Variant calling

Practical exercises

1 Get ready

Before you start, make a directory tree as the one depicted below in your home area. The commands that you get throughout these exercises are written down based on a directory tree like this. After you have done this, move into the "snp" directory.

$HOME -> course -> workflow -> snp

The next step is to copy some files you are going to need from /work/projects/norbis/

We are going to work with two different .fastq files, MMR_1370_subject.fastq and MMR_664_control.fastq. These files contain reads from exome sequencing of two different mice, one with a mendelian disease, and one control. The files are in the directory /work/projects/norbis/data/ along with our reference file mm10s.fa, that contains the fasta sequence for chromosome 9 and 18 from the mouse genome.

```
cp -r  /work/projects/norbis/workflows/snp/data  .
```

During these exercises, we will use several bioinformatics programs, some of which are not available as modules on Abel. These can be found in the directory

/work/projects/norbis/workflows/snp/programs/

Copy this directory with contents to your snp directory. Use ls to check the contents of the copied directories.

2 Quality control

Before starting the analysis of the data, it is important to do quality control, to make sure the data is of good enough quality. We will use a program called FastQC for this. FastQC is available as a module on Abel.

Make sure you are in the snp directory, and load FastQC with this command:

```
module load fastqc
```

We will perform several tests using FastQC, to check what there is to gain by parallelizing it. FastQC will write the output to your directory of choice, provided that this directory already exists. Make these three directories to store the outputs in:

```
mkdir fastqc_test_t_no
mkdir fastqc_test_t2
mkdir fastqc_test_t6
```

First, run FastQC with no parallelization for the two files. Use the time command to check the running time, and the -o option to get the output in the right directory. Write the times from the time outputs into a table like the one below for all the tests. Notice what is printed to screen for all of them.

```
time fastqc \
data/MMR_1370_subject.fastq \
data/MMR_664_control2.fastq \
-o fastqc_test_t_no
```

Next, use the -t option to get two threads

```
time fastqc –t2 \
```

```
data/MMR_664_control2.fastq \
data/MMR_1370_subject.fastq \
-o fastqc_test_t2
```

Then, use the -t option to get 6 threads

```
time fastqc -t 6 \
data/MMR_664_control2.fastq \
data/MMR_1370_subject.fastq \
-o fastqc_test_t6
```

#Threads	real	user	sys
No threads			
2 threads			
6 threads			

Use this command to check if the results are the same for these runs:

```
diff fastqc_test_t_no/MMR_664_control_fastqc.html fastqc_test_t2/MMR_664_control_fastqc.html
```

Take a look at the resulting html file for each of the two .fastq files. Is it good enough?

You can find a thorough description of FastQC here:

http://www.bioinformatics.babraham.ac.uk/projects/fastqc/

Scroll down to the Documentation section and click on “copy of the FastQC manual”, then “Analysis modules” to get to the part where the different analysis modules are described.

Questions:

1. Why does it not help to add more threads than the number of files?

2. Why is the real time the only one that changes when we add threads?

3. Can you think of a way to parallelize FastQC for only one file?

3 Quality filtering

As we can see from the FastQC output, the quality of the data are not yet good enough. Variant calling is sensitive to bad quality, so before we start the analysis, we need to do some filtering. We will use the program Prinseq to do this. All the possible options for this program are listed here:

http://prinseq.sourceforge.net/manual.html#STANDALONE

Take a look at this, to figure out what we are actually doing to our data. Prinseq has quite a long running time, and no threads option. We do the first file without trying to parallelize. While this is running, start another session to handle the other file. Prinseq works on each read separately, which means we can divide the fastq file into several parts and run Prinseq on each of them, then put the results together when it is done. We will try this, and see if there is time to gain from it.

```
time perl programs/prinseq-lite-0.20.4/prinseq-lite.pl \
-fastq data/MMR_1370_subject.fastq \
-trim_left 10 \
-trim_qual_right 30 \
-min_len 50 \
-min_qual_mean 28 \
-max_len 160 \
-out_good MMR_1370_subject \
-out_bad MMR_1370_subject_filtered
```

Now, for the other file. To split this file, we will use the unix split command. This will take the file name and the number of lines to split it into as parameters. To find the number of lines to use, start by checking how many lines is in the file.

```
wc -l data/MMR_664_control.fastq
```

Next, devide this number of lines by 4. (We want to parallelize with 4). You can open python2 or R to do this, python is quicker. We will need the line number we use to be dividible by four, so if needed, increase the number until it is.

The split command should be run like this

split –l 3533056 data/MMR_664_control.fastq

This will create four smaller fastq files called xaa, xab, xac and xad. Now, run prinseq-lite.pl for each one of them, using & at the end of the command for it to be run in the background.

```
time perl programs/prinseq-lite-0.20.4/prinseq-lite.pl \
-fastq xaa \
-trim_left 10 \
-trim_qual_right 30 \
-min_len 50 \
-min_qual_mean 28 \
-max_len 160 \
-out_good xaa_good \
-out_bad xaa_filtered &
```

For the other 3 files, run the same command, remember to change the filenames. (Use arrows to get the last command, it saves time.) When all 4 processes are finished, use cat to merge the 4 results files into one

```
cat xa?_good* > MMR_664_control.fastq
```

Questions:

1. Why did we need the line number to be dividible by 4?

2. What's your opinion on the overhead in this parallelization exercise?

2. Can you think of a way to parallelize the Prinseq code?

3. Do you think it would be useful?

4 Alignment

Now that our data are ready to use, we can start the alignment. We will use BWA (Burrows-Wheeler Alignment Tool) for this. It contains several alignment algorithms, we will use the one called mem. The BWA manual can be found here:

http://bio-bwa.sourceforge.net/bwa.shtml

BWA is available as a module on Abel use this command to load it:

```
module load bwa
```

The first step when using BWA is to make an index from the reference fasta file. Move into your data directory, and make the index with the following command:

```
time bwa index mm10s.fa
```

When the index is ready, we can start BWA-MEM. There is no option for parallelization when indexing, but the mem algorithm includes the possibility of using threads. We will use 7 threads when running. It will still take time, so use the chance to take a look at the manual.

Move back to the snp directory and run these two commands:

```
time bwa mem -t 7 -M -c 3000 \
data/mm10s.fa \
MMR_1370_subject.fastq \
> MMR_1370_subject.sam
```

```
time bwa mem -t 7-M -c 3000 \
data/mm10s.fa  \
MMR_664_control.fastq \
> MMR_664_control.sam
```

Sample name	real	user	sys
MMR_1370_subject			
MMR_664_control			

Questions:

1. Is BWA a mapper or an aligner?

2. How do you think the BWA-MEM algorithm is parallelized?

5 Preprocessing for GATK

Before we can do the variant calling, we have to do several preprocessing steps. These steps are done with Picard, samtools and GATK.

5.1 Sorting

The next step is to mark duplicates, but to do that, we need a coordinate sorted bam file. A bam file is the same as a sam file, but in binary format. To do the sorting and file type conversion, we use SortSam from Picard-tools. Use & to run the two files in parallel. These commands will give the output file we need:

```
time java -jar -Xmx5120M programs/picard-tools-2.2.1/picard.jar \
SortSam \
INPUT= MMR_1370_subject.sam \
OUTPUT=MMR_1370_subject_sorted.bam \
SORT_ORDER=coordinate

time java -jar -Xmx5120M programs/picard-tools-2.2.1/picard.jar \
SortSam \
INPUT= MMR_664_control.sam \
```

```
OUTPUT=MMR_664_control_sorted.bam \
SORT_ORDER=coordinate
```

5.2 Mark duplicates

Now we have what we need to mark the duplicate reads. This is done with MarcDuplicates from Picard-tools.

```
time java -jar -Xmx5120M programs/picard-tools-2.2.1/picard.jar \
MarkDuplicates \
INPUT=MMR_1370_subject_sorted.bam \
OUTPUT=MMR_1370_subject_sorted_dedup.bam \
METRICS_FILE=MMR_1370_subject_dedup_metrics.pic \
CREATE_INDEX=TRUE

time java -jar -Xmx5120M programs/picard-tools-2.2.1/picard.jar \
MarkDuplicates \
INPUT=MMR_664_control_sorted.bam \
OUTPUT=MMR_664_control_sorted_dedup.bam \
METRICS_FILE=MMR_664_control_dedup_metrics.pic \
CREATE_INDEX=TRUE
```

5.3 Creating .fai and .dict

Before continuing, we also need to make a .fai index and a .dict from the reference file mm10s.fa. The .fai index is made with samtools, which is available as a module on Abel. Load the module:

```
module load samtools
```

And make the index:

```
samtools faidx mm10s.fa
```

To create the dictionary, we use CreateSequenceDictionary from Picard-tools

```
java -jar -Xmx5120M programs/picard-tools-2.2.1/picard.jar \
CreateSequenceDictionary \
R=data/mm10s.fa \
O=data/mm10s.dict
```

5.4 Adding readgroups

For the realignment modules to work properly, the BAM files needs to have the right readgroups added to them. We do this with AddOrReplaceReadGroups from Picard-tools.

```
time java -jar -Xmx5120M programs/picard-tools-2.2.1/picard.jar \
AddOrReplaceReadGroups \
INPUT=MMR_1370_subject_sorted_dedup.bam \
 OUTPUT=MMR_1370_subject_sorted_dedup_addrg.bam \
RGID=MMR_1370_subject \
RGLB=MMR_1370_subject \
RGPL=Illumina \
RGPU=Illumina \
RGSM=MMR_1370_subject \
CREATE_INDEX=TRUE

time java -jar -Xmx5120M programs/picard-tools-2.2.1/picard.jar \
AddOrReplaceReadGroups \
INPUT=MMR_664_control_sorted_dedup.bam \
OUTPUT=MMR_664_control_sorted_dedup_addrg.bam \
RGID=MMR_664_control \
RGLB=MMR_664_control \
RGPL=Illumina \
```

```
RGPU=Illumina \
RGSM=MMR_664_control \
CREATE_INDEX=TRUE
```

5.5 Finding realignment targets

The files are now ready to be processed by GATK RealignmentTargetCreator, to find the areas where the reads should be realigned.

```
time java -Xmx4g -jar programs/GenomeAnalysisTK.jar \
-T RealignerTargetCreator \
-I MMR_1370_subject_sorted_dedup_addrg.bam \
-R data/mm10s.fa \
-o MMR_1370_subject_realignment_targets.intervals
```

```
time java -Xmx4g -jar programs/GenomeAnalysisTK.jar \
-T RealignerTargetCreator \
-I MMR_664_control_sorted_dedup_addrg.bam \
-R data/mm10s.fa \
-o MMR_664_control_realignment_targets.intervals
```

5.6 Realigning

With the files ready and the realignment target intervals in place, we can use GATK IndelRealigner to do the actual realignment.

```
time java –Xmx4g –jar programs/GenomeAnalysisTK.jar \
–T IndelRealigner \
-R data/mm10s.fa \
-I MMR_1370_subject_sorted_dedup_addrg.bam \
-targetIntervals MMR_1370_subject_realignment_targets.intervals \
-o MMR_1370_subject_realigned.bam

time java –Xmx4g –jar programs/GenomeAnalysisTK.jar  \
–T IndelRealigner \
-R data/mm10s.fa \
-I MMR_664_control_sorted_dedup_addrg.bam \
-targetIntervals MMR_664_control_realignment_targets.intervals \
-o MMR_664_control_realigned.bam
```

5.7 Build BAM index

Now that the files have changed again, BAM indexes need to be rebuilt.

```
time java -jar -Xmx5120M programs/picard-tools-2.2.1/picard.jar
BuildBamIndex \
INPUT=MMR_1370_subject_realigned.bam

time java -jar -Xmx5120M programs/picard-tools-2.2.1/picard.jar
BuildBamIndex \
INPUT=MMR_664_control_realigned.bam
```

Questions:

1. If you were to implement a new MarkDuplicates with an option to parallelize, what would you do?
2. How wold you parallelize a sorting algorithm?

6 Variant calling

We are now completely finished with the rather tedious, but neccesary, preprocessing. It's time to do the actual variant calling, and we use GATK HaplotypeCaller to do this part. The variant calling without parallelization will take about 20 minutes for each of our files, but GATK has options for threads, so we will use –nct 4 to speed things up. While waiting for your results, take a look at the documentation here:

https://www.broadinstitute.org/gatk/guide/tooldocs/org_broadinstitute_gatk_tools_walkers_haplotypecaller_HaplotypeCaller.php

```
time java -Xmx4g -jar programs/GenomeAnalysisTK.jar
-T HaplotypeCaller \
-nct 4
-R data/mm10s.fa \
-I MMR_1370_subject_realigned.bam \
-o MMR_1370_subject.raw.snps.indels.vcf

java -Xmx4g -jar programs/GenomeAnalysisTK.jar
-T HaplotypeCaller \
-nct 4 \
-R data/mm10s.fa \
-I MMR_664_control_realigned.bam \
 -o MMR_664_control.raw.snps.indels.vcf
```

We now have a set of called variants. This set is quite large, and we can expect that there are a lot of false positives, as GATK aims to be very sensitive. This means we will need to filter the variants. If possible, one should use variant quality score recalibration (VQSR) for this. This is a two step process using GATKs VariandRecalibrator and ApplyRecalibration. VQSR requires a sufficient number of known variation sites from well curated data. This kind of data is not necessarily available. Also, if the set you are working with is too small, VQSR can not be used. The other option is manual filtering, which is what we will try out now.

For this kind of filtering, JEXL (Java EXpression Language) is used. JEXL is a software library that is used by GATK. A description can be found here:

http://gatkforums.broadinstitute.org/gatk/discussion/1255/using-jexl-to-apply-hard-filters-or-select-variants-based-on-annotation-values

And you can find a how-to page here:

https://www.broadinstitute.org/gatk/guide/article?id=2806

Now, for our hard-filtering, we start by extracting only the SNPs from our resulting .vcf file.

```
java -Xmx4g -jar programs/GenomeAnalysisTK.jar
-T SelectVariants \
-R data/mm10s.fa \
-V MMR_664_control.raw.snps.indels.vcf \
-selectType SNP
-o MMR_664_control.raw.snps.vcf
```

```
java -Xmx4g -jar programs/GenomeAnalysisTK.jar
-T SelectVariants \
-R data/mm10s.fa \
-V MMR_1370_subject.raw.snps.indels.vcf \
-selectType SNP
-o MMR_1370_subject.raw.snps.vcf
```

The next step is to apply filters to these files. We will run GATK VariantFiltration to accomplish this. This program will mark the filtered SNPs as filtered, and give the name of the filter applied.

```
java -Xmx4g -jar programs/GenomeAnalysisTK.jar
-T VariantFiltration \
-R data/mm10s.fa \
-V MMR_664_control.raw.snps.vcf \
--filterExpression « QD<2 ||FS>60 ||MQ <40 || MQRankSum<-12.5 || ReadPosRankSum < -8  « \
--filterName « mouseSNPfilter« \
-o MMR_664_control.raw.snps_filtered.vcf
```

```
java -Xmx4g -jar programs/GenomeAnalysisTK.jar
-T VariantFiltration \
-R data/mm10s.fa \
-V MMR_1370_subject.raw.snps.vcf \
--filterExpression « QD<2 ||FS>60 ||MQ <40 || MQRankSum<-12.5 || ReadPosRankSum < -8 « \
--filterName « mouseSNPfilter« \
-o MMR_1370_subject.raw.snps_filtered.vcf
```

7 Compare the subject to the control

Now we have the final results for both our subject and our control. However, what we really want to look at, are the SNPs that are there in the subject, but not in the control. Make a python scripts that does this with an option to parallelize. Don't hesitate to ask for help along the way!