Calling short variants with GATK4: exercise instructions for BioHPC Cloud computers

Workshop contact

bukowski@cornell.edu

Data used in the exercise

We will use *D. melanogaster* WGS paired-end Illumina data with NCBI accessions SRR1663608, SRR1663609, SRR1663610, SRR1663611, corresponding to samples ZW155, ZW177, ZW184, and ZW185 respectively. To speed up the calculations, the data, originally at about 10 million read pairs per sample, has been down-sampled by 50%.

The exercise consists of several steps leading from WGS Illumina reads to four-sample variant calls, as specified in GATK Best Practices. To call variants in all four samples, all steps from read alignment through haplotype calling need to be performed for each of the four FASTQ file pairs. On a multi-CPU machine with large memory, such runs could be launched in parallel (for example, by manually submitting a given step in the background, each time for different sample). For the purpose of the exercise, please run all these steps for at least one sample of your choice. All intermediate results for other samples, needed in the final joint variant calling step, have been pre-computed and can be copied from the workshop directories (see detailed instructions below).

Log in to your workshop machine

The machine allocations are listed on the workshop website: https://biohpc.cornell.edu/ww/machines.aspx?i=112.

Details of the login procedure using ssh or VNC clients are available in the document https://biohpc.cornell.edu/lab/doc/Remote_access.pdf.

Use your **ssh client** with BioHPC Lab credentials to open an ssh session. If you wish, you can open multiple sessions to have access to multiple terminal windows (useful for program monitoring).

Alternatively, use the **VNC client** to open a VNC graphical session (you will need to first start the VNC server on the machine from "**My Reservations**" page reachable from https://biohpc.cornell.edu after logging in to the website. To close the VNC connection, click on the "X" in top-right corner of the VNC window (but **DO NOT log out!**). This will ensure that your session (all windows, programs, etc.) will keep running so that you can come back to it by logging in again.

General tips

Examine all the provided shell scripts (`*.sh`), for example, opening them in a text editor (such as `nano` or `gedit`). Read explanatory comments. Notice the use of environment variables to simplify and generalize scripts. For example, following the definition of a variable `ACC`

```
ACC=SRR1663609
```

any time **$ACC** or **${ACC}** appears in the script, it will be interpreted as **SRR1663609**. Note the technique of breaking long lines into smaller pieces terminated with the "\" character. For bash this is still a long line, but easier to read for us.

Monitor the progress of your activities using the top command, preferably run in a separate window:

```
top –u <yourID>
```

This will show dynamically updated list of your processes, with the most active ones on top. Since the GATK tools are written in Java, the process you will see most will be Java virtual machine called **java**. In the alignment stage, the process to look for will be the BWA aligner called **bwa**. The absence of any active processes (consuming CPU time) on your top list will indicate completion (or crash) of any scripts you were running. Pay attention to memory usage (%MEM column) of different runs.

Peek into the log files. Each time a script is run, the screen output from all commands is saved into a log file (say, **script.log**). Although the messages written to that file may sound cryptic at times, they generally allow the user to figure out which stage of the calculation is running at the moment. It also contains useful timing information (start and end dates of individual stages, elapsed time, ETA time). To look into the log file, you can use any of the following commands

`more script.log`	(page through the file from the beginning)
`tail -100 script.log`	(display the last 100 lines of the file)
`tail -f script.log`	(continuously display incoming lines)

Of course, you can also look at the whole file by opening it in a text editor. Upon exit, **discard any changes** you may have inadvertently made.

Look into the working directory (/workdir/<yourID>). As the run progresses, various intermediate files are being produced. Executing ls –al once in a while will allow you to see those files and how they increase in size.

If you can't see the expected output file even though it seems that the script has ended, it usually means that something went wrong. Examine the screen log file (e.g., open it in text editor) looking for error messages.

You can disconnect. If a step takes longer than you are willing to wait, you can disconnect from your VNC session (click on the cross in top right corner of the VNC window, but do **not** "Log Out"!). All your programs and windows will continue running and you can examine the results when you reconnect at a later time. Also, if you are working via ssh client (rather than VNC), you can safely log out of your ssh session as long as the script you are running was submitted in the **background** through **nohup** (as recommended throughout this exercise). The script will still be running (or will have finished) next time you log in to the machine.

Fetch input files and scripts to your local scratch directory

If not yet done, create your subdirectory in the scratch file system **/workdir**. In the following, we will assume the user ID **<yourID>** – please replace it with your own user ID.

```
cd /workdir
mkdir <yourID>
cd <yourID>
mkdir tmp
```

(The last line will create a temporary directory where Java will be instructed to store its scratch files.) Copy the exercise files from the shared location to your scratch directory (it is essential that all calculations take place here):

```
cp -L /shared_data/Variants_workshop_2018/* .

cp -r /shared_data/Variants_workshop_2018/genome .
```

When the copy operation completes, verify by listing the content of the current directory with the command **`ls -al`**. You should see read files:

```
SRR1663608_thinned_1.fastq.gz
SRR1663608_thinned_2.fastq.gz
SRR1663609_thinned_1.fastq.gz
SRR1663609_thinned_2.fastq.gz
SRR1663610_thinned_1.fastq.gz
SRR1663610_thinned_2.fastq.gz
SRR1663611_thinned_1.fastq.gz
SRR1663611_thinned_2.fastq.gz
```

Check the file sizes – they should be around 410-430 MB each. Along with the read files, several shell scripts **`*.sh`** are provided, corresponding to various stages of the pipeline. The subdirectory **`genome`** contains a FASTA file **`genome.fa`** with the *D. melanogaster* reference genome. The files **`*.vcf`** contain information about known variant sites – we will use them in the base quality recalibration stage.

Check the sequencing quality (optional)

This step summarizes the sequencing quality of the data. It is recommended to run this step before starting the assembly – it may help set the read trimming parameters for Trinity run.

```
cd /workdir/<yourID>
mkdir qcreport
fastqc -o qcreport *.fastq.gz
```

- **`-o qcreport`** : specify the output directory (./**`qcreport`**) where the QC reports will be stored, on directory per fastq file.
- All the fastq files should be specified, separated by space " ". The wildcard * also does the job.
- After it is done, you can use any sftp client (e.g., **FileZilla**) to copy the **`qcreport`** directory to you laptop computer, and open the **`fastqc_report.html`** file in each subdirectory with a web browser. If you are working in graphical environment (i.e., via VNC), you can launch the Firefox browser directly on your Linux workstation and navigate to **`fastqc_report.html`** files.

Prepare reference genome

The file **genome.fa** in subdirectory genome is the reference we will be aligning the reads to. Before starting the pipeline, the genome needs to be indexed for the BWA aligner. Also, length information for all chromosomes needs to be summarized.

All this can be done by running the shell script **prepare_genome.sh**. Run the script as follows:

```
nohup ./prepare_genome.sh >& prepare_genome.log &
```

The script will run in the background and should take a few minutes to complete (use **top –u <yourID>** in a separate window to monitor your processes). After it completes, list the content of subdirectory **genome**. The files **genome.fa.fai** and **genome.fa.dict** are simple text files summarizing chromosome sizes and starting byte positions in the original FASTA file. The other files constitute the BWA index.

Align reads to reference

The script **aln_bwa.sh** will start the **bwa** aligner for accession **SRR1663609** (sample ZW177). Look at the script (e.g., with the **nano** editor). In particular, note the definition of **read group** in bwa command line (bwa will insert it into the alignment file it generates). Note also that the raw output from the **bwa mem** command (normally written to standard output STDOUT in human-readable **SAM** format) is piped (using the "**|**" syntax) into another command, **samtools view -Sb**, which converts it right away into a compressed binary file in **BAM** format, equivalent to SAM, but several times smaller. Thanks to this piping mechanism, the large, intermediate output file from **bwa mem** does not need to be written to disk.

In our exercise, each sample is sequenced once on one Illumina lane.

Start the script (in **/workdir/<yourID>**):

```
nohup ./bwa_aln.sh >& bwa_aln_SRR1663609.log &
```

The program will run in the background, saving any screen output to the log file we decided to name **bwa_aln_SRR1663609.log**. Approximate run time: **10 minutes**.

The result will be the file **SRR1663609.bam**, describing the alignments in **BAM format**. This file is a binary file. Briefly examine its contents using the command **samtools** which reads this binary file and converts it to human-readable text format, printing it to the screen:

```
samtools view -h SRR1663609.bam | less
```

Piping the output from **samtools** through **less** allows us to view this output page by page; to exit the paginator - press **q**. The option -h make **samtools** print the entire content of the BAM file, including the header.

If you wish, repeat the run with different sample(s) (it will come in handy in the next session, when we will be calling variants from multiple samples). To do this, edit **bwa_aln.sh**, and change the SRR

accession and the corresponding sample name where needed (don't forget the read group declaration!), and re-run the script. If you wish, you can include all four alignment commands in the script, to be executed one by one.

Sort and mark duplicates

The BAM file obtained previously will now be sorted over genomic coordinate and any duplicate fragments will then be marked to be ignored in downstream analysis. To do this, launch the script **`sort_dedup_index.sh`**:

```
nohup ./sort_dedup_index.sh SRR1663609 >& sort_dedup_index_SRR1663609.log &
```

The script takes one argument (the SRR accession). To re-run it for a different accession, just change the argument in the command above – there is no need to change anything in the script itself. However, edit the script and examine the syntax of the commands and any informative comments it may contain. The run for one sample will take approximately **10 minutes**. Examine the log file for any errors and detailed timing information. It is actually convenient to monitor the log file during the run (**`tail –f sort_dedup_index_SRR1663609.log`**).

The result of the run will be the file **`SRR1663609.sorted.dedup.bam`**, accompanied by the index file **`SRR1663609.sorted.dedup.bai`**.

Check the alignment stats summary of the obtained file, running the **`samtools`** command

```
samtools flagstat  SRR1663609.sorted.dedup.bam
```

Can you tell how many reads have been mapped? How many have been marked as duplicate?

Base quality score recalibration

The script **`recalibrate.sh`** first collects the mismatch data from alignments and then uses this information to re-calibrate base quality scores of reads in the BAM file obtained in the previous step. It is important that the mismatch data is collected from a BAM file in which reads have been locally re-aligned around indels, since these alignments are less likely to contain false mismatches. Any well-known variant positions should be excluded from the mismatch count. We have such known variant position sets for the four chromosomes: **chr2L, chr2R, chr3L**, and **chr3R** (in files **`chr*.vcf`** provided with the exercise data). These are specified in the **`BaseRecalibrator`** command (using **`–knownSites`** option). **`BaseRecalibrator`** is only using alignments from these four chromosomes (as specified by the **`–L`** options). This way, any unknown but true variant sites of other chromosomes will not skew the recalibration database. The database is then applied in the second step to the full alignment file.

Before the known variant sites can be used in recalibration procedure, the corresponding vcf files have to be *indexed*. This can be accomplished by running the script **`index_vcfs.sh`**, which uses the GATK's indexing tool **`IndexFeatureFile`**:

```
./index_vcfs.sh >& index_vcfs.log &
```

When the command above completes, you should see four new files: **chr2L.vcf.idx**, **chr2R.vcf.idx**, **chr3L.vcf.idx**, and **chr3R.vcf.idx**. VCF file indexing will take just a few minutes and needs to be done only once before running the base score recalibration procedure.

To launch the recalibration procedure for sample **SRR1663609**, execute

```
nohup ./recalibrate.sh SRR1663609 >& recalibrate_SRR1663609.log &
```

The output consists of two files: the recalibration table **SRR1663609.sorted.dedup.recal_data.table** and the recalibrated BAM file **SRR1663609.sorted.dedup.recal.bam**. Estimated run time: **7 minutes** (per sample).

Visualize the alignments using IGV

You need to be connected to the machine with VNC. In a terminal window, enter

```
igv
```

The IGV window will appear in a few moments. In the navigation bar, go to Genomes and select **Load Genome from File**. Browse to **/workdir/<yourID>/genome** and select the file **genome.fa**. I the chromosome selection dropdown (which initially says **All**) you should see *D melanogaster* chromosome names. Now load one of the BAM files created during the exercise: go to **File → Load from File**, browse to **/workdir/<yourID>**, and select one of the BAM files. Select one of the chromosomes and zoom in (using the slider in top right corner) enough to see alignments.

IGV has plenty of display options, some available in **View** menu tag, some showing up upon a right click within the display window. Try to familiarize yourself with the program's capabilities. For example, try different read coloring schemes based on various properties of aligned pairs (right click within window, then select **Color alignments by**…). Notice that left-clicking on (or just hovering over – it is configurable) a read will display detailed information about it, as found in the BAM file. You may find it helpful to use IGV's **Help** link to learn more.

Run GATK HaplotypeCaller on individual samples

In this step, we will run the **HaplotypeCaller** on our BAM files to produce genotype likelihood information for each sample for each locus in the genomic region of interest. Typically, this region would be the whole genome. To save time, we will concentrate on one chromosome, **chr2R** (see the **–L** option in the **hc.sh** script). To launch the calculation for one of the sample (e.g., **SRR1663609**), type

```
nohup ./hc.sh SRR1663609 >& hc_SRR1663609.log &
```

The screen output from the command will be written to the log file specified above. The expected result will be the file **SRR1663609.g.vcf**. containing the intermediate genotyping data for this sample.

The estimated run time of this step is 1 hour, and it has to be repeated for the other 3 samples. In real life, calculation for different samples would be run concurrently as different processes on the same multi-core machine, or on separate machines. It is also possible to parallelize the calculation over

genomic coordinate, i.e., run separate jobs per sample per (a piece of) a chromosome (which can be specified with the **-L** option of **HaplotypeCaller** – see **hc.sh** script).

Parallelism can also be achieved using a different version of haplotype caller (as well as other tools), parallelized using a scheduler called **Spark**, seamlessly bundled with GATK4 package. This version, called **HaplotypeCallerSpark** can be used instead of **HaplotypeCaller** and accepts the same command-line options. In addition, options controlling multithreading can also be used. For example, specifying **--spark-master local[4]** on command line will request the run to use 4 CPU threads in parallel, resulting in faster execution. At this point the Spark-parallelized components of GATK4 are still in development, so - following the GATK website - we cannot yet recommend them for production runs. However they are expected to become mainstream in near future. For now, we provide a Spark-based variant of haplotype caller script, **hc_spark.sh**, which uses **HaplotypeCallerSpark** and runs it on 4 threads, producing **g.vcf** files with names pre-foxed by **Spark_** (to distinguish them from the **.g.vcf** files obtained using the regular **HaplotypeCaller**).

For the purpose of the exercise, we would recommend that you run this step using both **hc.sh** and **hc_spark.sh** scripts for at least one of the samples, e.g., the one for which you prepared the BAM file in previous section. Any BAM (and bai) files you may be missing can by fetched from the shared workshop directory, e.g.,

```
cp /shared_data/Variants_workshop_2018/processed_bams/SRR1663610*.ba* .
```

and similarly for other samples. If you are short of time, you can simply skip this calculation and fetch all the ready-made ***.g.vcf** files (see next Section). In any case, please examine the **hc.sh** and **hc_spark.sh** scripts.

Once the ***.g.vcf** file(s) are in your work directory, examine them (e.g., open in **nano** text editor – possible for short test files like these) and confront with the gVCF format description at https://www.broadinstitute.org/gatk/guide/article?id=4017.

Have you completed the HaplotypeCaller step for all four samples?

If yes, you should have generated four *.**g.vcf** files (along with the corresponding indexes) and can proceed to joint variant calling part of the exercise. If not, copy the pre-made *.**g.vcf** files and the corresponding index files ***.g.vcf.idx** to your directory now:

```
cp /shared_data/Variant_workshop_2018/premade_gvcf/*.g.vcf* .
```

Joint variant calling with GenotypeGVCFs

Before the intermediate, sample-level files ***.g.vcf** results can be used to call variants jointly on all four samples, they have to be merged into a single, multi-sample **g.vcf** file using the GATK4's tool **CombineGVCFs**. The script **combineGVCFs.sh** calling this command is run as follows:

```
nohup ./combineGVCFs.sh >& combineGVCFs.log &
```

and results in a merged file **`all.g.vcf`**.

Once the merged file is ready, the script **`genotypeGVCFs.sh`** uses the **`GenotypeGVCFs`** tool to perform the joint variant calling

```
nohup ./genotypeGVCFs.sh >& genotypeGVCFs.log &
```

As usual, the script will run in the background, saving all screen output to the log file. The script takes no arguments, since all the necessary information (sample IDs) is hard-coded in the script explicitly. The output (as specified in the GATK command line) is the file **`all.vcf`**, containing the raw (i.e., not yet filtered or recalibrated) variant calls for our 4-sample "population". Open this file in a text editor and examine its content. **Estimated run time: 5-6 minutes**.

Have you completed the BWA alignment step for all four samples?

If yes, you should have generated four **`*.bam`** files (along with the corresponding indexes **`*.bai`**) and can proceed to next part of the exercise, which is an example of using FreeBayes for joint variant calling. If not, copy the pre-made *.**`bam`** files and the corresponding index files **`*.bai`** to your directory now:

```
cp /shared_data/Variant_workshop_2018/processed_bams/*.ba? .
```

Joint variant calling using FreeBayes

FreeBayes is a variant-calling program by Erik Garrison et al., https://github.com/ekg/freebayes. It is independent from GATK. Similarly to GATK's **HaplotypeCaller**, **FreeBayes** uses haplotype-based approach to variant detection, although implemented differently.

The input for **FreeBayes** consists of alignment BAM files for all samples involved. Unlike GATK, the **FreeBayes**-based pipeline does not require extensive preparation of alignments, such as local re-alignment or base score recalibration. However, since such pre-processing won't hurt, we can use our processed BAM files obtained in the previous session (you can copy those from the workshop shared space, as described above). Assuming all BAM files are in place, call variants using the script **`fb.sh`**:

```
nohup ./fb.sh >& fb.log &
```

The result will be the variant file **`fb.vcf`**. **Estimated run time: 20 minutes**.

Examining the resulting VCF file, notice that the parameters in the ANNOTATION field generated by **FreeBayes** are generally different than those emitted by GATK callers.

Filter variants with VariantFiltration

The VCF files obtained in previous steps are raw results, likely to contain a lot of false positives, depending on the stringency options used in calling. Since the calling steps are time-consuming, it is

generally advisable to set these options to emit an inclusive set of variants, and then filter this set over various parameters, such as those recorded in the INFO field of a VCF file. In GATK, the option **`-stand_call_conf`** controls the lower threshold on the quality (the QUAL field of VCF) of variants to be output. This option should be set to some low value (e.g., 5).

Filtering of the raw set of variants can be accomplished using many different tools. In a lot of cases, one can simply utilize standard Linux text parsing tools, like **grep**, **awk**, or **sed**. For example, to extract a subset of variants with QUAL greater than, say, 60, from raw **`all.vcf`** we could use the following commands:

```
grep "#" all.vcf > all.qual60.vcf
grep -v "#" all.vcf | awk '{if($6>60) print}'>> all.qual60.vcf
```

The first command extracts the VCF header lines (containing "#") into a new VCF file, while the second command processes the non-header lines, appending them to the new file only if the sixth column (that's where QUAL is) is above 60.

GATK offers a tool called **VariantFiltration**, which allows more complex filtering patterns. An example script using this tool is called **`filter_vcf.sh`**. As you examine this script, you will notice that different filtering criteria are applied to SNPs than for indels. To accomplish this, SNPs and indels are first extracted to separate files, these files are filtered, and then the SNP and indel filtered files are merged back into a single filtered file. To run the filtering script, enter

```
nohup ./filter_vcf.sh all >& filter_vcf.log &
```

(note that we are supplying the prefix of the VCF file name, i.e., *without* the **`.vcf`** extension as argument). The filtered VCF file will be called **`all.filtered.vcf`** (the corresponding index file **`*.vcf.idx`** will also be created). Other intermediate files (with separate SNPs, indels, filtered and not, along with their indexes) will also be produced – these may be deleted.

Examine the filtered VCF file. Notice the change in the FILTER field. Instead of dot "." (no filtering information), this field will now contain flags **PASS** (variants which passed the filter) and **my_snp_filter** or **my_indel_filter** (both these strings were defined in filtering command) – marking variants which failed the respective filters. Note that **no variant has been removed from the file**. The ones that failed filtering are just marked as such.

Estimated run time: **3 minutes**.

Basic stats and comparison of variant sets

Using Linux commands

Given a VCF file, its simplest properties may be obtained by running standard Linux text parsing tools. For example, to get the number of variants in a file, run the following:

```
grep -v "#" all.vcf | wc -l
```

(**grep** filters out the header lines and pipes its output into **wc -l** which counts the remaining lines and displays the result on screen). To extract sites located between positons 10000 and 20000 on chromosome chr2R and save them in a file, simply run

```
grep –v ”#” all.vcf | awk ’{if($1==”chr2R” && $2 >=10000 && $2 <=20000) print}’ > extracted_records
```

(note that in this case the chromosome condition **$1==”chr2R”** is not really needed, because our VCF file only contains data for chr2R, however, it would be necessary for a more general input). To quickly find out how many variants passed filtering, simply type

```
awk ’{if($7==”PASS”) print}’ all.filtered.vcf | wc -l
```

More complex analysis and operations on VCF files can be accomplished using specialized software tools, such as those contained in GATK package and those from the **vcftools** package (independent from GATK).

Using GATK4's Concordance and GenotypeConcordance tools

GATK4 offers interesting functions, **Concordance** and **GenotypeConcordance**, to summarize various statistics of a variant set and compare it to another variant set obtained from the same data, but with a different method, for example. A script **var_compar.sh**, based on this function, will compare any two VCF files, e.g., **all.vcf** and **fb.vcf** (obtained using GATK4 and FreeBayes, respectively):

```
./var_compar.sh fb all ZW155 >& var_compar.log &
```

The command above will generate four file **fb.all.comp.site_summary** with the site concordance summary, and three other files (**fb.all.comp.genotype_concordance_contingency_metrics**, **fb.all.comp.genotype_concordance_detail_metrics**, **fb.all.comp.genotype_concordance_summary_metrics**) with more detailed analysis of genotype concordance for sample **ZW155**. Some of these files have long lines and are best viewed in Excel (after being transferred to your laptop).

Using vcftools

vcftools (A. Auton, A. Amrcketta, https://vcftools.github.io/index.html) is a popular toolkit for analyzing and manipulating VCF files. Here are some usage examples (try them on your VCF files):

Obtain basic VCF statistics (number of samples and variant sites):

```
vcftools --vcf all.vcf
```

Extract subset of variants (chromosome chr2R, between positions 1M and 2M) and write tem a new VCF file

```
vcftools –vcf all.vcf --chr chr2R --from-bp 1000000 --to-bp 2000000 --recode –recode-INFO-all -c > subset.vcf
```

Get allele frequencies for all variants and write them to a file

```
vcftools --vcf all.vcf --freq -c > all.freqs
```

Compare two VCF files (will print out various kinds of compare info in files **`fb.all.compare.*`**):

```
vcftools --vcf fb.vcf –diff all.vcf --out fb.all.compare
```