Connecting to the cluster and getting started

In this workshop all practical exercises will be conducted using the **levrek1** super-computer clusters at the TUBITAK ULAKBIM High Performance and Grid Computing Center. To access the **levrek1** cluster we will use our **terminal** to remotely login to the ULAKBIM servers. We will do this by using our loginname: **egitim** and password: **q1w2e3r4**. To to this first open up your terminal and then punch in the following command:

```
iso@ISOs-MacBook-Pro~$ ssh egitim@levrek1.ulakbim.gov.tr
```

when we do this the server will prompt us for our password, which we will enter as given above.

```
iso@ISOs-MacBook-Pro~$ ssh egitim@levrek1.ulakbim.gov.tr
password:
```

While typing in your password notice that nothing will appear on the screen so input you password carefully and don't panic!!

Congratulations you are now connected to the **levrek1** server at ULAKBIM and the first thing you will want to do is create a directory (folder) where you will store all your data and conduct your analysis. Creating a directory is done with the **mkdir** (short for make directory) command as shown below. You can name you directory anything you want but the best option is to go with your name. Below I have created one for myself.

```
mkdir iksaglam
```

Now that we have created a directory lets check if it is there. To list all items, files, folders or sub directories in a directory we

simply list them by using the **ls** (short for list) command.

```
ls iksaglam
```

now let us move in to our newly created directory using the **cd** (short for change directory) command

```
cd iksaglam
```

we can always go back to our previous directory by typing **cd ..**

De-multiplexing raw read files and basic shell scripting

Most NGS run results will be supplied to you as raw sequence reads with information from all individuals dumped into one big (tow big if it is paired end library) file. So the first thing we would want to do is to split reads from each individual into their own files and at the same time get read of such stuff as adapters, barcodes or cut-sites.

An example of a raw sequence files can be found in the Charr directory, which contains paired-end reads obtained by RAD-sequencing. Now lets take a look at these files without moving out of our current directory. We can very simply do this by using the **ls** command and supplying the command with the path to the files:

```
ls ../Charr/*
```

above is what is called a **relative path**. We can also use an **absolute path**:

```
ls /truba/home/egitim/Charr/* or ls ~/Charr/*
```

In the above command **~** is short for your home directory in this case **truba/home/**

We have also made use of our very first **wildcard "*".** Wildcards are **a set of characters that can be used as a substitute for all members of some class of characters.** In the above command ***** stands for **all**, so we have just given the command to list all files in a given directory. Wildcards are very useful for searching within directories or filtering and manipulating files. Below is a list of basic wildcards and their uses:

?(question mark): this can represent any *single* character. If you specified something at the command line like "hd?" GNU/Linux would look for hda, hdb, hdc and every other letter/number between a-z, 0-9.

***(asterisk):** this can represent any number of characters (including zero, in other words, zero or more characters). If you specified a "cd*" it would use "cda", "cdrom", "cdrecord" and *anything* that starts with "cd" also including "cd" itself. "m*l" could by mill, mull, ml, and anything that starts with an m and ends with an l.

[ ] (square brackets): specifies a range. If you did m[a,o,u]m it can become: mam, mum, mom if you did: m[a-d]m it can become anything that starts and ends with m and has any character a to d inbetween. For example, these would work: mam, mbm, mcm, mdm. This kind of wildcard specifies an "or" relationship (you only need one to match).

{ } (curly brackets): terms are separated by commas and each term must be the name of something or a wildcard. This wildcard will copy anything that matches either wildcard(s), or exact name(s) (an "or" relationship, one or the other).

[!]: This construct will match any character, as long as it is not listed between the [and]. This is a logical NOT. For example *rm myfile[!9]* will remove all myfiles* (ie. myfiles1, myfiles2 etc) but won't remove a file with the number 9 anywhere within it's name.

\ (backslash): is used as an "escape" character, i.e. to protect a subsequent special character. Thus, "\\" searches for a backslash. Note you may need to use quotation marks and backslash(es).

Now lets get back to our topic. When we list all files in the Charr directory we can see 5 files.

```
align_pe_reads.sh
BarcodeSplitListBestRadPairedEnd.pl
Charr_reduced.metadata
count_no_align.sh
run_BestRadSplit8.sh
SOMM082_S1_L001_R1_reduced.fastq
SOMM082_S1_L001_R2_reduced.fastq
```

Two of the files ending with ".fastq" are our raw read files. Notice that because this is a paired-end library we have two raw read files, one for the forward reads (R1) and one for the reversed reads (R2). Now using our knowledge of wildcards lets do something fancy and list only the fastq files. Using the * wildcard we can filter the ls command so it will only give us a list of fastq files present in the Charr directory. To do this we simply go:

```
ls ~/Charr/*.fastq
```

Now lets take a look inside these files. We can do this using the **"less"** command. **less** is a terminal pager program used to view (but not change) the contents of a text file. Therefore it is very useful when we want to view the contents of a file but make sure that we don't accidentally change its contents.

```
less ~/Charr/SOMM082_S1_L001_R1_reduced.fastq
```

Based on what we learned in the morning session can you understand the contents of this file. Specifically see if you can isolate for yourself the following items.

```
Which lines are the headers?
Which lines are the sequences?
Which lines are the quality scores?
What is the name of the sequencer?
Which flow cell lane was the sequencing conducted in?
Where is the pair member information?
```

Now open up the second pair of our raw read files.

```
less ~/Charr/SOMM082_S1_L001_R2_reduced.fastq
```

What is different between the two files?
How can we match sequences from this file with their pair in the first file?

Now let us separate reads from each individual into their own file using our barcode information. Lucky we have information on which barcode goes with which individual in our **metadata** file. A metadata file is a master file which keeps all sorts of useful information about our libraries. Lets take a look

```
less ~/Charr/Charr_reduced.metadata
```

What type of information is stored here?
Where are the barcodes?

Using the information in our metadata file we can now separate individual reads into their own file via a custom perl script "BarcodeSplitListBestRadPairedEnd.pl". Perl is a very useful and flexible scripting language that is commonly used in bioinformatics for custom tasks. Another very popular language is "Python" and it is of utmost importance that those interested in bio-informatics learn

at least one of these scripting languages.

The “BarcodeSplitListBestRadPairedEnd.pl” is a very simple script that looks inside both raw read files, searches for sequences starting with any of the given barcodes and if it finds a match writes header, sequence and quality score information of those reads to a new file. To have an idea about how the script works lets take a quick look inside

```
less ~/Charr/ BarcodeSplitListBestRadPairedEnd.pl
```

Let's break down the first few lines which contain a lot of useful information:

```
#!/usr/bin/perl

if ($#ARGV == 3) {
        $file1 = $ARGV[0];
        $file2 = $ARGV[1];
        $barcode = $ARGV[2];
        $prefix = $ARGV[3];
} else {
        die;
}
```

The first line **(#!/usr/bin/perl)** is what is called a **shebang** line. This line tells our computer the language the text is written in so it can properly interpret the commands.

The second line starts an **if** statement, which defines a set of logical conditions necessary for the script to run. Here the **if** statement tells us that the script will search for **4 arguments** and if it finds those will carry on executing if not (**else**) it will die.

From these simple lines we can understand that for this script to run we need to give it **4 arguments** or "**infiles**". Also note that perl counts arguments starting from **0** and not **1**. This is a common practice adopted by nearly all scripting and computing languages.

Now before we run our script lets copy all files to our own directory to

Now lets give our script the arguments it needs to run. Arguments are any entries that follow a script and are separated by a space. So to run our barcode-split script we need to punch in the following command:

```
BarcodeSplitListBestRadPairedEnd.pl
~/Charr/SOMM082_S1_L001_R2_reduced.fastq
~/Charr/SOMM082_S1_L001_R2_reduced.fastq
GGAGATCGCATGCAGG,GGAATGTTGCTGCAGG,GGAACGCTTATGCAGG,GGAGTCACTATGCAGG,G
GGCGAGTAATGCAGG,GGTCTTCACATGCAGG,GGCTGAGCCATGCAGG,GGTTCACGCATGCAGG,GG
ACGCTCGATGCAGG,GGACGACAAGTGCAGG, Charr_reduced
```

Notice that the script does not recognize different barcodes as separate arguments because they are separated by a comma and not space!

After running the above command lets list the contents of our directory to see if the script worked. If everything ran smoothly we should see 20 additional files which correspond to our barcode split individuals. Each file is named after the barcode associated with those reads and pair read information (RA, forward read; RB, reverse read).

One way we can quickly check if things went in order is to count the

number of reads in each file. If the script worked correctly each corresponding RA and RB file should have identical number of reads. We can get this information by using the **wc** (word count command). In this instance we want to count the number of lines, so we run the following command:

```
wc -l *R[AB]*.fastq | less
```

This command will output on to the screen the number of lines in each file. Note here that we used two different commands in sequence: we first used the **wc -l** command to count the number of lines in each fastq file (notice the use of wild cards to count only the files we want) and then we transferred the output of this command to **less**. This kind of action is called **piping** and is achieved through using the **pipe "|"** symbol.

Now that we have separated reads belonging to each individual the next thing we would want to do is to rename the files to something more meaningful. For example instead of using individual barcodes as labels we can use our original sample IDs. Thanks to our metadata file we know exactly which barcode corresponds to which sample ID. For example the barcode **GGAGATCGCATGCAGG** corresponds to the sample Charr_003. Therefore lets rename **Charr_reduced_GGAGATCGCATGCAGG_RA.fastq** and **Charr_reduced_GGAGATCGCATGCAGG_RB.fastq** to **Charr_003_R1.fastq** and **Charr_003_R2.fastq**. We can easily achieve this using the **mv** (short hand for move) command:

```
mv Charr_reduced_GGAGATCGCATGCAGG_RA.fastq Charr_003_R1.fastq
mv Charr_reduced_GGAGATCGCATGCAGG_RB.fastq Charr_003_R2.fastq
```

We can repeat this for the remaining nine individuals until all files have been renamed. Although this might be trivial for 10 individual imagine if we had 50, 100 or 1000? Since in most biological studies

we usually deal with hundreds of samples it would be a pretty practical and time saving if we could automate these processes.

Luckily we can do this by what is called **shell scripting**. **Shell** is a command language interpreter that executes commands from the standard input device (keyboard) or from a file. **Shell scripts** can be considered highly simple computer programs which are designed to be run by the **Unix shell**, the command line interpreter used in **Linux** or **Mac** operating systems.

Therefore we can use simple shell scripts to run file manipulations, program executions, and for printing our results. The best part of shell scripting is that it allows us to automate our workflow and combine several different programs by taking the output of one program and using it as the input of another. This is where the common phrase **building a pipeline** or **using a pipeline** comes from.

Below is a simple shell script (**run_BestRadSplit8.sh**) that automatically performs what we have done so far (i.e de-multiplexing and renaming).

```
#!/bin/bash

F1=$1
F2=$2
index=$3
pop=$4
F3=$(awk '{print $4}' $pop | sed 1d | tr '\n' ',' | awk '{print $1}')
n=$(wc -l ${pop} | awk '{print $1}')



BarcodeSplitListBestRadPairedEnd.pl $F1 $F2 $F3 $index

x=2
while [ $x -le ${n} ]
do
```

```
        string="sed -n ${x}p ${pop}"
        str=$($string)

        var=$(echo $str | awk -F"\t" '{print $1, $2, $3, $4}')
        set -- $var
        c1=$1   ### label ###
        c2=$2   ### form ###
        c3=$3   ### sex  ###
        c4=$4   ### Barcode ###

                mv ${index}_RA_${c4}.fastq ${c1}_R1.fastq
                mv ${index}_RB_${c4}.fastq ${c1}_R2.fastq

        x=$(( $x + 1 ))

done
```

To run this shell scrip we need to supply it with four arguments: The forward and reverse raw reads, the name of our index (**Charr_reduced**) and a metadata file (**Charr_reduced.metadata**). Lets run this shell script and see what happens.

```
sbatch -J iks run_BestRadSplit8.sh SOMM082_S1_L001_R1_reduced.fastq SOMM082_S1_L001_R2_reduced.fastq Charr_reduced Charr_reduced.metadata
```

If all went smoothly we should have 20 files containing forward and reverse reads of each individual, designated by our original sample IDs.

Now that we have de-multiplexed our raw reads and split them up into individuals its time to align them to the reference.

Aligning multiple individuals to a reference

In order to call SNPs, genotypes or start analyzing our data in general we need to align our reads to a reference genome. To do this we will be using the short-read aligner BWA and later SAMtools to filter and manipulate aligned files.

Lets start by aligning a single individual (**Charr_003**) to our reference genome. As a reference we will be using the **Rainbow trout** (*Oncorhynchus mykiss*) genome which is a close ancestor to our species the **White-Spotted Charr** (*Salvelinus leucomaenis*). The reference genome is located in the director **~/references/**. Lets take a look if our reference is there:

```
ls ~/references/*.fa
```

Now that we have located our reference we have to first index it so that we can start aligning individuals to this genome. To index our reference execute the following command:

```
bwa index -a bwtsw ~/references/Oncorhynchus_mykiss_chr.fa
```

After indexing we are now ready to start aligning our individual to the reference. To do this execute the following command:

```
bwa mem ~/references/Oncorhynchus_mykiss_chr.fa Charr_003_R1.fastq
Charr_003_R2.fastq > Charr_003.sam
```

Lets look inside the newly created alignment (i.e. SAM) file. Try to identify some of the following

```
Where are the sam flags and cigar strings?
Can you interpret some of the sam flags and cigar strings?
Can you identify properly paired reads?
Can you separate between uniquely mapped reads and those which have
mapped to multiple positions?
```

Now to save space let us sort and convert our alignment file (SAM) into the binary format BAM. We will do this using the very useful program SAMtools by executing the following command:

```
samtools view -bS Charr_003.sam | samtools sort -  Charr_003_sorted
```

This command should create a binary and sorted version (**Charr_003_sorted.bam**) of **Charr_003.sam.** Although we cannot view binary files directly we can do by using the samtools command **view**

```
samtools view Charr_003_sorted.bam | less -S
```

Now before we start using the alignment file for designating SNPs or genotyping it is always a good idea to remove PCR clones which can severely bias designating a site as homozygous or heterozygous. Luckily for us we can do this in samtools but before filtering samtools requires all reads to be properly paired. Therefore we have to first make sure that our alignments consist of only paired reads.

We can filter our alignment for proper pairs using the following command. Notice while filtering we use the information contained in the samflag binary code.

```
samtools view -b -f 0x2 Charr_003_sorted.bam >
Charr_003_sorted_proper.bam
```

Once our alignment only consists of proper pairs we can go ahead and remove duplicates

```
samtools rmdup Charr_003_sorted_proper.bam
Charr_003_sorted_proper_rmdup.bam
```

Now that we have completed all filtering steps we can go ahead and index our alignment file.

```
samtools index Charr_003_sorted_proper_rmdup.bam
```

We now have our final bam file all indexed and ready for analysis. However we still have to complete all of the above steps for all the remaining samples in our study. Since we only have 9 other samples to process this might not seem like much but it is always best to automate such procedures because any real study is bound to contain many more samples.

We can easily do this by putting all above commands into a shell script (**align_pe_reads.sh**) and looping over all samples:

```
#!/bin/bash -l

pop=$1
n=$(wc -l ${pop} | awk '{print $1}')
ref=$2

#       bwa index -a bwtsw ${ref}

x=2
while [ $x -le ${n} ]
do

        string="sed -n ${x}p ${pop}"
        str=$($string)
```

```
        var=$(echo $str | awk -F"\t" '{print $1, $2, $3, $4}')
        set -- $var
        c1=$1   ### Indv label ###
        c2=$2   ### mig form ###
        c3=$3   ### sex  ###
        c4=$4   ### barcode  ###



                echo "#!/bin/bash" > ${c1}.sh
                echo "" >> ${c1}.sh
                echo "bwa mem ${ref} ${c1}_R1.fastq ${c1}_R2.fastq > ${c1}.sam" >>
${c1}.sh
                echo "samtools view -bS ${c1}.sam | samtools sort - ${c1}_sorted"
>> ${c1}.sh
                echo "samtools view -b -f 0x2 ${c1}_sorted.bam > $
{c1}_sorted_proper.bam" >> ${c1}.sh
                echo "samtools rmdup ${c1}_sorted_proper.bam $
{c1}_sorted_proper_flt.bam" >> ${c1}.sh
                echo "samtools index ${c1}_sorted_proper_flt.bam" >> ${c1}.sh

                sbatch --mem=32G -c 1 ${c1}.sh

        x=$(( $x + 1 ))

done
```

Now lets run this shell script to quickly set up all our samples for analysis.

```
sbatch -J iks --mem=8G align_pe_reads.sh Charr_reduced.metadata
~/references/Oncorhynchus_mykiss_chr.fa
```

If all went well we should now have 10 alignment files along with their indexes.

Now there is one last step we should take before jumping into downstream analysis and that is checking the quality of our data.

Samtools has an excellent command for giving us alignment statistic called flagstat. Lets run this command for one of our alignments.

```
samtools flagstat Charr_022_sorted.bam
```

Now lets run the same statistics for our filtered bam files and observe how statistics change

```
samtools flagstat Charr_022_sorted_proper.bam
samtools flagstat Charr_022_sorted_proper_rmdup.bam
```

As with all previous analysis we can of course automate all of this by creating a shell script (**count_no_align.sh**).

```
#!/bin/bash -l

pop=$1
n=$(wc -l ${pop}.metadata | awk '{print $1}')

x=2
while [ $x -le $n ]
do

        string="sed -n ${x}p ${pop}.metadata"
        str=$($string)

        var=$(echo $str | awk -F"\t" '{print $1, $2, $3, $4}')
        set -- $var
        c1=$1   ### Indv label ###
        c2=$2   ### form  ###
        c3=$3   ### sex ###
        c4=$4   ### barcode ###

                samtools view -c -f 1 -F 12 ${c1}_sorted.bam >> no_align.txt
                samtools view -c -f 1 -F 12 ${c1}_sorted_proper.bam >> no_prop.txt
                samtools view -c -f 1 -F 12 ${c1}_sorted_proper_rmdup.bam >>
no_rmdup_align.txt
```

```
                wc -l ${c1}_R1.fastq >> no_R1_reads.txt

        x=$(( $x + 1 ))

done

        awk '{c=$1/2; print c}' no_R1_reads.txt > no_reads.txt
        awk '{print $1}' ${pop}.metadata | sed 1d > names.txt
        paste -d" " names.txt no_reads.txt no_align.txt no_prop.txt
no_rmdup_align.txt > ${pop}.txt
        sed -i '1iIndv no_reads no_align no_prop no_rmdup_align' ${pop}.txt
        awk -v OFS="\t" '$1=$1' ${pop}.txt > ${pop}.reads
        rm *.txt
```

This shell script uses the samflag binary codes to count the number of alignments in all our bam files and then we run some simple awk and sed commands to create a nice table that summarizes our results.

Now lets run this script and see what happens

```
sh count_no_align.sh Charr_reduced
```

If all ran correctly we should have an output file named Charr_reduced.reads. This file give us basic statistics starting from raw reads to filtered alignments. It is always a good idea to have these types of files to evaluate alignment statistics.