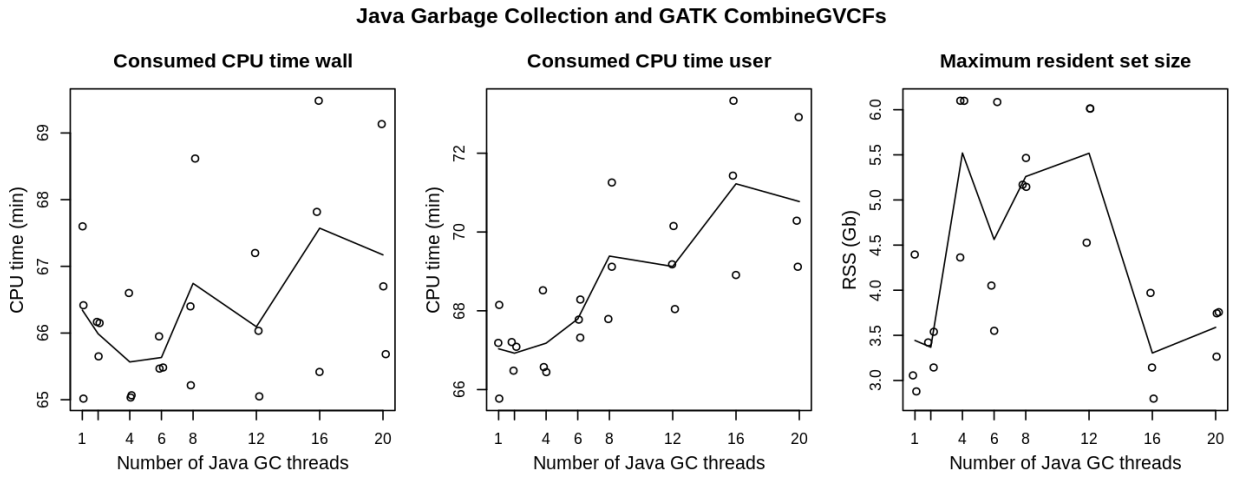


GatherVcfs is not noticeably influenced by the number Java GC threads. Only wall time of the first measurement is considerably higher (approx. 2 min). This is due to page caching of the processed data, which are kept in memory after they are fist accessed. For the first measurement data have to be obtained from permanent memory first and are thereby stored in memory for the next measurements. GatherVcfs was configured to use **two Java GC threads**.

Deprecated: Effect on GATK CombineGVCFs

CombineGVCFs was substituted with GatherVcfs, which is more efficient. This section is only for reference purposes.

```
FILE=SRR3041413
/usr/bin/time -o ${LOG_FILE} --append -v \
gatk --java-options -XX:ParallelGCThreads=${GC} CombineGVCFs \
   -O ${TMP_DIR}/${FILE}_tmp.gvcf.gz \
   -R processed_reference/GCF_000002315.6_GRCg6a_genomic.fa.gz \
   -V 04_haplotypeCaller/${FILE}/interval_2.gvcf.gz \
   -V 04_haplotypeCaller/${FILE}/interval_4.gvcf.gz \
   -V 04_haplotypeCaller/${FILE}/interval_1.gvcf.gz \
   -V 04_haplotypeCaller/${FILE}/interval_3.gvcf.gz 2> ${TMP_DIR}/${FILE}_tmp.log
```



For GATK CombineGVCFs the impact of the number of Java GC threads only show a moderate effect, which is even covered by statistic variance between the measurements. Wall time is only slight light affected, were the number of negative outliers might be reduced for lower thread counts. The situation is a bit more clear for the user time, where lower thread counts are clearly favourable, but only by a few percent of the total run time. For memory usage the range is much wider (approx. 3 to 6 Gb). A constant that could be seen also in other measurements (not show) was a low and less varying memory consumption when using 2 Java GC threads. Using **two Java GC threads** seem to be favorable for GATK CombineGVCFs.

OVarFlow and Java GC

Interestingly not every GATK tool behaves identical. Still if there is a preference, it has always been observed in favour of low Java GC thread numbers. Some tools, like SortSam, only show a clear tendency in one of the observed parameters (in this case total CPU time). For CombineGVCFs on the other hand the tendency is not as pronounced as for SortSam or MarkDuplicates. Still there is a preference for low Java GC thread numbers.

As can be seen from the above measurements, choosing the optimal number of Java GC threads can have an enormous effect on resource usage. The obtained results were incorporated into OVarFlow, with the following settings for `ParallelGCThreads`:

- GATK SortSam: 2
- GATK MarkDuplicates: 2
- GATK HaplotypeCaller: 2
- GATK GatherVcfs: 2
- GATK CombineGVCFs: 2
- other GATK applications: 4

This is consistent with a block post in the GATK forum (date of post Oct 2017; posted during transition from GATK 3 to 4, seemingly valid for both versions):

> *You would be better off setting it [Java GC thread count] to 2-4 threads. Performance gets worse beyond that typically from what the developers have seen.*

3.12.2 Java Heap Space (-Xmx)

Global settings of the Java virtual machine (JVM) can cause major performance impacts on the respective GATK tool. In this regard Java Garbage Collection (GC) is only one aspect. Settings of the Java heap size cause a major influence on memory consumption of the JVM. Here two values affect heap size, as can be shown via `java -X`:

```
...
-Xms<size>        set initial Java heap size
-Xmx<size>        set maximum Java heap size
...
```

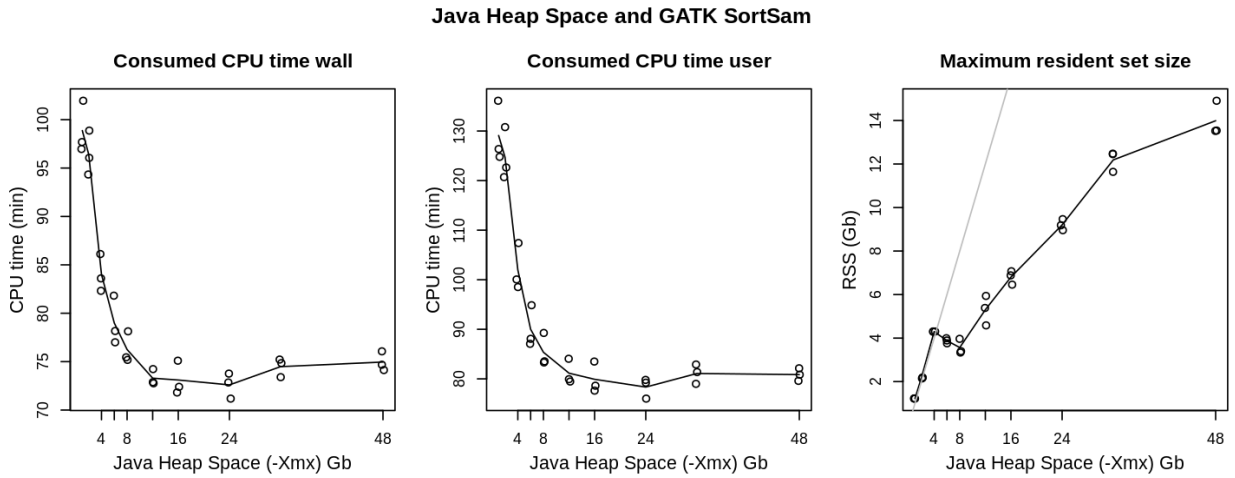
When starting the JVM `-Xms` is not set (and is not as important), but the values of `-Xmx` will be set depending on the given amount of memory the respective machine has to offer. The values of a certain machine can be determined via `java -XshowSettings:vm 2>&1 | head`. On various different machines the following values were obtained:

Memory	Max. Heap Size (Estimated)
16 Gb	3.48 Gb
64 Gb	13.98 Gb
256 Gb	26.67 Gb
512 Gb	26.67 Gb
1 Tb	26.67 Gb

Again just like with the number of Java GC threads, there is a situation were the default behavior is dependent upon the respective machine parameters. Finally heap size can have considerable effects on runtimes and obviously even more on memory usage. Therefore those GATK tools that work in parallel on several files were also monitored for various predefined heap sizes (1, 2, 4, 6, 8, 12, 16, 24, 32 and 48 Gb). Besides performance impacts too small values for the heap size will result in the lack of memory and can result in an java.lang.OutOfMemoryError.

Effect on GATK SortSam

```
FILE=SRR3041137; GC=2
/usr/bin/time -o ${LOG_FILE} --append -v \
gatk --java-options "-Xmx${XMX}G -XX:ParallelGCThreads=${GC}" SortSam \
   -I 01_mapping/${FILE}.bam \
   -SO coordinate \
   -O ${DIR}/02_sort_gatk_${FILE}.bam \
   --TMP_DIR ./GATK_tmp_dir/ 2> ${DIR}/02_sort_gatk_${FILE}.log
```

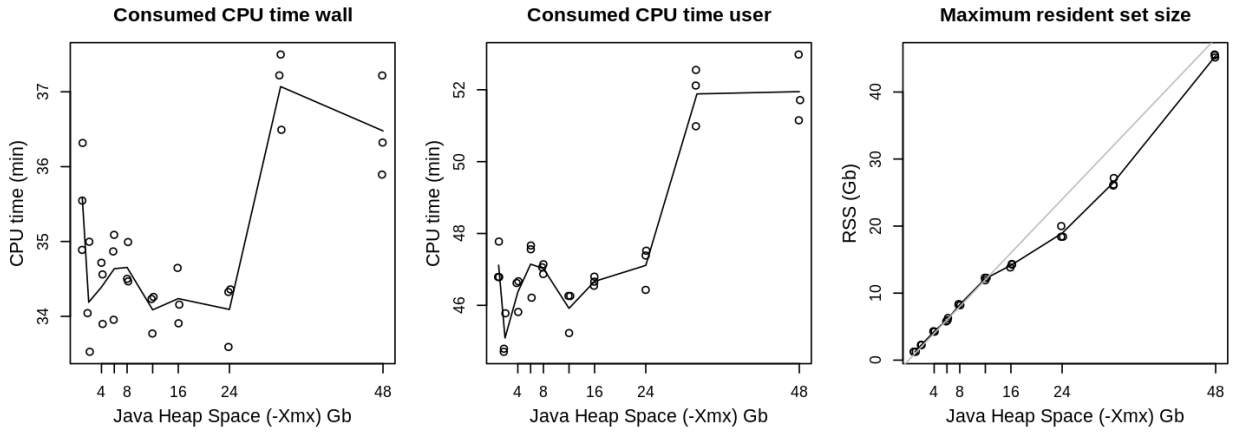


CPU usage is negatively effected by low heap sizes, reaching a sustainable minimum at approx. 12 Gb. Generally memory usage raises with higher values for the Xmx setting, but with a drop at 8 Gb. The gray line in the RSS plot indicates parity between measured RSS and set Xmx values (meaning RSS = Xmx). It is obvious that CPU and memory usage cannot be minimized at the same time. Still simultaneous optimization of both parameters is possible with Xmx settings of 8 or 12 Gb. OVarFlow was set at **10 Gb for SortSam**.

On a side note: setting identical values for Xms and Xmx did not result in higher memory usage. Even with higher Xms values memory will be initialized with a 0-page. But memory is only counted as RSS, when it is actually accessed and written to.

Effect on GATK MarkDuplicates

```
FILE=SRR3041413; GC=2
/usr/bin/time -o ${LOG_FILE} --append -v \
gatk --java-options "-Xmx${XMX}G -XX:ParallelGCThreads=${GC}" MarkDuplicates \
   -I 02_sort_gatk/${FILE}.bam \
   -O ${DIR}/03_mark_duplicates_${FILE}.bam \
   -M ${DIR}/03_mark_duplicates_${FILE}.txt \
   -MAX_FILE_HANDLES 300 \
   --TMP_DIR ./GATK_tmp_dir/ 2> ${DIR}/03_mark_duplicates_${FILE}.log
```

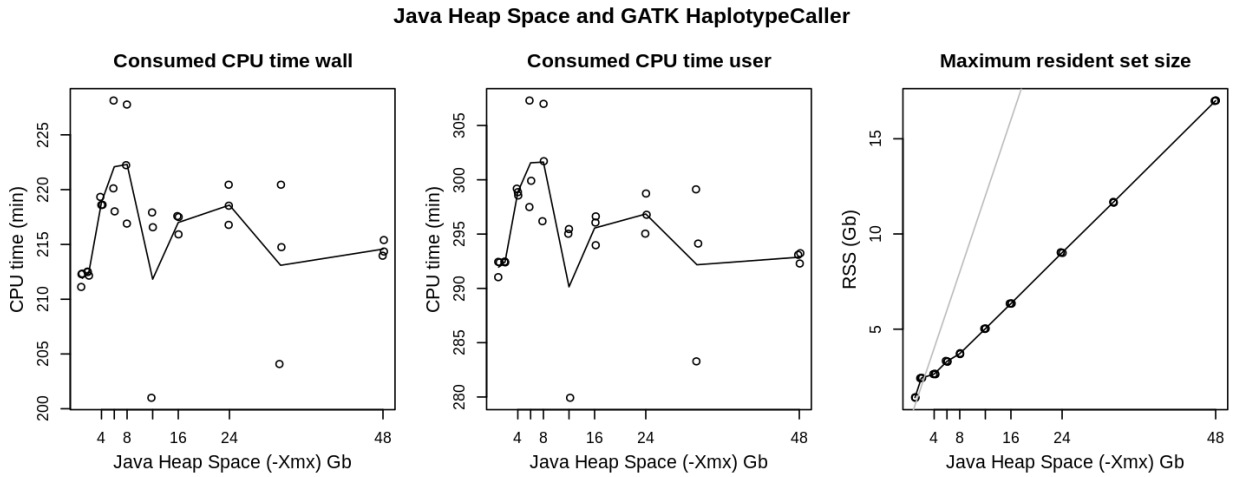


From 1 to 24 Gb Xmx settings, CPU usage is not noticeably affected. Only 32 and 48 Gb were moderately more

demanding. Memory usage on the other hand raises nearly linear with Xmx settings. Lower heap size are clearly preferable for MarkDuplicates and were set to **2 Gb**.

Effect on GATK HaplotypeCaller

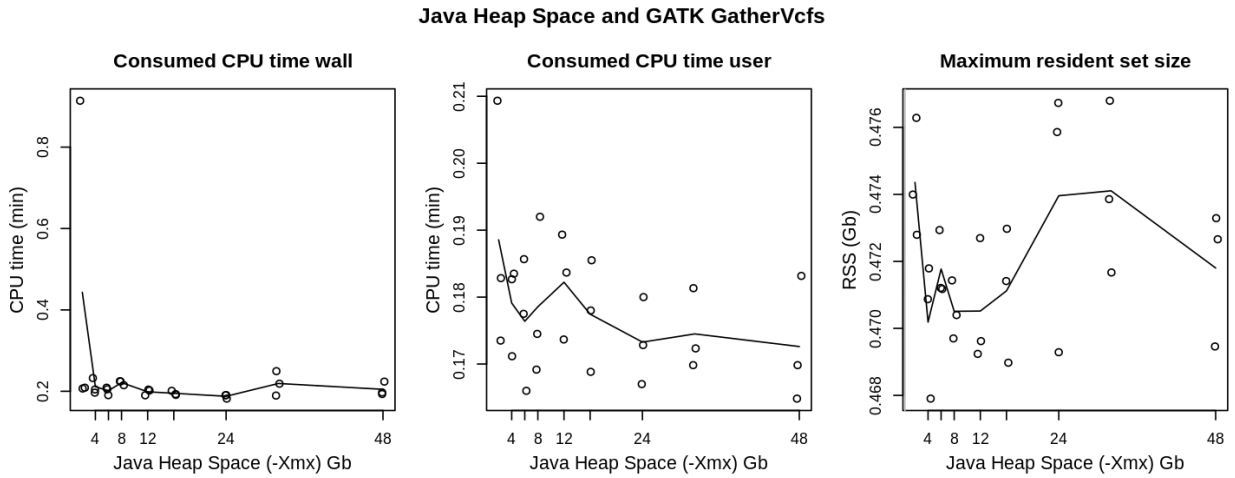
```
FILE=SRR3041137; GC=2
/usr/bin/time -o ${LOG_FILE} --append -v \
gatk --java-options "-Xmx${XMX}G -XX:ParallelGCThreads=${GC}" HaplotypeCaller \
   -ERC GVCF -I 03_mark_duplicates/${FILE}.bam \
   -R processed_reference/GCF_000002315.6_GRCg6a_genomic.fa.gz \
   -O ${DIR}/${FILE}_tmp.gvcf.gz \
   -L "NC_006093.5" 2> ${DIR}/${FILE}_tmp.log
```



CPU usage of HaplotypeCaller is not effected by different Java heap sizes. Again there is a near linear relation between Xmx settings and actual memory usage, but starting from 4 Gb memory usage stays way below the allowed heap sizes. HaplotypeCaller was set to use **2 Gb** memory for the Java heap size.

Effect on GATK GatherVcfs

```
FILE=SRR3041137; GC=2
/usr/bin/time -o ${LOG_FILE} --append -v\
gatk --java-options "-Xmx${XMX}G -XX:ParallelGCThreads=${GC} GatherVcfs \
   -O ${DIR}/05_gathered_samples_${FILE}.gvcf.gz \
   -I 04_haplotypeCaller/${FILE}/interval_1.g.vcf.gz \
   -I 04_haplotypeCaller/${FILE}/interval_2.g.vcf.gz \
   -I 04_haplotypeCaller/${FILE}/interval_3.g.vcf.gz \
   -I 04_haplotypeCaller/${FILE}/interval_4.g.vcf.gz \
   --TMP_DIR ./GATK_tmp_dir 2> ${DIR}/05_gathered_samples_${FILE}.log
```

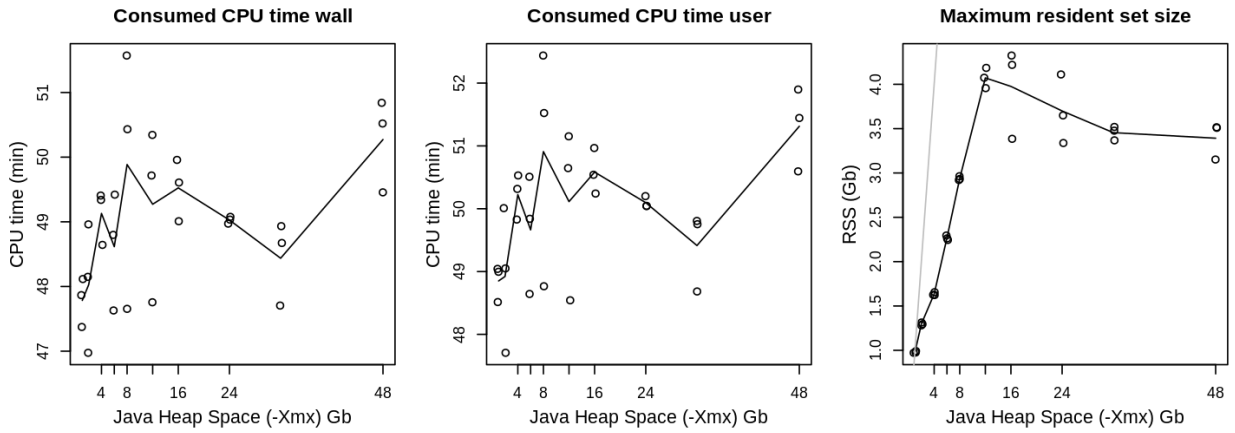


GatherVcfs is not significantly influenced by Java heap size settings. Only wall time of the first measurement is considerably higher. This is due to page caching of the processed data, which are kept in memory after they are first accessed. Also overall resource usage is very moderate and a significant advantage over CombineGVCFs, which was previously employed for this step. To allow for some resource tolerance heap size was set to **2 Gb**.

Deprecated: Effect on GATK CombineGVCFs

CombineGVCFs was replaced by GatherVcfs.

```
FILE=SRR3041137; GC=2
/usr/bin/time -o ${LOG_FILE} --append -v \
gatk --java-options "-Xmx${XMX}G -XX:ParallelGCThreads=${GC}" CombineGVCFs \
   -O ${DIR}/05_gathered_samples_${FILE}.gvcf.gz \
   -R processed_reference/GCF_000002315.6_GRCg6a_genomic.fa.gz \
   -V 04_haplotypeCaller/${FILE}/interval_2.gvcf.gz \
   -V 04_haplotypeCaller/${FILE}/interval_4.gvcf.gz \
   -V 04_haplotypeCaller/${FILE}/interval_1.gvcf.gz \
   -V 04_haplotypeCaller/${FILE}/interval_3.gvcf.gz 2> ${DIR}/05_gathered_samples_${FILE}
↪.log
```



If there is a clear effect on CPU usage of the allowed heap size on CombineGVCFs it is mostly hidden under statistic variance. On the other hand effects on RSS values are rising from 1 to 12 Gb, where a maximum is reached. Java heap

size of CombineGVCFs was set to **2 Gb**.

OVarFlow and Java heap size

Overall CPU usage is only barely affected by different heap sizes. Only SortSam is an exception, were low heap sizes will significantly increase runtime. As expected, lower heap size settings (Xmx) are favorable to save some memory (RSS). Still some interesting drops in memory usage could be observed for some Xmx values.

To maximize performance while minimizing resource usage of OVarFlow the following values for the heap size (`-Xmx<n>G` were set within the Snakefile:

- GATK SortSam: 10 Gb
- GATK MarkDuplicates: 2 Gb
- GATK HaplotypeCaller: 2 Gb
- GATK GatherVcfs: 2 Gb
- GATK CombineGVCFs: 2 Gb

By manually specification of a Java heap size, memory usage of the GATK tools could clearly be improved over the default values that applied to a machine with 64 Gb main memory.

3.12.3 File size / sequencing depth

The previous sections investigated the effects of different Java GC thread numbers (`-XX:ParallelGCThreads`) and various Java heap sizes (`-Xmx`) on the resource usage (CPU and memory) of several GATK tools. This served the identification of optimized Java GC settings and heap sizes. Optimized Java GC settings were then applied to OVarFlow.

This section now focuses on:

- Resource usage over the complete run time of the single GATK tools with optimized Java GC and heap size settings.
- The influence of sequencing depth and file size on the resource usage.

Therefore profiles of CPU and memory usage (RSS) were recorded for different input data. In doing so, this section also provides clues about the resources that are required by the single GATK tools and ultimately the entire workflow.

Again the previously utilized data sets were used to evaluate resource usage:

SRR3041137 34 x average coverage after mapping, 18.7 Gb, 2 x 125 bp

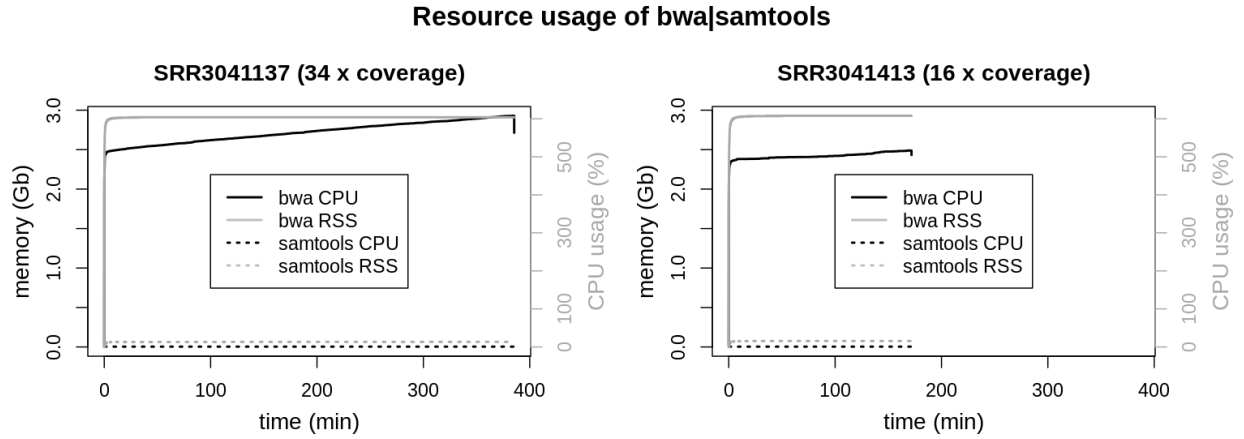
SRR3041413 16 x average coverage after mapping, 10.8 Gb, 2 x 150 bp

Those data sets were suitable to cover common different properties of the sequencing files, that could cause a peak load on CPU or memory. To account for statistical deviations in resource usage, every data set was evaluated twice (solid and dashed plots). Resource usage was monitored every three seconds using `ps -p <pid> -o rss,%mem,%cpu`. The shell script used for monitoring can be found within the repository of OVarFlow alongside the R script used to generate the final graphs.

bwa mem | samtools

The principle command use to evaluate the respective data set is given below:

```
File=<SRR3041137|SRR3041413>
bwa mem -M -t 6 \
  -R "@RG\tID:id_${FILE}\tPL:illumina\tPU:dummy\tCN:SRR\tLB:lib_${FILE}\tSM:${FILE}" \
  processed_reference/GCF_000002315.6_GRCg6a_genomic.fa.gz \
  FASTQ_INPUT_DIR/${FILE}_R1.fastq.gz FASTQ_INPUT_DIR/${FILE}_R2.fastq.gz \
  2> ${MON_DIR}/${FILE}_stderr_bwa.log | \
samtools view -b /dev/fd/0 -o ${MON_DIR}/${FILE}.bam \
  2> ${MON_DIR}/${FILE}_stdout_samtools.log
```

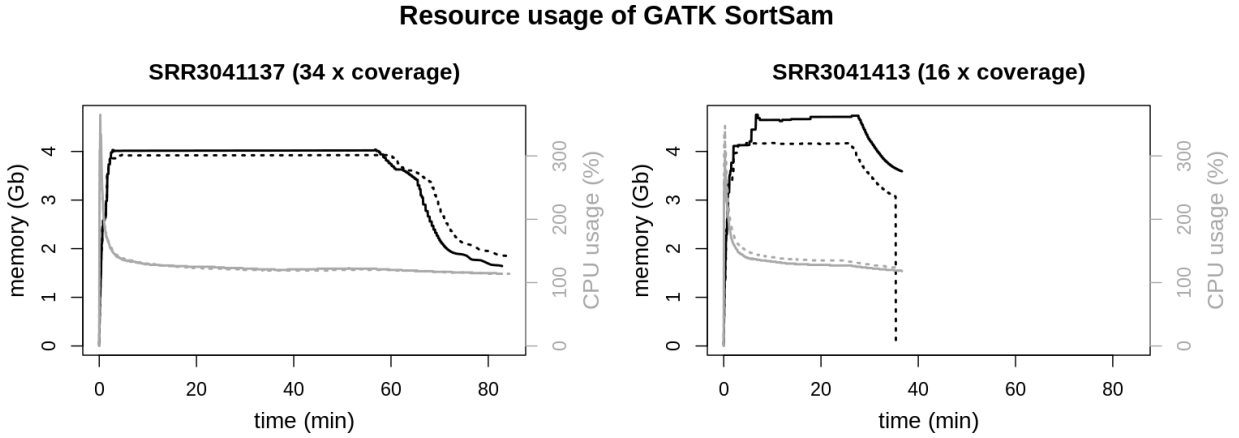


For the `bwa mem | sortsam` pipeline only one measurement is shown. Graphs of the second measurement were congruent and could not be distinguished from the first measurement. Resource usage of the two piped tools (`bwa mem` and `samtools view`) was recorded separately.

Resource usage of the compression from sam to bam format by `samtools view` can be neglected compared to bwa mem. As `bwa mem` was configured to use 6 threads, the CPU load is very consistent at 600 %. Interestingly memory usage of `bwa mem` is increasing linearly over the whole runtime of the process. RSS maxes out at 3 Gb for the larger data set and only reaches 2.5 Gb for the smaller one. To account for larger genomes and even higher coverage data sets, the maximum memory usage was set to 4 Gb within the Snakefile.

GATK SortSam

```
GC=2; Xmx=10; File=<SRR3041137|SRR3041413>
gatk --java-options "-Xmx${Xmx}G -XX:ParallelGCThreads=${GC}" SortSam \
  -I 01_mapping/${FILE}.bam \
  -SO coordinate \
  -O ${MON_DIR}/${FILE}.bam \
  --TMP_DIR ./GATK_tmp_dir/ 2> ${MON_DIR}/02_sort_gatk_${FILE}.log &
```



The dashed graph for SRR3041413 shows a steep decline, whereas the other graphs show a sudden end. This is an artifact of the measurement and not a systematic difference. Single measurements were recorded every three seconds. Most of the time the respective command just finished within this three second interval, allowing for no additional measurement. Sometimes a measurement was taken just while a process was freeing resource, recording this decline in resource usage.

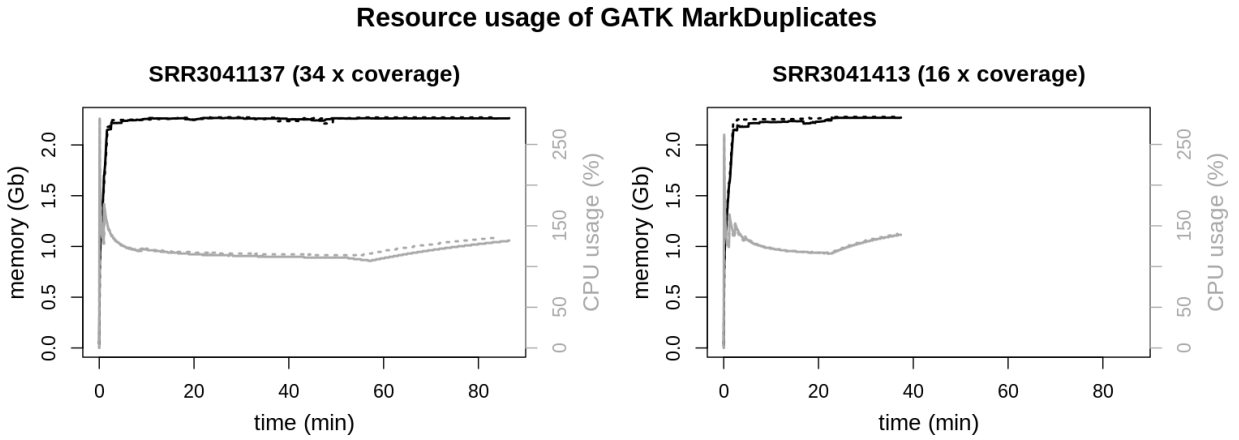
Total CPU and memory usage are barely altered by sequencing depth or file size. Memory consumption reaches a plateau at approx. 4 Gb. There is a sharp peak at the beginning of the runtime for the CPU usage, dropping quickly to slightly above 100 % (meaning a little over a single thread is used). The clearest effect is on total runtimes. The larger file needs longer to process.

Aside from the `-Xmx` parameter, the amount of memory that is used by SortSam can be modified by setting the option `--MAX_RECORDS_IN_RAM`, which will inversely increase the number of file handles. The tool documentation of SortSam states:

> *When writing files that need to be sorted, this will specify the number of records stored in RAM before spilling to disk. Increasing this number reduces the number of file handles needed to sort the file, and increases the amount of RAM needed.*

GATK MarkDuplicates

```
GC=2; Xmx=2; File=<SRR3041137|SRR3041413>
gatk --java-options "-Xmx${Xmx}G -XX:ParallelGCThreads=${GC}" MarkDuplicates \
  -I 02_sort_gatk/${FILE}.bam \
  -O ${MON_DIR}/03_mark_duplicates_${FILE}.bam \
  -M ${MON_DIR}/03_mark_duplicates_${FILE}.txt \
  -MAX_FILE_HANDLES 300 \
  --TMP_DIR ./GATK_tmp_dir/ 2> ${MON_DIR}/03_mark_duplicates_${FILE}.log &
```

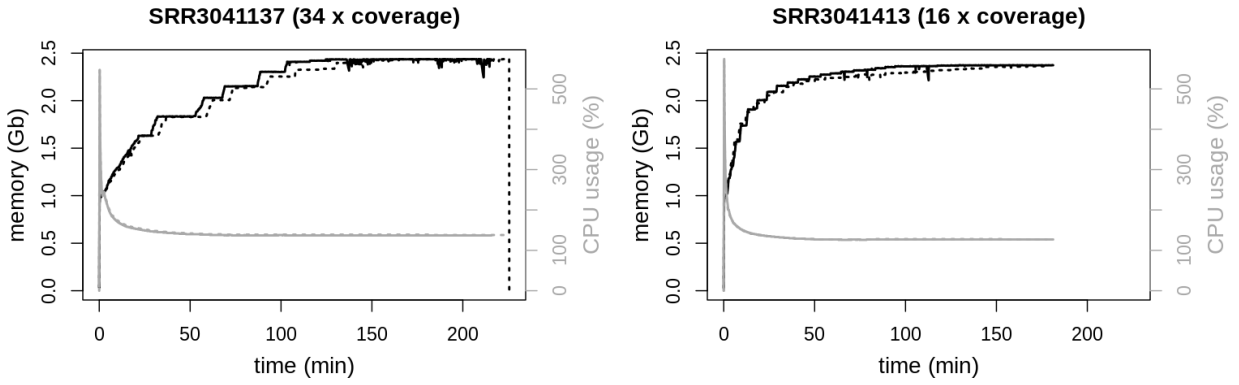


Both data sets show similar resource usage. Memory usage maxes out at a little over 2 Gb, where a plateau is reached. Also CPU usage is similar with a peak load at the beginning of the process and a continuous usage between 100 to 150 % during the rest of the runtime.

The tool documentation of MarkDuplicates mentions two parameters to modify memory usage, `--MAX_RECORDS_IN_RAM` and `--SORTING_COLLECTION_SIZE_RATIO`.

GATK HaplotypeCaller

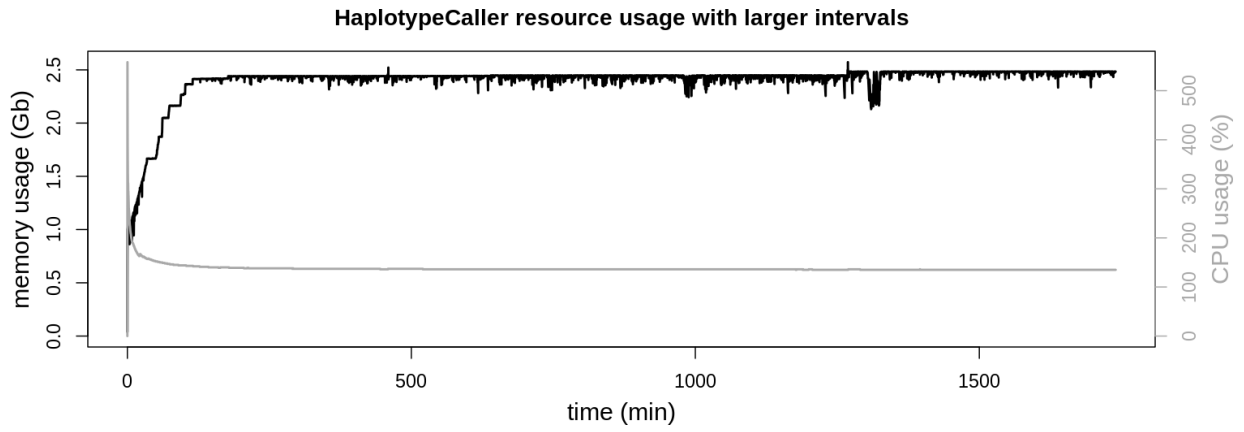
```
GC=2; Xmx=2; File=<SRR3041137|SRR3041413>
gatk --java-options "-Xmx${Xmx}G -XX:ParallelGCThreads=${GC}" HaplotypeCaller \
  -ERC GVCF \
  -I 03_mark_duplicates/${FILE}.bam \
  -R processed_reference/GCF_000002315.6_GRCg6a_genomic.fa.gz \
  -O ${MON_DIR}/${FILE}.gvcf.gz \
  -L "NC_006093.5"  2> ${MON_DIR}/${FILE}.log &
```



Again resource usage of HaplotypeCaller is very similar for different file sizes or sequencing depth, respectively. Both reach a plateau around 2.5 Gb memory usage. CPU loads have a very pronounced peak load quickly declining to approx. 135 % CPU load for the rest of the runtime.

HaplotypeCaller possesses the longest runtimes of the used GATK tools. In the above graphics a smaller contig (NC_006093.5; 36,374,701 bp) was evalutated to reduce runtime. To identify any particularities during longer runtimes, the application was also observed for a longer period. Therefore an interval comprising the two larges contigs

(NC_006088.5, 197,608,386 bp & NC_006089.5, 149,682,049 bp; sample SRR3041137) was also evaluated.

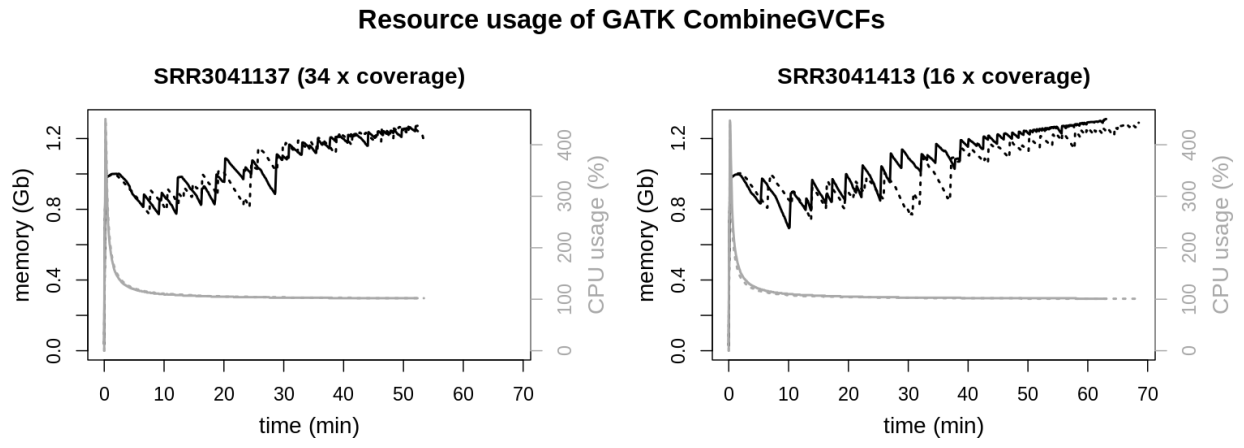


CPU usage stays at approx. 135 % during the majority of the runtime. Also resident set size keeps its plateau around 2.5 Gb of memory usage, only with minor fluctuations.

GATK CombineGVCFs

Combining the interval files was migrated to use GatherVcfs, which is considerably more efficient. This section is kept for reference purposes. Also CombineGVCFs is still used to consolidate all individuals into a single cohort. Due to its very brief runtime GatherVcfs was not accessed further.

```
GC=2; Xmx=2; File=<SRR3041137|SRR3041413>
gatk --java-options "-Xmx${Xmx}G -XX:ParallelGCThreads=${GC}" CombineGVCFs \
  -O ${MON_DIR}/${FILE}.gvcf.gz \
  -R processed_reference/GCF_000002315.6_GRCg6a_genomic.fa.gz \
  -V 04_haplotypeCaller/${FILE}/interval_2.gvcf.gz \
  -V 04_haplotypeCaller/${FILE}/interval_4.gvcf.gz \
  -V 04_haplotypeCaller/${FILE}/interval_1.gvcf.gz \
  -V 04_haplotypeCaller/${FILE}/interval_3.gvcf.gz 2> ${MON_DIR}/05_gathered_samples_$
↪{FILE}.log &
```



CPU and memory usage are similar for both files. Only runtimes are elongated for the larger data set. Interestingly file size and initial sequencing depth are not proportional for the input data (gvcf files). For the data set possessing the higher coverage HaplotypeCaller created a smaller gvcf file set (SRR3041137: 34 x coverage, 3.7 Gb) as for the data

with the lower coverage (SRR3041413: 16 x coverage, 4.8 Gb). Runtime of the process is clearly influenced by the total file size of the input data and not the initial sequencing depth.

Conclusions

From the above measurements some conclusions can be drawn concerning:

File size / sequencing depth There is a clear correlation between file size and the runtime of the respective process. The larger the file the longer it takes to evaluate it. Within the workflow there is also a clear correlation between sequencing depth and file size up to data processing by HaplotypeCaller. Obviously higher sequencing depth means larger bam files. HaplotypeCaller on the other hand converts the input data (bam files) into a totally new format (gvcf files), where higher sequencing depth does not necessarily mean more detected variants. So generally runtimes are influenced by the file size of the input data, which is connected to the size of bam files but not necessarily to gvcf file size.

Similarities and differences in resource usage of the tools One striking similarity of all observed GATK tools is their peak CPU load at the beginning of each process, quickly dropping again in the first few minutes. Only the peak height, meaning absolute CPU usage, is different. The highest peaks were observed for HaplotypeCaller with approx. 600 % CPU load (meaning 6 parallel threads). On the lower end MarkDuplicates was only using up to 250 % CPU (2 1/2 threads). Over the remaining runtime no tools showed a CPU usage of more than 150 %.

Memory usage on the other hand tends to reach a plateau, which is not necessarily identical to the limits specified by the Java heap size.

Maximum resource usage Monitoring CPU and memory usage over the complete runtime of a process helps to identify bottlenecks and the most demanding tools in both areas. There are no major differences in CPU usage. Beside the peak load each observed GATK tool uses 100 - 150 % of the CPU (1 to 1 1/2 threads), with CombineGVCFs being very close to a single thread (approx. 103 %). This was considered in the Snakefile by specifying **two threads** for all of the above tools besides CombineGVCFs, which was kept at the default of **one thread**.

In any case maximum memory usage is more important, than maximum CPU usage. In case that CPU becomes a bottleneck runtimes of the total data evaluation will simply increase. This might be annoying as processes might get paused but won't do further harm. Memory usage on the other side is a different issue. Memory usage cannot be postponed. On a system that runs out of memory the out of memory kill (OOM killer) will be invoked and select a task to kill to free up memory for the sake of the total system.

Therefore additional precautions were taken, to prevent OVarFlow from running out of memory. The maximum RSS values of the respective tool are decisive for planing of resource usage. Within the OVarFlows Snakefile the resources keyword is used to specify the maximum amount of memory that was observed within the above measurements. Thereby the amount of memory of the system OVarFlow is executed upon, can be specified if needed (keep in mind, the resources keyword will not enforce those limits). In this case a full command line would look like this:

```
snakemake -np --cores <threads> -s /path/to/Snakefile --resources mem_gb=<amount of␣
↪system memory>
```

The following resource requirements were set for the respective tool (resources and threads keyword in the Snakefile), while always adjusting upwards to the next full gigabyte value:

Tool	Java settings		Snakemake keyword	
	GC	heap size	resources (`mem_gb`)	threads
bwa|samtools			4 Gb	6
SortSam	2	10 Gb	5 Gb	2
MarkDuplicates	2	2 Gb	3 Gb	2
HaplotypeCaller	2	2 Gb	3 Gb	2
GatherVcfs	2	2 Gb	2 Gb	1
CombineGVCFs	2	2 Gb	2 Gb	1

The above thread values are not identical to the Java GC thread settings nor to the Java heap size. These values only reflect the approximate resource usage of the respective tool and are meant for resource management or planning by Snakemake, respectively.

3.12.4 Entire Workflow

Previously only single applications were benchmarked and optimized. Ultimately it's the resource usage of the entire workflow, that's of concern. Thus the entire OVarFlow workflow was monitored. Again CPU as well as memory usage were observed. This served several purposes:

1. Demonstrating the effect of the applied Java options on the entire workflow, thereby validating the effect.
2. Identification of excessive resource utilization during the workflow, that might still be left.
3. Giving an example of what can be done with a specific given hardware and how to maximize its utilization by adjusting some Snakemake options without introducing new overloads.

Again chicken (*Gallus gallus*) served as a reference organism, with the reference genome and annotation given in the benchmarking introduction. Six different runs from the study PRJEB12944 were used as sample data, with average coverages between 24 and 28 (ERR1303580, ERR1303581, ERR1303584, ERR1303585, ERR1303586, RR1303587).

Benchmarking was performed on a single cluster node (SGE), whose resources were exclusively reserved for OVarFlow. Exclusive reservation was achieved by requesting as many slots as the hardware provided parallel threads (in this case 40). The following hardware specifications where given:

- memory (file: */proc/meminfo*):

```
MemTotal:       264105108 kB (= 251.9 Gb)
SwapTotal:      124999676 kB (= 119.2 Gb)
```

- CPU (command: `lscpu`):

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 62
Model name:            Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz
Stepping:              4
```

(continues on next page)

(continued from previous page)

```
CPU MHz:               2042.538
CPU max MHz:           3300.0000
CPU min MHz:           1200.0000
BogoMIPS:              4989.96
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              25600K
NUMA node0 CPU(s):     0-9,20-29
NUMA node1 CPU(s):     10-19,30-39
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca␣
↪cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb␣
↪rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc␣
↪cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16␣
↪xtpr pdcm pcid dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx␣
↪f16c rdrand lahf_lm cpuid_fault epb pti intel_ppin ssbd ibrs ibpb stibp tpr_
↪shadow vnmi flexpriority ept vpid fsgsbase smep erms xsaveopt dtherm ida arat pln␣
↪pts md_clear flush_l1d
```

Resource usage of the entire machine was recorded every 30 second throughout the workflow using two shell commands:

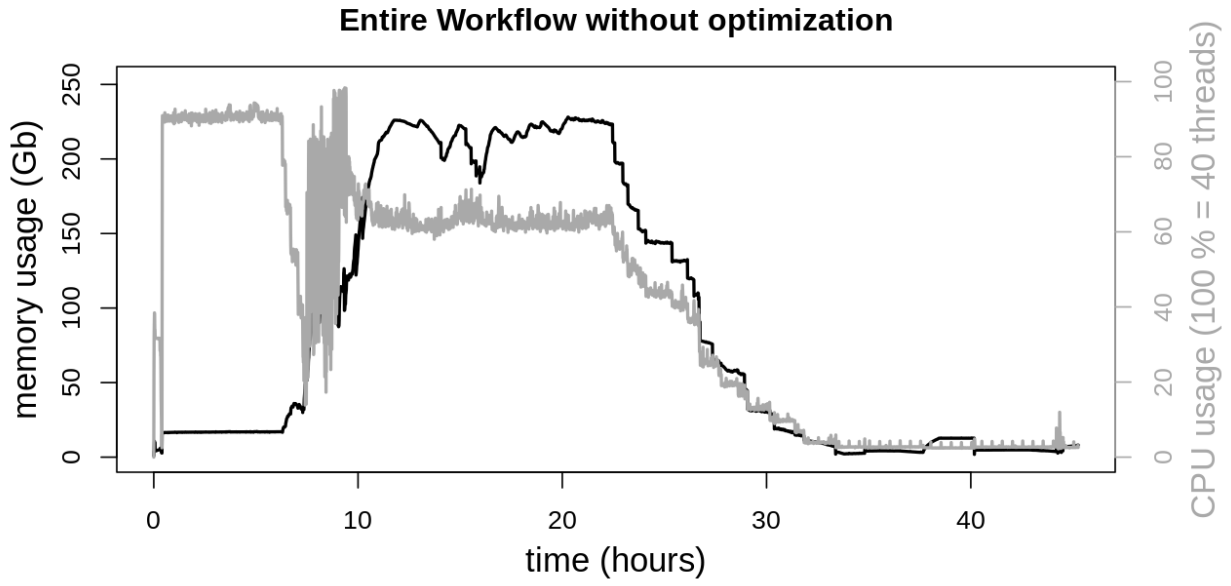
```
mpstat 30 > mpstat_statistics &
sar -r 30 > sar_statistics &
```

Processing of measurements was done using some shell commands including `sed` and `awk` to convert the file format into something more suitable for plotting CPU and memory usage with R. Again the respective scripts are deposited in the OVarFlow repository.

Workflow without optimization

To obtain a baseline measurement the entire workflow was executed without any Java optimizations. Therefore the settings for the number of parallel GC threads as well as the amount of memory available for the heap space where automatically chosen by the JVM. CPU utilization was already considered in the Snakefile through the `threads` keyword, as stated in the following table:

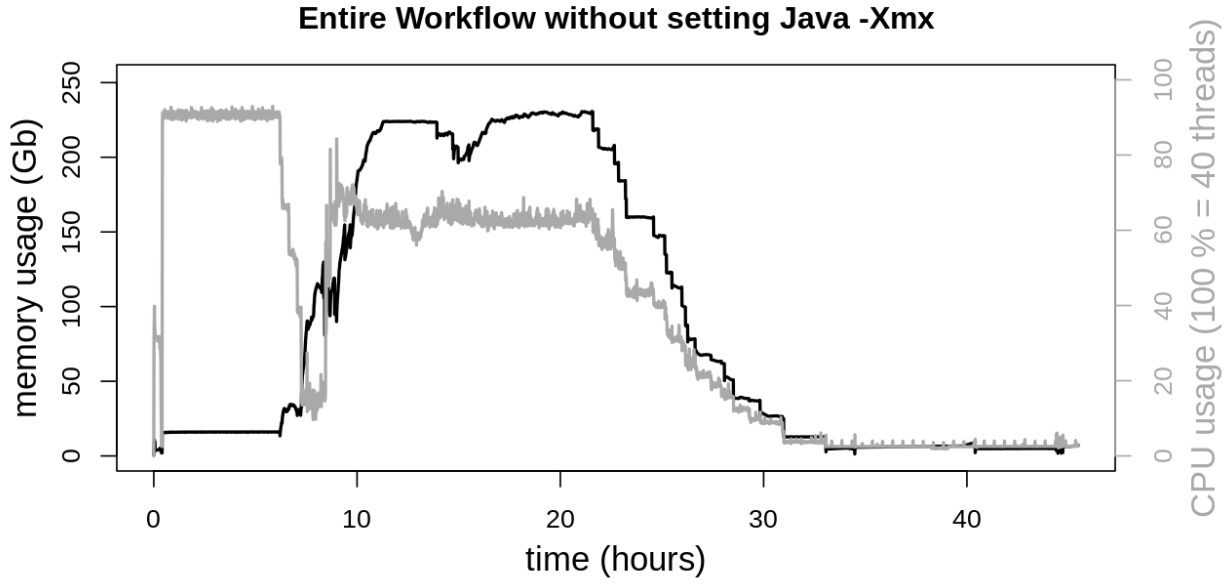
Tool	Rule	Threads
bwa / samtools	mapping	6
SortSam	sort_sam_gatk	2
MarkDuplicates	mark_duplicates	2
HaplotypeCaller	haplotypeCaller	2
CombineGVCFs	gather_intervals	1 (default)
default for other tools		1



System resources, both CPU and memory, show a high degree of utilization. Overall there are three phases of the workflow, that can be distinguished. The first phase (approx. to 7 h) is about mapping of the reads. Here high CPU utilization is desirable, while memory usage is rather low. In the second phase various GATK applications (SortSam, MarkDuplicates, HaplotypeCaller and CombineGVCFs) are executed in parallel. Here CPU utilization shows considerably more variability while memory usage peaks at around 225 Gb. In the third phase (after approx. 33 h) all tasks that can be executed in parallel are finished, and both CPU and memory utilization is rather moderate.

Workflow with Java GC optimization

The Java VM was tweaked by limiting the number of GC threads to two (`-XX:ParallelGCThreads=2`) for SortSam, MarkDuplicates, HaplotypeCaller and CombineGVCFs. Other GATK tools were limit to four GC threads. Finally the Snakefile was invoked limiting parallel threads to 38 (`snakemake -p --cores 38`).



As expected memory usage is not altered by the Java GC modification. The effect on CPU usage is very noticeable, being more consistent at the beginning of the second phase. Also CPU usage doesn't peak as high as previously. This is related to the execution of SortSam and MarkDuplicates. CPU utilization of both applications was considerably affected by the GC thread count (see Java Garbage Collection section). HaplotypeCaller and CombineGVCFs were considerably less affected by Java GC, as is reflected by the barely altered plateau at approx. 60 - 70 % CPU utilization. This moderate degree of CPU utilization is due to the fact, that Snakemake only allows for the specification of integer thread numbers, while HaplotypeCaller uses approx. 130 % of the CPU. 70 % of the second CPU core remains unused when specifying 2 threads for the HaplotypeCaller.

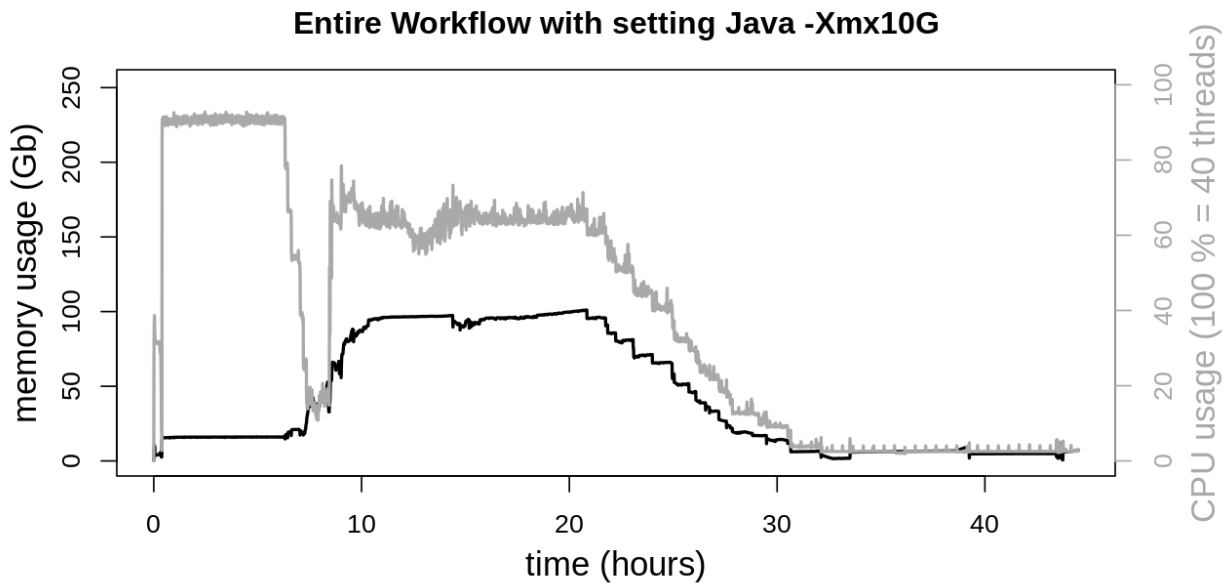
Workflow with Java heap optimization

The second regulating screw to the JVM is heap size (`-Xmx`) with some considerable effects on memory consumption. Two different settings were evaluated.

Workflow with Java -Xmx10G

At first equal heap sizes of 10 Gb were tested and applied to the four GATK tools SortSam, MarkDuplicates, HaplotypeCaller and CombineGVCFs. This was achieved by setting an environment variable:

```
export _JAVA_OPTIONS=-Xmx10G
```



CPU usage is not significantly affected. But memory consumption in the second phase of the workflow is drastically lowered. Previously memory consumption reached a plateau at approx. 225 Gb, which could be lowered to approx. 100 Gb.

Workflow with optimized heap sizes

To further optimize memory consumption of OVarFlow, Java heap sizes were every more granularly tuned to the single GATK tools:

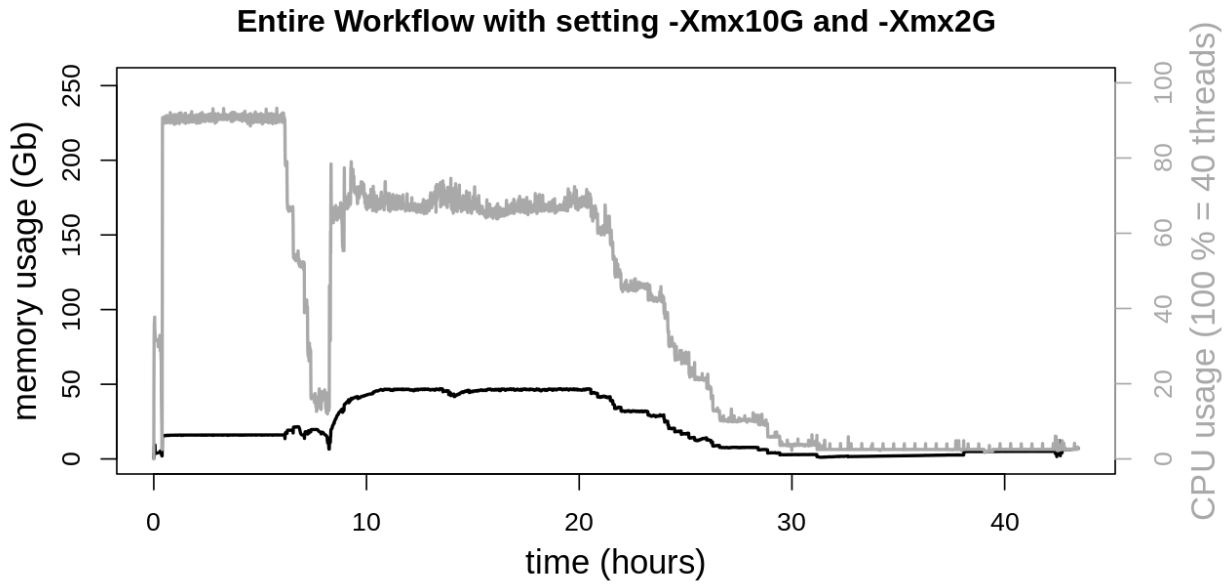
- GATK SortSam

```
export _JAVA_OPTIONS=-Xmx10G
```

- GATK MarkDuplicates, HaplotypeCaller and CombineGVCFs

```
export _JAVA_OPTIONS=-Xmx2G
```

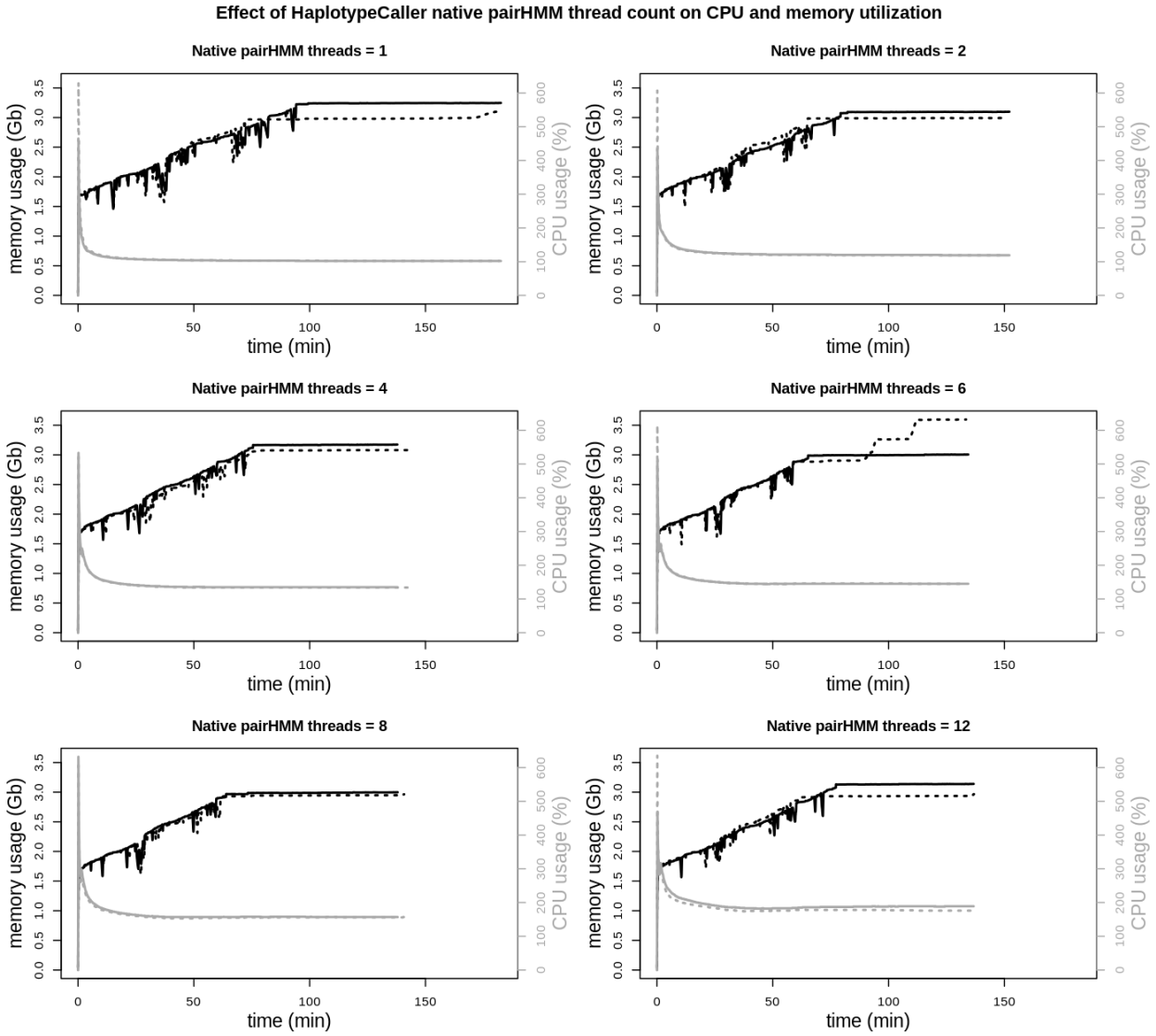
In doing so memory consumption of the second phase could further be reduced considerably. Memory utilization doesn't even exceed 50 Gb, which is another reduction by approx. 50 % as compared to only using `-Xmx10G`. Despite this drastic reduction CPU usage was not negatively affected by those changes.



3.12.5 Maximizing CPU utilization

As was noticed in previous benchmarkings of the entire workflow, in the HaplotypeCaller phase CPU utilization reached a plateau at approx. 60 - 70 % of all available resources. This can be explained by the fact, that each individual HaplotypeCaller process uses approx. 135 % of the CPU (i.e. 1.35 cores or threads are used). In the Snakefile on the other side only integer values for thread usage are possible. Finally, even though only 1.35 threads are used, 2 threads have to be reserved.

Basically to allow for an overall better CPU utilization of the entire workflow, CPU utilization of HaplotypeCaller has to be shifted to full integer values. To do so, the effects of different settings for the `--native-pair-hmm-threads` option (1, 2, 4, 6, 8 and 12 threads) were analyzed.



The impact of the number of native pairHMM threads is seen in the total runtimes and also CPU utilization. Total runtimes are declining from 1 to 4 threads, without a noticeable further improvement with higher thread counts. CPU utilization, apart from the always given initial peak load, is continuously rising, as can also be seen from the average CPU utilization:

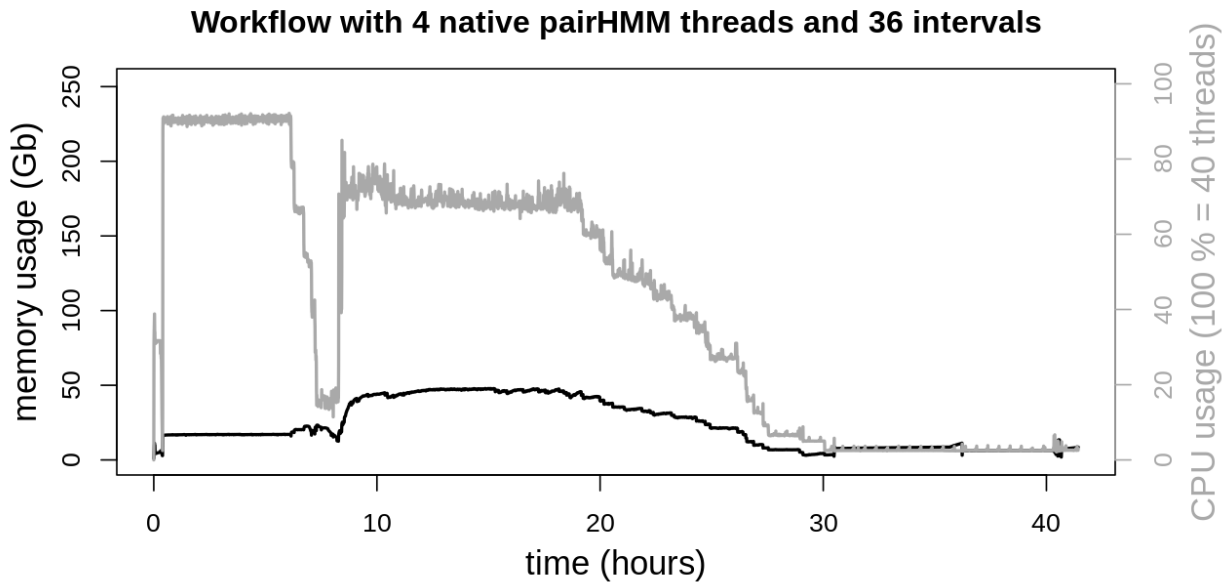
native pairHMM threads	average CPU usage (%)
1	106
2	124
4	140
6	152
8	164
12	188

A higher CPU utilization, close to two threads, is achieved when using 12 native pairHMM threads. But as runtimes are not improved over using 4 threads, there is no benefit in increasing the native pairHMM thread count. On the other side when using only 1 native pairHMM thread, CPU usage is nearly reduced to a single thread. Considering the advantage of being able to run twice the amount of HaplotypeCaller processes in parallel, the slightly increased runtime of the

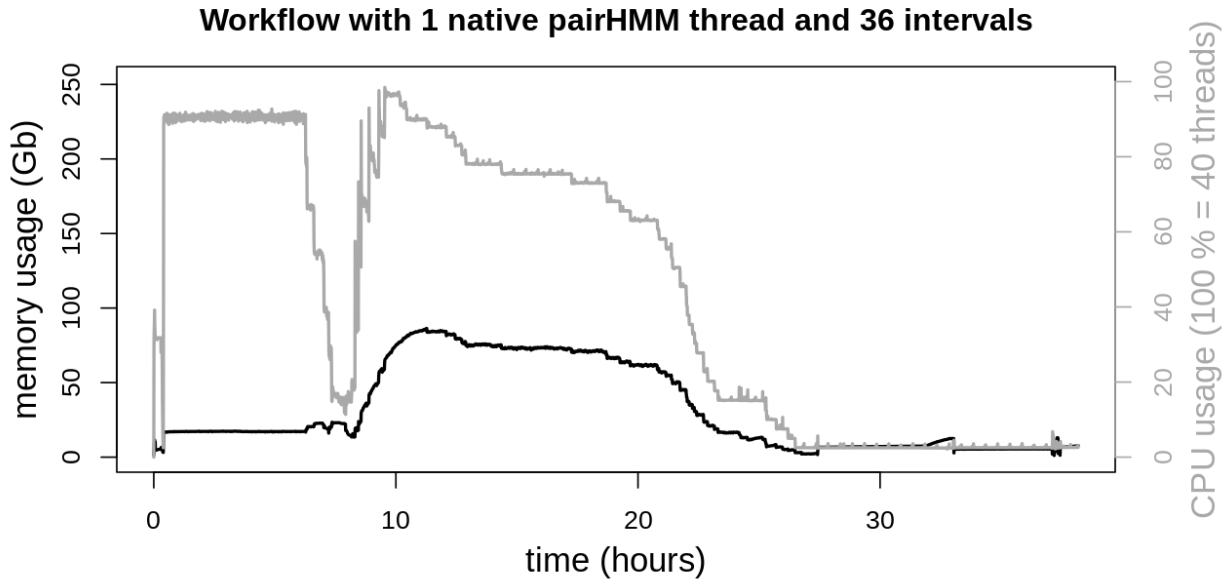
individual process should be negligible.

To test the above hypothesis, the entire workflow was benchmarked again with optimized settings (see previous workflow benchmarking). In two identical data evaluations only the number of native pairHMM threads and the number of Snakemake threads provided to the HaplotypeCaller were altered. Further settings were as follows:

- 6 intervals per sequencing data set with 6 data sets, resulting in a total of 36 intervals to call
- a maximum of 38 Snakemake threads (`--cores 38`)
- `-XX:ParallelGCThreads=2` for SortSam, MarkDuplicates, HaplotypeCaller and ComineGVCFs
- `-Xmx10G` for SortSam
- `-Xmx2G` for MarkDuplicates, HaplotypeCaller and CombineGVCFs



The initial settings reserved 2 threads for every HaplotypeCaller. With a maximum of 38 parallel Snakemake jobs no more than 19 parallel HaplotypeCaller were possible. With this setup again a plateau was reached at approx. 70 % total CPU utilization.



With the reduced CPU usage of a single native pairHMM thread and up to 38 parallel Snakemake jobs, all 36 intervals could be evaluated in parallel by the HaplotypeCaller. As not all intervals are of the same length, short ones are finished before larger ones. This results in a peak load above 90 % total CPU utilization and progressively declining CPU load as more and more HaplotypeCallers are finishing on their respective interval.

When comparing both data evaluations and also previous benchmarkings of the entire workflow, a reduction in the total runtime is observed. When reducing the CPU load of a single HaplotypeCaller but increasing the number of parallel jobs, the entire workflow was finished in 38 h 11 min. On the other side the workflow without reducing performance of the individual HaplotypeCaller was finished in 41 h 26 min. This clearly shows the overall runtime advantage of the higher parallelization, despite the individual HaplotypeCaller being less performant.

The above procedure is especially interesting when hardware resources are limited and a maximum degree of system utilization is desirable. On the other hand, if hardware resources are not a limiting factor, for instance with a large cluster that would idle anyways, not limiting the individual HaplotypeCaller would result in a slightly reduced runtime. In such circumstances the default of 4 native pairHMM threads is the optimum. For general usage OVarFlow includes the default setting of four native pairHMM threads. Configuration is enabled through the `config.yaml` file.

3.13 Hardware recommendations

Resource requirements to perform variant calling with OVarFlow depend on the size of the respective project. Here the main factors are the organisms genome size and of course the number of data sets or individuals within the study, respectively. Non the less some general recommendations can be made. A first impression can also be obtained from the benchmarking of the “entire workflow” (especially the last image of the section).

Concerning the hardware three key components have to be considered:

CPU Of course a high single thread performance is always helpful to accelerate calculations. But major parts of variant calling can be parallelized, so that several data sets (and even intervals) can be processed in parallel. Therefore a high number of CPU cores is considerably more helpful. Finally a shortage of processing power will result in longer waiting times till results are calculated.

As a final note: the CPU must support the AVX (Advanced Vector Extensions) instruction set extension, as this drastically increases calculations performed by HaplotypeCaller.

Main memory (RAM) OVarFlow has been designed and tested to be quite efficient when it comes to memory usage. Still no final number of memory requirements can be given. In the end memory requirements depend on the number of samples, number of HaplotypeCaller intervals and processing steps that are run in parallel. A lack of memory cannot be compensated (swapping is not advisable here) and will result in invocation of the out of memory killer (OOM killer), thereby terminating running processes. The solution to such a situation would be to run less parallel Snakemake jobs or to provide more main memory.

Data storage Variant calling depends on large amounts of sequencing data, that are processed in multiple steps. This produces large amounts of intermediate data. Several terabytes of data can be produced easily. For instance 526 Gb of compressed sequencing data (32 fastq.gz files) resulted in 2.8 Tb of output data, when processed with OVarFlow. Besides storage space, file system latency and throughput are of concern. Slow storage, be it local disk storage or network storage, can slow down the whole analysis as well.

No single hardware component acts in isolation. Therefore the system has to be considered as a whole unit and single bottlenecks have to be identified. Adding more CPU cores won't help if memory is already scarce.

Also high performance computing (HPC) is best suited for variant calling, smaller projects could also be realised with smaller hardware budgets. Here a very rough estimation shall be given, based upon hardware availability in 2020. The given estimations come without any warranty and are based upon personal experience and estimates:

Desktop Computers Most desktop computers are not suitable for variant calling, as they do not offer the required number of CPU cores. For smaller projects, meaning less then 10 individuals or smaller genomes (e.g. *Drosophila melanogaster* with approx. 144 Mb genome size) a desktop CPU like AMD Ryzen (TM) 9 5950X or 9 3950X (16 cores / 32 threads) or comparable combined with at least 64 Gb main memory (better 128 Gb) might be suitable.

High End Desktop In the last couple of years High End Desktop (HEDT) computers approached the performance previously reserved to server computers. Especially with AMD's line of Threadripper (TM) processors up to 64 cores and 128 threads are available within a single CPU (as of the begining of 2021). When combined with 256 Gb of main memory medium sized projects with some dozens of individuals might be possible. Storage space will probably be a problem though.

Servers and Clusters For large scale projects with hundreds of samples a dedicated infrastructure is definitely required. The usage of a compute cluster, for instance based upon Son of Grid Engine, is the way to go.

3.14 The basic variant calling workflow

This section lists the most basic commands that are needed to perform variant calling. Therefore it is targeted at those people that are:

- not interested in the usage of OVarFlow as a full-fledged workflow for variant calling, but are interested in the commands involved,
- want to get a basic understanding of what's going on under the hood of this workflow but get confused by the syntax of Snakemake and the Snakefile used,
- novice users of GATK as GATK best practises can be confusing in the beginning.

This section lists the major shell commands in their most basic form, that are needed to perform variant calling with GATK. Essentially this will give you a very general overview of variant calling using GATK 4. It is highly recommended to also have at least a look at the section covering "GATK particularities". This section tries to cover some of GATK's very subtle usage issues which can have considerable effects on runtime and resource usage. Those issues won't be covered here.

To make use of the description provided here, a basic understanding of the Unix shell bash (or similar) is required.

3.14.1 Overview of the workflow

In the sections below details of the exact variant calling workflow are outlined. This section is supposed to give a schematic and rather rough summary of the process. One might distinguish three phases:

- Data pre-processing
  0. `fastqc`: quality control of reads (optional but recommended)
  1. `bwa mem`: mapping to the reference genome
  2. `gatk SortSam`: sorting of the reads
  3. `gatk MarkDuplicates`: as the name implies
- Preparation of variants
  4. `gatk HaplotypeCaller`: actual variant calling
  5. `gatk GatherVcfs`: pooling of intervals (optional)
  6. `gatk CombineGVCFs`: pooling of called individuals
  7. `gatk GenotypeGVCFs`: genotyping of the called variants
  8. `gatk SelectVariants`: separating of SNPs and indels
  9. `gatk VariantFiltration`: hard filtering of SNPs and indels
  10. `gatk SortVcf`: merging of SNPs and indels
  11. `gatk SelectVariants`: removal of filtered variants
- Variant annotation
  12. `snpEff`: variant annotation

3.14.2 Preparing the workflow

For the variant calling workflow outlined below, several files and databases have to be prepared in the first place. This is mostly related to the preprocessing of reference genomes and annotations. It is advisable to perform those steps before the actual data evaluation, as the workflow might fail if one database cannot be created. This is especially annoying if it happens in the last step, where a database for the usage of `snpEff` (which is used for the functional annotation of detected variants) is required.

Compression of the reference

To save some disc space reference genomes should usually be compressed. This is often accomplished via `gzip`. The drawback of this program is, that compression cannot be indexed easily. Therefore the reference genome should be compressed with `bgzip`, which is very similar to `gzip`.

```
gunzip --to-stdout <ref_genome.gz> | bgzip > <ref_genome_recompressed.gz>
```

Indexing the reference genome

```
samtools faidx <ref_genome_recompressed.gz>
```

Creating a bwa index

```
bwa index <ref_genome_recompressed.gz>
```

A dictonary for GATK

```
gatk CreateSequenceDictionary -R <ref_genome_recompressed.gz>
```

A database for snpEff

From personal experience it is advisable to use an annotation from the RefSeq in gff format. Processing of those files with `snpEff` has been unproblematic so far. Gtf formatted files as available from Ensembl might not be parsable by `snpEff`. When in doubt try creating the snpEff database first.

For `snpEff` to read the annotation, it shouldn't be compressed and be named *genes.gff*. Furthermore the reference should be copied to the same directory and named *sequences.fa.gz*. Compression of the reference sequence is fine. The directory to which the files are copied should be named after the reference genome you're using (`_name_`), but without any file specific extension.

```
gunzip --to-stdout <ref_anno.gz> > <snpEffDB/_name_/genes.gff>
cp <ref_sequence.gz> <snpEffDB/_name_/sequence.fa.gz>
```

`snpEff` can create global databases, which will reside in your home directory by default. It is advisable to create an annotation database in your local project directory.

```
snpEff -Xmx12g build -dataDir ${PWD}/snpEffDB \
  -configOption <_name_>.genome=<_name_> \
  -gff3 -v <_name_>
```

Usage of `snpEff` can be daunting in the beginning. Fortunately the online documentation is quite comprehensive.

3.14.3 The actual variant calling

For this example procedure it is assumed, that for each individual to be analyzed, all reads are contained in only one fastq file containing the forward reads (R1) and one file containing the reverse reads (R2). If there are several files for forward and reverse reads, those have to be merged in advance.

Quality control via FastQC

```
fastqc -o <output dir> -f fastq <input file.fastq.gz>
```

Mapping of fastq files

When using GATK `bwa mem` is probably the most widely used mapper. Of course an index database has to be created in the first place (`idxbase`). Directly piping into samtools will produce compressed bam files. Writing stderr to separate files will preserve any potential error or log messages.

```
bwa mem -M -t <number_of_threads> -R <read_group_tag> <idxbase> \
  <forward_reads.fastq> <reverse_reads.fastq> 2> <bwa_log_message> | \
  samtools view -v /dev/fd/0 -o <output_mapping.bam> 2> <samtools_log_message>
```

Sorting of mapped bam files

Here as well as in subsequent steps the "output mapping" of the last command will act as input of this step.

```
gatk SortSam -I <input_mapping.bam> -SO coordinate -O <output_mapping.bam>
```

Marking of duplicated reads

```
gatk MarkDuplicates -I <input_mapping.bam> -O <output_mapping.bam> \
  -M <metrics.txt>
```

Creating an index of marked mappings

```
samtools index <output_mapping.bam>
```

Variant calling with HaplotypeCaller

Optionally OVarFlow is capable of calling variants on several intervals per individual. By executing the HaplotypeCaller on intervals a high degree of parallelization is achieved and analysis times are reduced considerably.

When applying a *.gz suffix to the output genomics variant call format file (GVCF; ending: g.vcf) the resulting file will automatically be compressed.

```
gatk HaplotypeCaller -ERC GVCF -I <input_mapping.bam> -R <reference_genome> \
  -O <interval_xy.g.vcf.gz>
```

Gathering of intervals per individual

If variant calling was performed in parallel on intervals, the resulting intervals have to be combined.

```
gatk GatherVcfs -O <individual_xy.g.vcf.gz> -I <interval_1.g.vcf.gz> \
  -I <interval_2.g.vcf.gz> ... -I <interval_n.g.vcf.gz>
```

This step could also be performed by `CombineGVCFs`, but `GatherVcfs` is considerably quicker. A downside of `GatherVcfs` is, that it can only handle intervals that preserve the initial order of contigs. This issue is automatically taken care of in the workflow.

Combining the individual variant files

For every processed individual a single file is created. Those files have to be combined.

```
gatk CombineGVCFs -O <combinde_variants.g.vcf.gz> -R <reference_genome>  \
  -V <individual_1.g.vcf.g> -V <individual_2.g.vcf.gz> ... -V <individual_n.g.vcf.gz>
```

Processing the Genotype

Even though the HaplotypeCaller writes genotype information into the initial GVCF file, this information is lost when merging the individual GVCF files. By performing joint genotyping over several individuals the genotyping accuracy is improved and the genotype information is restored.

```
gatk GenotypeGVCFs -R <reference_genome> -V <input.g.vcf.gz> -O <output.vcf.gz>
```

Quality filtering of variants

HaplotypeCaller will also create some false positive variant calls. GATK offers two approaches to reduce the amount of false positives.

- Variant quality score recalibration (VQSR): This approach needs a so-called "truth set" of already known variants for the respective organism. This truth set is used in a machine learning approach, to learn the profile of likely real variants. This method is most feasible with well studied organisms, exhibiting highly reliable variant datasets.
- Hard filtering: In hard filtering solid thresholds are applied onto the quality parameters of each called variant. Variants not meeting those quality thresholds will be discarded.

A third approach shall also be mentioned. In a kind of an iterative process first hard filtering is used to create an initial data set of variants. This initial variant set is then used in a second step to perform VQSR. This approach has the potential drawback of introducing a bias within the hard filtering which is then learned and applied in the VQSR. Therefore a single hard filtering step was used in this workflow.

Separating SNPs and indels

Different thresholds have to be applied for SNPs and indels. They have to be separated in the first step.

```
gatk SelectVariants -V <input.vcf.gz> -select-type SNP -O <output_snps.vcf.gz>
```

```
gatk SelectVariants -V <input.vcf.gz> -select-type INDEL -select-type MIXED  \
  -O <output_indels.vcf.gz>
```

It is imported to combine the options `-select-type INDEL` and `-select-type MIXED` as otherwise positions showing both types of variants will be lost.

Hard filtering

In hard filtering various filters are applied. It is important not to chain the single filters via the logical or operator (||) (see GATK). In this case the entire filter would pass as soon as a single filter condition is not fulfilled.

```
gatk VariantFiltration -V <input_snps.vcf.gz> \
  -filter 'QD <2.0' --filter-name 'QD2' \
  -filter 'QUAL < 30.0' --filter-name 'QUAL30' \
  -filter 'SOR > 3.0' --filter-name 'SOR3' \
  -filter 'FS > 60.0' --filter-name 'FS60' \
  -filter 'MQ < 40.0' --filter-name 'MQ40' \
  -filter 'MQRankSum < -12.5' --filter-name 'MQRankSum-12.5' \
  -filter 'ReadPosRankSum < -8.0' --filter-name 'ReadPosRankSum-8' \
  -O <output_snps.vcf.gz>
```

```
gatk VariantFiltration -V <input_indels.vcf.gz> \
  -filter 'QD < 2.0' --filter-name 'QD2' \
  -filter 'QUAL < 30.0' --filter-name 'QUAL30' \
  -filter 'FS > 200.0' --filter-name 'FS200' \
  -filter 'ReadPosRankSum < -20.0' --filter-name 'ReadPosRankSum-20' \
  -O <output_indels.vcf.gz>
```

Merging SNPs and indels

After the filtering step SNPs and indels can be reunified.

```
gatk SortVcf -I <input_snps.vcf.gz> -I <input_indels.vcf.gz> -O <sorted_variants.vcf.gz>
```

Remove filtered variants

Hard filtering will only tag variants not meeting the filtering criteria. Still they have to be removed from the dataset. This step basically creates your finished data set containing the called variants.

```
gatk SelectVariants -V <sorted_variants.vcf.gz> -O <filtered_variants.vcf.gz> \
  --exclude-filtered true
```

Annotating the variants

A further tool might be used to create a functional annotation of your variants. Based upon a valid genome annotation, functional annotation of the variants will determine if the respective variant is for instance within a coding region of a gene. Also the effect of a variant will be computed, telling if a synonymous, non synonymous or nonsense mutation is given through the respective variant. The program `snpEff` is one option between many.

```
snpEff -Xmx12g <_name_> <filtered_variants.vcf.gz> \
  -dataDir <path/to/snpEffDB> \
  -configOption <_name_>.genome=<_name_> \
  -stats <snpEff_summary.html> | \
  bgzip > <annotated_variants.vcf.gz>
```

The option `-Xmx12g` will increase the memory available to the Java virtual machine. This might be needed for larger genomes. Otherwise it's optional and can be changed or bypassed.

3.14.4 DAG of the workflow

Snakemake creates a directed acyclic graph (DAG) of the workflow, within the so-called DAG phase. This DAG can be visualized using the `dot` command. To give an example of the workflow, two input datasets (each with forward and reverse reads) were analyzed using three HaplotypeCaller intervals. The resulting DAG of this workflow is shown in the following figure. Each rounded box represents the execution of a single Snakemake rule (knot of the graph) the arrows show the succession of the rules (edges of the graph).

3.15 The extended BQSR workflow

The previous section listed the most basic form of variant calling, that will deliver first results. Those results can already be sufficient. On the other hand, the GATK team recommends to also perform base quality score recalibration (BQSR). Therefore a second workflow was created, that can optionally be performed in succession to the first workflow. This workflow will use the results generated in the first, basic workflow to perform BQSR and thereby hopefully deliver further improved variant calls.

Just as before in the basic workflow, this section lists the major shell commands in their basic form to perform BQSR and improved variant calling. The actual commands implemented in the workflow will include additional options to obtain improved performance of the respective GATK application. Those optimizations are not covered here.

The BQSR workflow further processes some files generated in the basic variant calling workflow. The generation of those files has to be looked up in the previous workflow.

3.15.1 Overview of the workflow

This section gives a brief overview of the exact GATK tools that are used and the succession of their usage. Beginning from step 16, the workflow is basically a repetition of the variant calling as performed in the basic workflow.

- Base quality optimization

  13. `gatk BaseRecalibrator`: generation of recalibration table for BQSR
  14. `gatk ApplyBQSR`: actual base quality score recalibration of reads
  15. `gatk BaseRecalibrator`: recalibration table after BQSR

      `gatk AnalyzeCovariates`: comparison of recalibrated bases

- Variant calling
  16. `gatk HaplotypeCaller`: actual variant calling
  17. `gatk GatherVcfs`: pooling of intervals (optional)
  18. `gatk CombineGVCFs`: pooling of called individuals
  19. `gatk GenotypeGVCFs`: genotyping of called variants
  20. `gatk SelectVariants`: separating of SNPs and indels
  21. `gatk VariantFiltration`: hard filtering of SNPs and indels
  22. `gatk SortVcf`: merging of SNPs and indels
  23. `gatk SelectVariants`: removal of filtered variants
- Variant annotation
  24. `snpEff`: variant annotation

3.15.2 Base quality optimization

As the name implies base quality score recalibration (BQSR) is a processing step of the reads to optimize the given quality scores. During sequencing the base callers can introduce systematic errors, when judging the base quality (phred score). This step is supposed to improve those quality scores and therefore differentiation between real variants and just wrongly called bases. Further details are listed by the GATK team.

Initial analysis of base quality scores

```
gatk BaseRecalibrator -R <reference_genome> -I <mapping_of_marked_duplicates.bam> \
  --known-sites <called_variants.vcf.gz> -O <first_recalibration.table>
```

Change the given quality scores

```
gatk ApplyBQSR -R <reference_genome> -I <mapping_of_marked_duplicates.bam> \
  -bqsr <first_recalibration.table> -O <optimized_read_mapping.bam>
```

Analysis of recalibration effects

Second analysis of base quality scores, to judge the effect of the quality score recalibration.

```
gatk BaseRecalibrator -R <reference_genome> -I <optimized_read_mapping.bam> \
  --known-sites <called_variants.vcf.gz> -O <second_recalibration.table>
```

After the analysis of the improved mappings, the results before and after quality score optimization can be compared.

```
gatk AnalyzeCovariates -before <first_recalibration.table> \
  -after <second_recalibration.table> -plots <analysis_results.pdf>
```

3.15.3 Improved variant calling

The following steps 16 till 24 are basically identical to the variant calling steps as performed in the basic workflow, beginning with the HaplotypeCaller and following steps. After variant calling, annotation of the called variants is performed via SnpEff. A detailed description of the actual commands can be found in the basic workflow description.

3.15.4 DAG of the BQSR workflow

Again, the succession of rules (a directed acyclic graph - DAG) that are applied by Snakemake to evaluate the input data can be visualized. In the given example graph two data sets are evaluated. For each of the data sets variant calling through HaplotypeCaller is performed on three different intervals. Therefore in the given example, six HaplotypeCaller processes might be executed in parallel, given that sufficient hardware resources are available.

It has to be noted, that the shown graph cannot stand by itself. Previous data evaluation through the basic workflow has to be performed, as the generated results are further processed in the BQSR workflow.

3.16 OVarFlow 2

When processing large cohorts of individuals (more than 50 to 100 individuals), OVarFlow 1 offered some additional leeway for performance improvements. These issues were addressed in OVarFlow 2. Previously, only the HaplotypeCaller was parallelized based on genomic intervals that could be processed simultaneously. By rearranging two steps of the workflow, the GATK tools CombineGVCFs and GenotypeGVCFs are able to process the same genomic intervals that the HaplotypeCaller was utilizing. This allows these two steps, which can take quite a long time for large cohorts, to be dramatically accelerated through parallelization.

However, the first version of OVarFlow remains a valid tool for analyzing small to medium sized cohorts. OVarFlow 2 will be especially beneficial when analyzing very large cohorts, comprising hundreds of individuals. For instance, when using whole genome sequencing data for genome-wide association studies (GWAS), OVarFlow 2 provides significant improvements which can be in the order of several weeks when dealing with very large cohort sizes.

3.16.1 Usage of OVarFlow 2

Fortunately, OVarFlow 2 could be designed in a way that the usage remains identical to the first generation of the workflow. This also includes the CSV and Yaml configuration files. Therefore, all previous usage descriptions remain valid.

Only one alteration on the end user side has to be considered: preexisting GVCF files (*.g.vcf.gz) cannot be included in the analysis. To remain fully compatible with OVarFlow 1, the corresponding entry was not remove from the CSV configuration file. If any GVCF files are specified here, OVarFlow 2 will war the user, that these files won't be used in the analysis. Instead, the fastq files should be provided for all individuals to be analyzed.

The only real difference when executing OVarFlow 2 is that a different Snakefile has to be specified on the command line, namely `Snakefile_OVarFlow2`. Nevertheless, a brief recap of the most important commands shall be given here:

- Perform a dry run of the workflow:

```
snakemake -np --cores <threads> --snakefile Snakefile_OVarFlow2
```

- Perform the actual run:

```
snakemake -p --cores <threads> --snakefile Snakefile_OVarFlow2
```

3.16.2 Overview of the workflow

Here, a brief overview of the tools used in the workflow and the sequence of their use is given. The tools used are identical to the first version of OVarFlow, but due to some rearrangements a higher degree of parallelization was achieved.

- Data pre-processing
  0. `fastqc`: quality control of reads (optional but recommended)
  1. `bwa mem`: mapping to the reference genome
  2. `gatk SortSam`: sorting of the reads
  3. `gatk MarkDuplicates`: as the name implies
- Preparation of variants
  4. `gatk HaplotypeCaller`: actual variant calling (on intervals)
  5. `gatk CombineGVCFs`: pooling of called individuals (on intervals)
  6. `gatk GenotypeGVCFs`: genotyping of the called variants (on intervals)
  7. `gatk GatherVcfs`: pooling of intervals
  8. `gatk SelectVariants`: separating of SNPs and indels
  9. `gatk VariantFiltration`: hard filtering of SNPs and indels
  10. `gatk SortVcf`: merging of SNPs and indels
  11. `gatk SelectVariants`: removal of filtered variants
- Variant annotation
  12. `snpEff`: variant annotation

Compared to OVarFlow 1, only the steps 5, 6 and 7 are rearranged and use some slightly modified commands (apart from some changes to the Snakefile that were necessary to make things work). Here, the most basic syntax of these three commands shall be provided, but no special Java options or tuning parameters will be given.

Pooling of intervals per individual

```
gatk CombineGVCFs -O <interval_n.g.vcf.gz> -L <interval_n.list> -R <reference_genome> \
 -V <individual_1/interval_n.g.vcf.gz> -V <individual_2/interval_n.vcf.gz> \
 ... -V <individual_n/interval_n.g.vcf.gz>
```

Processing the Genotypes

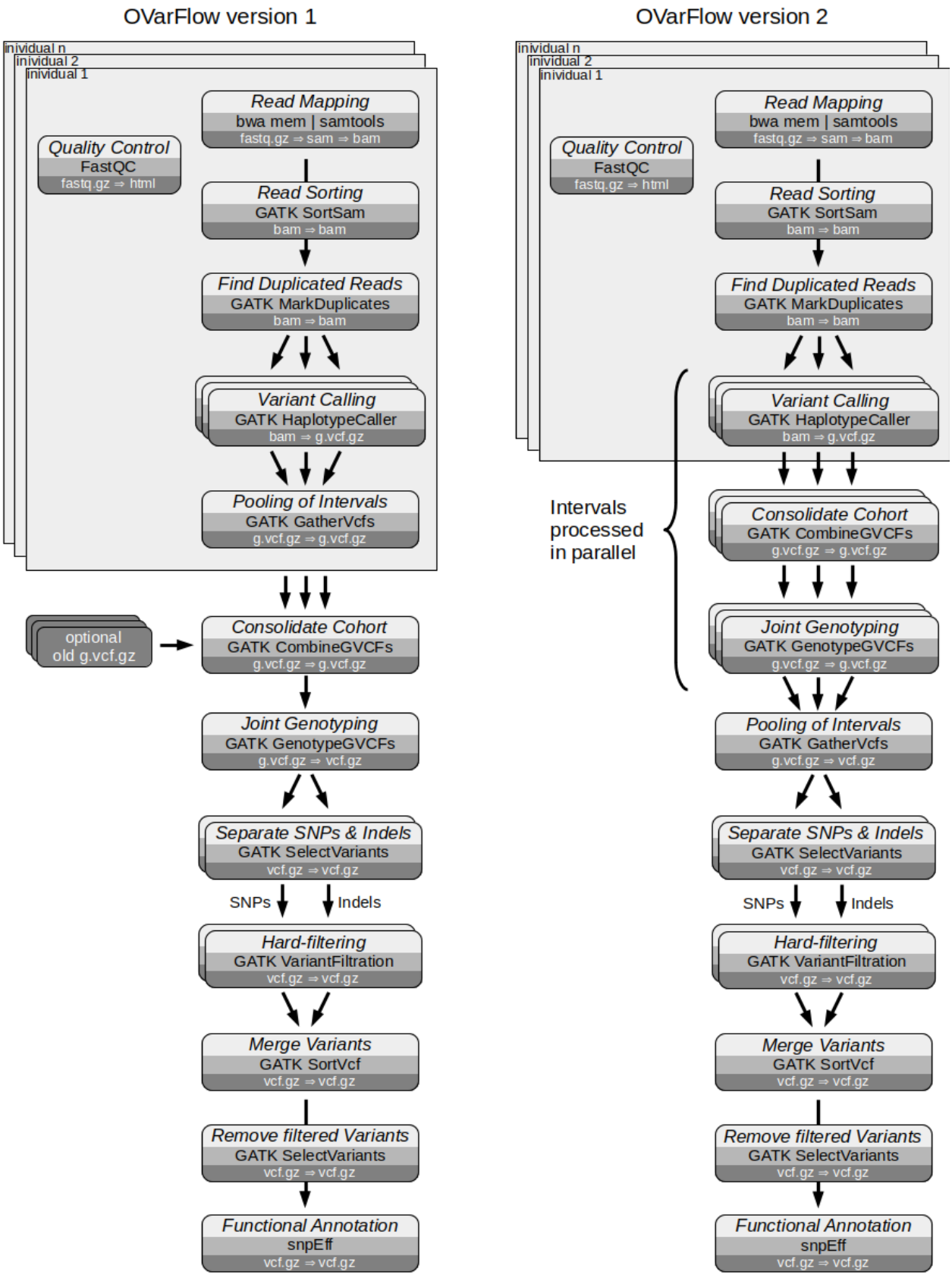
```
gatk GenotypeGVCFs -L <interval_n.list> -R <reference_genome> \
 -V <interval_n.g.vcf.gz> -O <interval_n.g.vcf.gz>
```

Gathering of the intervals

```
gatk GatherVcfs -o <all_genotyped_calls.g.vcf.gz> -I <interval_1.g.vcf.gz> -\
 -I <interval_2.vcf.gz> ... -I <interval_n.g.vcf.gz
```

3.16.3 Comparison of OVarFlow 1 & 2

The flow chat below illustrates the differences in the workflow between version one and two. The most obvious is that GATK CombinegVCFs and GenotypeGVCFs will also work in parallel. This was achieved by postponing *Pooling of the Intervals*.


OVarFlow version 1
individual n
individual 2
individual 1
Quality Control
FastQC
fastq.gz ⇒ html
Read Mapping
bwa mem | samtools
fastq.gz ⇒ sam ⇒ bam
Read Sorting
GATK SortSam
bam ⇒ bam
Find Duplicated Reads
GATK MarkDuplicates
bam ⇒ bam
Variant Calling
GATK HaplotypeCaller
bam ⇒ g.vcf.gz
Pooling of Intervals
GATK GatherVcfs
g.vcf.gz ⇒ g.vcf.gz
optional
old g.vcf.gz
Consolidate Cohort
GATK CombineGVCFs
g.vcf.gz ⇒ g.vcf.gz
Joint Genotyping
GATK GenotypeGVCFs
g.vcf.gz ⇒ vcf.gz
Separate SNPs & Indels
GATK SelectVariants
vcf.gz ⇒ vcf.gz
SNPs
Indels
Hard-filtering
GATK VariantFiltration
vcf.gz ⇒ vcf.gz
Merge Variants
GATK SortVcf
vcf.gz ⇒ vcf.gz
Remove filtered Variants
GATK SelectVariants
vcf.gz ⇒ vcf.gz
Functional Annotation
snpEff
vcf.gz ⇒ vcf.gz
OVarFlow version 2
individual n
individual 2
individual 1
Quality Control
FastQC
fastq.gz ⇒ html
Read Mapping
bwa mem | samtools
fastq.gz ⇒ sam ⇒ bam
Read Sorting
GATK SortSam
bam ⇒ bam
Find Duplicated Reads
GATK MarkDuplicates
bam ⇒ bam
Variant Calling
GATK HaplotypeCaller
bam ⇒ g.vcf.gz
Intervals
processed
in parallel
Consolidate Cohort
GATK CombineGVCFs
g.vcf.gz ⇒ g.vcf.gz
Joint Genotyping
GATK GenotypeGVCFs
g.vcf.gz ⇒ g.vcf.gz
Pooling of Intervals
GATK GatherVcfs
g.vcf.gz ⇒ vcf.gz
Separate SNPs & Indels
GATK SelectVariants
vcf.gz ⇒ vcf.gz
SNPs
Indels
Hard-filtering
GATK VariantFiltration
vcf.gz ⇒ vcf.gz
Merge Variants
GATK SortVcf
vcf.gz ⇒ vcf.gz
Remove filtered Variants
GATK SelectVariants
vcf.gz ⇒ vcf.gz
Functional Annotation
snpEff
vcf.gz ⇒ vcf.gz

3.16.4 Comparative Benchmarking

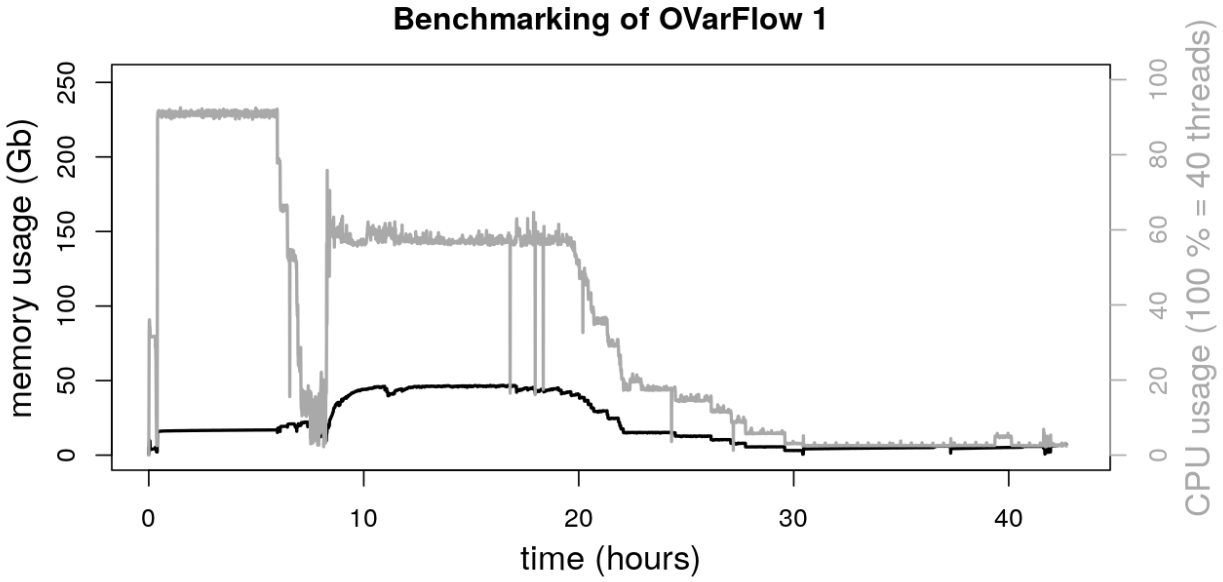
As explained before, OVarFlow 2 advances parallelizition by performing more computations on genomic intervals. To see the impact of parallelizing GATK CombineGVCFs and GenotypeGVCFs, the entire workflows of OVarFlow 1 and 2 were benchmarked. Benchmarking was performed as previously described in section "Resource Optimization -> Benchmarking & Optimizations -> Entire Workflow". In addition, the same sequencing data and hardware resources were used as before.

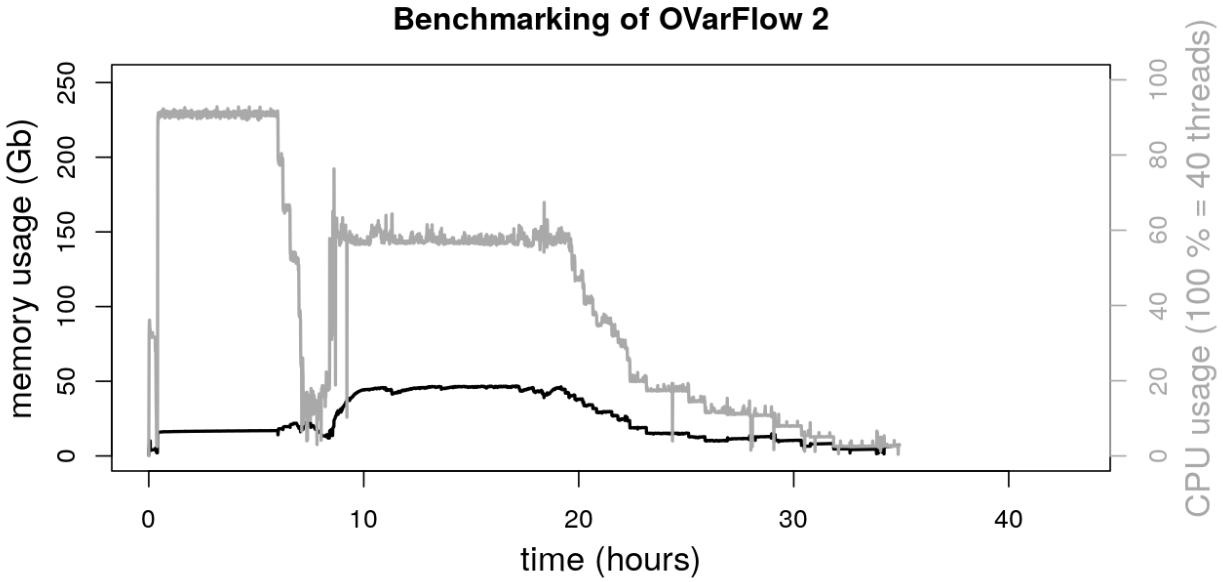
Reference organism	*Gallus gallus*
Reference genome	GCF_000002315.6 (GRCg6a)
Sequencing data	ERR1303580, ERR1303581, ERR1303584, ERR1303585, ERR1303586, RR1303587

Both workflows utilized the same yaml configuration file:

```yaml
heapSize:
    SortSam          : 10
    MarkDuplicates   : 2
    HaplotypeCaller  : 2
    GatherIntervals  : 2
    GATKdefault      : 12

ParallelGCThreads:
    SortSam          : 2
    MarkDuplicates   : 2
    HaplotypeCaller  : 2
    GatherVcfs       : 2
    CombineGVCFs     : 2
    GATKdefault      : 4

Miscellaneous:
    BwaThreads       : 6
    BwaGbMemory      : 4
    GatkHCintervals  : 4
    HCnpHMMthreads   : 4
    GATKtmpDir       : "./GATK_tmp_dir/"
    MaxFileHandles   : 300
    MemoryOverhead   : 1

Debugging:
    CSV              : False
    YAML             : False
```

The actual computations were performed on a single cluster node (SGE) reserved exclusively for OVarFlow. The cluster node provided 20 cores / 40 threads (Intel Xeon E5-2670) and 251.9 Gb of main memory.





The total runtime of both workflows was:

- OVarFlow 1: 42.675 h
- OVarFlow 2: 34.91 h

Thereby, the total runtime of OVarFlow 2 could be reduced by about 22 % compared to OVarFlow 1. Of course, this value is very specific for the given hardware, sequencing data, reference genome, and settings mentioned above. By utilizing even more intervals in parallel, further runtime reduction might be possible. However, a time saving of 20 % by utilizing OVarFlow 2 is a reasonable estimate.

3.17 GATK Pitfalls

Variant calling is no trivial task, involving many different applications. Not only is the usage and interaction of those applications often far from obvious, but also especially some GATK tools possess some very intricate pitfalls. Dealing with those for the first time can be very time consuming. During the development of OVarFlow many of those pitfalls were unraveled and are now taken care of by the workflow. Still the most annoying and time consuming issues shall be explained here.

gatk SortSam `--TMP_DIR` SortSam creates a lot of temporary files of the form `/tmp/<user_name>/`` `sortingcollection.nnnnnnnnnnnnnnnnnnn.tmp` (with n being any number). Depending on the size of the input data this can easily add up to dozens of gigabytes, thereby quickly consuming space under `/tmp`. Depending on the partition scheme further problems can result from this. Also in case of abortion of SortSam, those temporary files are not automatically deleted. In those cases manual deletion or user lockout might be required.

To circumvent this problem OVarFlow stores temporary files under `/path/to/project_dir/GATK_TMP_DIR/`. This is archived by SortSam's `--TMP_DIR <directory>` option. As the project directory has to provide reasonable amounts of storage, large temporary files shouldn't cause any problems here.

gatk MarkDuplicates `--TMP_DIR` Just like SortSam MarkDuplicates creates a lot of temporary files. MarkDuplicates will create a directory of its own, like `/tmp/<user_name>/CSPI.nnnnnnnnnnnnnnnnnn.tmp/` (with n being any number). Inside this directories a lot of different files are created.

Again the option `--TMP_DIR <directory>` was utilized, to redirect the creation of temporary files to `/path/to/project_dir/GATK_TMP_DIR/`.

gatk MarkDuplicates `-ASO | --ASSUME_SORT_ORDER` MarkDuplicates offers an option to specify the given sort order of the input data, including `unsorted`. Surprisingly there is an error message when using this option:

> *picard.PicardException: This program requires input that are either coordinate or query sorted (according to the header, or at least ASSUME_SORT_ORDER and the content.) Found ASSUME_SORT_ORDER=unsorted and header sortorder=unsorted*

Therefore the previous step of using GATK SortSam is obligatory. As a side note, even though no in-depth investigation was performed, sorting with `samtools` also was causing issues. So sticking to SortSam is recommended.

gatk MarkDuplicates `-MAX_FILE_HANDLES` A final issue with MarkDuplicates is that it opens a plethora of file handles. Depending on the setup of the respective operating system, this can cause very subtle difficulties. The maximum number of allowed open file descriptors may be too small for MarkDuplicates. The current limits may be displayed via `ulimit -Hn` (hard limit) and `ulimit -Sn` (soft limit). Often 4096 is a given limit. In OVarFlow the option `-MAX_FILE_HANDLES` was set to 300.

gatk HaplotypeCaller and AVX instruction GATK was heavily optimized to make use of the given instruction set of the respective CPU. Therefore run times can be heavily influenced by the presence of CPU instructions (e.g. SSE, SSE2, AVX, AVX512). This was also mentioned in the Video of the GATK4: Live Launch Invent (approx. at 23 min). Some performance optimizations were also achieved by a cooperation with Intel engineers:

> *This resulted in performance optimizations with improvements for PairHMM, used in Haplotype caller, for Intel® Xeon® processors with Intel® Advanced Vector Extensions 512 (Intel® AVX 512) and Intel FPGAs.*

From personal experience (no systematic assessment), the absence of AVX will cause approx. five times longer runtimes of HaplotypeCaller. As this is a matter of days or even weeks, OVarFlow will refuse to run on a CPU without AVX.

gatk HaplotypeCaller parallelization In previous versions of GATK HaplotypeCaller an option for parallelization was available (`-nct` and `-nt`). This is no longer the case with GATK 4. To achieve multithreading with GATK

4, the GATK team recommends the use of (Apache) Spark. Unfortunately at the time of writing (June 2020, GATK 4.1.7) the Spark version of HaplotypeCaller is still BETA, with the official waring:

> *Use the non-spark HaplotypeCaller if you care about the results.*

To circumvent this problem, OVarFlow a so called scatter gather approach is used by OVarFlow. This allows for the automatic parallel evaluation of a user defined number of intervals per analyzed individual.

Java / GATK `-XX:ParallelGCThreads=<2-4>` The Java virtual machine (JVM) will automatically set the number of garbage collection (GC) threads, depending on the number of available CPU cores or threads, respectively. The performance of some GATK tools is heavily impacted by the number of GC threads. Performance is clearly declining at higher thread counts. The best performance is seen at 2-4 Java GC threads. A detailed analysis can be found within the *Java GC benchmarking section* of this documentation.

gatk GatherVcfs and suffix of input data The HaplotypeCaller generates a Genomic VCF file, abbreviated as GVCF. Sometimes the file suffix `.gvcf` is used for such files (or `.gvcf.gz` for the compressed data). GatherVcfs will not accept a GVCF file if the suffix is `.gvcf`, it explicitly has to be `.g.vcf` (or `.g.vcf.gz`). The same file with the wrong ending will produce a quite uninformative Java stack trace:

```
java.io.UncheckedIOException: java.nio.charset.MalformedInputException: Input␣
↪length = 1
        at java.io.BufferedReader$1.hasNext(BufferedReader.java:574)
        at java.util.Iterator.forEachRemaining(Iterator.java:115)
        ...
```

Also the order of the contigs in the individual GVCF files has to be identical to the order of the contigs as defined in the VCF header section. Fortunately the error message is more informative here:

```
ERROR   2020-11-09 14:03:20     GatherVcfs      There was a problem with gathering␣
↪the INPUT.java.lang.IllegalArgument
Exception: First record in file /path/to/file/interval_2.g.vcf.gz is not after␣
↪first record in previous file /path/to/file/interval_1.g.vcf.gz
```

To avoid the above problems CombineGVCFs can be used instead of GatherVCFs, but the merging of interval files will take considerably longer and costs more resources.

Java / GATK `-Xmx<value>g` Just as default Java GC thread numbers depend on the amount of given cores, the default heap allocation by the JVM depends upon the amount of given system memory. The default values of a specific machine can be determined via `java -XshowSettings:vm`. This is of importance as memory usage of some GATK tools is dependent upon the *Max. Heap Size* value. Higher values will result in higher memory consumption, thereby potentially limiting the amount of parallel processes. In the worst case the system might even run out of memory, ultimately leading to termination of processes.

GATK `--tmp-dir` Many GATK applications will create files like this `/tmp/tmp_read_resource_nnnnnnnnnnnnnnnnnnn.config`, where `n` denotes any number. Those files won't be deleted upon program termination. It appears like tools derived from Picard applications won't create such files but "native" GATK applications like HaplotypeCaller, CombineGVCFs, SelectVariants and so forth. The creation of such files can also be redirected to a different directory, but the syntax of this option for "native" GATK tools is a little different `--tmp-dir <directory>`, with lower case letters as opposed to `--TMP_DIR` for Picard tools.

fastqc `-Xmx1g` Although not part of GATK, the tool `fastqc` has pitfalls of its own. Interestingly for small fastq files (approx. 50 MB gzip fastq test file) the memory allocation pool has to be set explicitly. Otherwise the Java VM aborts with the following error:

```
...$ fastqc -o testFASTQC -f fastq FASTQ_INPUT_DIR/GGA081_R1.fastq.gz
Started analysis of GGA081_R1.fastq.gz
```

(continues on next page)

(continued from previous page)

```
Approx 5% complete for GGA081_R1.fastq.gz
Approx 10% complete for GGA081_R1.fastq.gz
Approx 15% complete for GGA081_R1.fastq.gz
Approx 20% complete for GGA081_R1.fastq.gz
Exception in thread "Thread-1" java.lang.OutOfMemoryError: Java heap space
        at uk.ac.babraham.FastQC.Utilities.QualityCount.<init>(QualityCount.java:33)
        at uk.ac.babraham.FastQC.Modules.PerTileQualityScores.
↪processSequence(PerTileQualityScores.java:281)
        at uk.ac.babraham.FastQC.Analysis.AnalysisRunner.run(AnalysisRunner.java:88)
        at java.lang.Thread.run(Thread.java:745)
```

Therefore before invoking `fastqc` it is advisable to set an environment variable for the Java VM like this: `export _JAVA_OPTIONS='-Xmx1g -XX:ParallelGCThreads=4`. It is also advisable to reduce the number of Java GC threads.

snpEff OVarFlow performs functional annotation of the identified variants using `snpEff`. To perform variant annotation this tool makes use of a database of the annotated genome. Even though this is not a bug but a feature, one should be aware of the fact, that snpEff normally stores this database some within the users home directory. As those databases can consume considerable amounts of space within the users home. Therefore OVarFlow changes this default behavior and stores the snpEff database within the project directory. Keeping all project specific data together in one place is advisable anyways.

3.18 Citation

If you're using OVarFlow or if the documentation provided here was helpful in your own work, please consider citing OVarFlow. The final peer-reviewed publication is:

Bathke, J., Lühken, G. OVarFlow: a resource optimized GATK 4 based Open source Variant calling workFlow. BMC Bioinformatics 22, 402 (2021). https://doi.org/10.1186/s12859-021-04317-y.

A preprint was made available at the BioRxiv:

Jochen Bathke, Gesine Lühken. OVarFlow: a resource optimized GATK 4 based Open source Variant calling workFlow. 2021. https://doi.org/10.1101/2021.05.12.443585.

3.19 License

- The source code of OVarFlow itself is licensed under the terms of the GPLv3.
- The accompanying documentation of OVarFlow (the document you're reading right now) is licensed under the terms of the Creative Commons license CC-BY SA 3.0, either in your national translation or if not applicable in the global version.

3.20 Contact

OVarFlow has been developed at the Professorship of Pet and Pathogenetics at the Justus-Liebig-University Giessen. In case of comments or questions about OVarFlow, contact the institute. Due to high amounts of spam e-mail addresses cannot be posted publicly.

3.21 Repository

The source code of OVarFlow and its accompanying documentation can be found at:

- GitLab

Prebuild containers with all nessessary software components can be found at:

- Docker hub (Docker images)
- Zenodo.org, doi: 10.5281/zenodo.4746639 (Singularity containers)

3.22 Change Log

3.22.1 2.0

Substantial performance improvements were achieved by applying parallelization strategies to additional steps of the workflow.

- A new Snakefile for OVarFlow 2.0 was created, rearranging the workflow (CombineGVCFs and GenotypeGVCFs were both parallelized).
- Preexisting GVCF files won't be used by OVarFlow 2.0.
- Comprehensive documentation of OVarFlow 2.0 was created.
- Flowchart comparing OVarFlow 1 and OVarFlow 2 was added to the documentation.
- Benchmarking to compare the performance of OVarFlow 1 and 2.

3.22.2 1.2.0

- Improved scheduling of bwa memory usage (especially useful for Slurm).
- Documentation for bwa memory scheduling added.
- Improved documentation for memory scheduling via config.yaml.
- Two new alternative target rules added to the workflow.
- Documentation provided for these new target rules.
- Minor corrections to the documentation, such as typos and wording.

3.22.3 1.1.0

- Improved scheduling of memory usage (especially useful for Slurm usage).
- Documentation for Slurm usage added.
- Java memory overhead enabled for better resource scheduling (affects Snakefile, config.yaml and documentation).
- Some refactoring of the Snakefile (mainly formatting).
- Change log added to documentation.
- Citation of final publication added.

3.22.4 1.0.1

Various improvements were added to the documentation. No changes to the workflow.

- New section about error identification added.
- Better explanation of snakemake options -j and –cores.
- bioRxiv link added.
- Small fixes, like typos.

3.22.5 1.0

- Initial release of OVarFlow, including the “normal” variant calling workflow and the optional BQSR workflow.