#### Parameters and Command Line

NextFlow was downloaded and installed on the Cardiff University/Supercomputing Wales Hawk supercomputer scratch space along with other dependencies. Understanding of procedure and concepts was made possible using the official NextFlow documentation, outlining information such as processes, channels, execution and Input-Output defining (Regulation 2019). Processes were executed via Slurm, with pre-built Slurm scripts integrated into the NextFlow script for each process. Along with justifications for each of the flags and parameters used, the following design was implemented for the NextFlow analysis script:

#### **Step 0 - Process buildIndex (figure 6)**

##### Input

file genome from genome\_file

The reference genome is input to the step via creation of the `genome\_file` `fromPath` channel (see figure 6).

##### Output

file "${genome}" into genome\_index

##### Command Line

bwa index ${genome}

BWA index was used under default conditions, with no output database defined because of the `publishDir` parameter used within the NextFlow process (figure 6).

genome\_file=Channel.fromPath( "/scratch/c.c1866917/NextFlow/resources/GRCm38\_sm.fa" )

gen\_index="/scratch/c.c1866917/NextFlow/resources/GRCm38\_sm.fa"

process buildIndex {

executor "slurm"

cpus 8

memory "64 GB"

clusterOptions="-A scw1448"

publishDir "${baseDir}/../resources", mode: "copy"

input:

file genome from genome\_file

output:

file "${genome}" into genome\_index

script:

"""

module purge

module load bwa

bwa index ${genome}

"""

}

**Figure 6:** The buildIndex process in the NextFlow script. The input file `genome` is defined from the `genome\_file` channel, created via the `Channel.fromPath` command which takes a file from a directory path. The output file `${genome}` is then sent into the `genome\_index` channel for later use. Pre-existing modules are purged – to avoid package/dependencies conflict - and then required `bwa` module is loaded to the Slurm executor. The executor is defined as `”slurm”`, the number of CPUs set as 8, memory allowance set to 64GB and project code (`clusterOptions`) set as the MSc Bioinformatics code to access the Hawk supercomputer (-A scw1448). The publish directory, set with mode `”copy”` so that the output is sent there and not just kept in the work directory, is set as the `resources` directory (`${baseDir}` is defined as the current directory that the script is launched from at the beginning of the script).

#### **Step 1 - Process fastqcOriginal (figure 7)**

##### Input

set pair\_id, file(fwd\_reads), file(rev\_reads) from read\_pairs

The variable `pair\_id` is set from the `read\_pairs` channel, which is created via a flattened `FilePairs` channel which takes the paired end reads (`params.reads`) and makes an input tuple (size 2) so they can be easily accessed seperately.

params.reads="/scratch/c.c1866917/NextFlow/input/ATACseq/\*\_{1,2}.fastq.gz"

Channel

.fromFilePairs( params.reads, size: 2, flat: true)

.ifEmpty { error "Cannot find any reads matching: ${params.reads}" }

.into { read\_pairs; read\_pairs2 }

##### Output

file "\*\_fastqc.{zip,html}" into fastqc\_original

##### Command Line

fastqc -f fastq -q ${fwd\_reads}

fastqc -f fastq -q ${rev\_reads}

FastQC is run for both forward and reverse reads, under virtually default parameters. The `-f` flag (`--format`) is used to define the input format as `fastq`, the `-q` flag (`--quiet`) used to supress all standard out progress messages and only present standard error. The `${fwd\_reads}` and `${rev\_reads}` are then used as placeholders for the inputs (forward and reverse read files).

params.reads="/scratch/c.c1866917/NextFlow/input/ATACseq/\*\_{1,2}.fastq.gz"

Channel

.fromFilePairs( params.reads, size: 2, flat: true)

.ifEmpty { error "Cannot find any reads matching: ${params.reads}" }

.into { read\_pairs; read\_pairs2 }

process fastqcOriginal {

executor "slurm"

cpus 8

memory "64 GB"

clusterOptions="-A scw1448"

publishDir "$baseDir/../output/FastQC/aa047/Original", mode: "copy"

input:

set pair\_id, file(fwd\_reads), file(rev\_reads) from read\_pairs

output:

file "\*\_fastqc.{zip,html}" into fastqc\_original

script:

"""

module purge

module load java

module load FastQC

fastqc -f fastq -q ${fwd\_reads}

fastqc -f fastq -q ${rev\_reads}

"""

}

**Figure 7:** The fastqcOriginal process in the NextFlow script. Pre-existing modules are purged – to avoid package/dependencies conflict - and then required `java` language module - as the FastQC package runs in Java - is loaded to the Slurm executor along with `FastQC`. Other parameters are set as per figure 6.

#### **Step 2 - Process trimmomatic (figure 8)**

##### Input

set pair\_id, file(fwd\_reads), file(rev\_reads) from read\_pairs2

The variable `pair\_id` is set from this channel.

##### Output

set pair\_id, "${fwd\_reads.getSimpleName()}\_trimmed.fastq.gz", "${rev\_reads.getSimpleName()}\_trimmed.fastq.gz" into trimmed\_fastqc, bwa\_align

file "\*\_unpaired.fastq.gz" into trimmed\_unpaired

To maintain the pairwise nature of the read files, the outputs of the trimmomatic process are separated, with the pair\_id variable set, and are defined as their base file name (via the `.getSimpleName()` parameter) with the corresponding `\_trimmed` suffix. All unpaired reads (`”\*\_unpaired.fastq.gz”`) are sent to the unused `trimmed\_unpaired` channel.

##### Command Line

java -jar ${TRIMMOMATIC} PE -phred33 \

${fwd\_reads} ${rev\_reads} \

${fwd\_reads.getSimpleName()}\_trimmed.fastq.gz ${fwd\_reads.getSimpleName()}\_unpaired.fastq.gz \

${rev\_reads.getSimpleName()}\_trimmed.fastq.gz ${rev\_reads.getSimpleName()}\_unpaired.fastq.gz \

ILLUMINACLIP:${contam}:2:30:10 HEADCROP:16 LEADING:3 TRAILING:3 SLIDINGWINDOW:4:15 MINLEN:36 CROP:55

As a Java run tool, Trimmomatic requires the `java -jar` prefix to the command; indicating the Java language is being used (`java`) and that a jar executable file is being run (`-jar`). The Trimmomatic executable, defined as a variable which points to the location of the file (see figure 8), is then specified in paired-end mode (`PE`). The fastq files that are referred to in the command have their quality scores in phred33 format, important for the threshold arguments within the executable (`-phred33`). The placeholders for the forward and reverse reads inputs are then given (`${fwd\_reads} `${rev\_reads}`), followed by the names of the output files for the trimmed reads (`\_trimmed.fastq.gz` suffix) and the unpaired reads (`\_unpaired.fastq.gz`); note how the `.getSimpleName()` parameter is used for the variables, so that only the base name of the input files is used to keep the output easily interpretable.

The parameters for the read trimming are then given - which have been carefully decided based on the results of the original FastQC reports - first defining the reads as Illumina reads (`ILLUMINACLIP`) and a variable that gives the path to the contaminants fasta file (`${contam}`), in which contains the relevant adapter, PCR sequences to be removed from the reads. In order to maintain efficient trimming whilst keeping enough of the reads, the seed mismatch count - the number of mismatches in the putative adapter sequence in the read which can still warrant a full match – is set as 2. The palindrome clip threshold – how accurate the match between the pairs of adapter ligated reads must be for them to be aligned in paired end palindromic method – is set as 30 perfect matches, along with the simple clip threshold – how accurate the matches between the putative and contaminants file adapter sequences must be – is set as 10 perfect matches.

The `HEADCROP` - the number of reads removed at the start of the read (regardless of quality) - is set as 16, with the `LEADING` set as 3 – the minimum phred score (base quality) for which any read will kept from the start of the read; moreover, the `TRAILING` is also set as 3 to remove bases with phred scores lower than this minimum from the end of the read. `SLIDINGWINDOW` is a method of trimming where a ‘window’, a particular sized (in this case 4) sequence of bases within the read, is focused on and moved linearly along the read; trimming the ‘window’ along with any upstream bases if the average base quality of the ‘window’ exceeds the threshold (in this case 15). Finally, the `MINLEN` defines the minimum length that any trimmed read must be to be kept as 36 – any reads shorter than this are dropped – and the `CROP` defines the maximal length of reads, for which bases at the end of the read (regardless of quality) will be removed to accommodate, as 55 bases.

params.reads="/scratch/c.c1866917/NextFlow/input/ATACseq/\*\_{1,2}.fastq.gz"

TRIMMOMATIC="/apps/genomics/trimmomatic/0.39/trimmomatic-0.39.jar"

contam=file("/scratch/c.c1866917/NextFlow/resources/contaminants.fa")

Channel

.fromFilePairs( params.reads, size: 2, flat: true)

.ifEmpty { error "Cannot find any reads matching: ${params.reads}" }

.into { read\_pairs; read\_pairs2 }

process trimmomatic {

executor "slurm"

cpus 8

errorStrategy {task.attempt < 4 ? 'retry' : 'ignore'}

memory { 35.GB \* task.attempt }

runTime="24:00:00"

clusterOptions="-A scw1448"

publishDir "${baseDir}/../output/Trimmomatic/aa047", mode: "copy"

input:

set pair\_id, file(fwd\_reads), file(rev\_reads) from read\_pairs2

output:

set pair\_id, "${fwd\_reads.getSimpleName()}\_trimmed.fastq.gz", "${rev\_reads.getSimpleName()}\_trimmed.fastq.gz" into trimmed\_fastqc, bwa\_align

file "\*\_unpaired.fastq.gz" into trimmed\_unpaired

script:

"""

module purge

module load java

module load trimmomatic

java -jar ${TRIMMOMATIC} PE -phred33 \

${fwd\_reads} ${rev\_reads} \

${fwd\_reads.getSimpleName()}\_trimmed.fastq.gz ${fwd\_reads.getSimpleName()}\_unpaired.fastq.gz \

${rev\_reads.getSimpleName()}\_trimmed.fastq.gz ${rev\_reads.getSimpleName()}\_unpaired.fastq.gz \

ILLUMINACLIP:${contam}:2:30:10 HEADCROP:16 LEADING:3 TRAILING:3 SLIDINGWINDOW:4:15 MINLEN:36 CROP:55

"""

}

**Figure 8:** The trimmomatic process in the NextFlow script. The number of CPUs, the executor, run time and project code are kept the same, however, an error strategy is implemented to increase the amount of memory when necessary. This error strategy will cause the process to retry at least 4 times, causing the `task\_attempt` variable to increase by 1 for each retry, which in turn increases the maximum avalailable memory (`35GB \* task\_attempt`). The input read pairs are maintained, as per the fastqcOriginal process. The `TRIMMOMATIC` and `contam` variables are defined as the locations of the Trimmomatic executable file and the contaminants file respectively. Pre-existing modules are purged – to avoid package/dependencies conflict - and then required `java` language module - as the Trimmomatic package runs in Java - is loaded to the Slurm executor along with `Trimmomatic`. Other parameters are set as per figure 6.

#### **Step 3 - Process fastqcTrimmed (figure 9)**

##### Input

set trimmed\_id, file(fwd\_reads), file(rev\_reads) from trimmed\_fastqc

##### Output

file "\*\_fastqc.{zip,html}" into fastqc\_trimmed

##### Command Line

fastqc -f fastq -q ${fwd\_reads}

fastqc -f fastq -q ${rev\_reads}

The command line is run in the same way as the fastqcOriginal process, except the input of the paired end read files comes from the output of the trimmomatic process.

process fastqcTrimmed {

executor "slurm"

cpus 8

memory "64 GB"

clusterOptions="-A scw1448"

publishDir "$baseDir/../output/FastQC/aa047/Trimmed", mode: "copy"

input:

set trimmed\_id, file(fwd\_reads), file(rev\_reads) from trimmed\_fastqc

output:

file "\*\_fastqc.{zip,html}" into fastqc\_trimmed

script:

"""

module purge

module load java

module load FastQC

fastqc -f fastq -q ${fwd\_reads}

fastqc -f fastq -q ${rev\_reads}

"""

}

**Figure 9:** The fastqcTrimmed process in the NextFlow script. The process is kept almost identical to the fastqcOriginal process (figure 7) except for the input paired end read files coming from the trimmomatic process output. The publish directory is also changed to the Trimmed FastQC report directory.

#### **Step 4 - Process bwaAlign (figure 10)**

##### Input

set trimmed\_id, file(fwd\_reads), file(rev\_reads) from bwa\_align

##### Output

file "\*.bam" into alignment\_file

##### Command Line

bwa mem -t 8 ${gen\_index} ${fwd\_reads} ${rev\_reads} | samtools view -Shu - > ${fwd\_reads.getSimpleName()}.bam

As discussed previously, the `mem` algorithm was selected to perform the alignment against the reference genome because of its optimisation for Illumina reads and better performance, against other bwa alignment algorithms. The number of threads to perform this alignment (`-t`) is set as 8 as alignment can be a time-consuming process, so more available cores will allow the alignment to run faster. The `${gen\_index}` variable is set as the location of the previously indexed (process 0 – buildIndex) reference genome (see figure 10). The forward and reverse reads are defined for the alignment step (`${fwd\_reads}` and `${rev\_reads}` respectively). The results of this alignment are then piped (`|`) into the next step to avoid unnecessary intermediate file creation. Samtools view is then used to enable interpretation of the alignment output; where the input is defined as SAM (`-S`), the headers are included in the output (`-h`) and the output set as uncompressed BAM (`-u`). The `-` placeholder is used to locate the piped output of the alignment. The interpreted alignment file is then redirected (`>`) to be stored as the base name of the forward reads with the .bam suffix (`${fwd\_reads.getSimpleName()}.bam`). The heavy use of piping and redirection in this step maintains an efficient command line via avoiding creation of unnecessary intermediate files and keeping the command line austere, keeping disk usage low and minimising system resources respectively.

process bwaAlign {

executor "slurm"

cpus 10

errorStrategy {task.attempt < 5 ? 'retry' : 'ignore'}

memory { 45.GB \* task.attempt }

runTime="24:00:00"

clusterOptions="-A scw1448"

publishDir "${baseDir}/../output/bwa/aa047", mode: "copy"

input:

set trimmed\_id, file(fwd\_reads), file(rev\_reads) from bwa\_align

output:

file "\*.bam" into alignment\_file

script:

"""

module purge

module load bwa

module load samtools

bwa mem -t 8 ${gen\_index} ${fwd\_reads} ${rev\_reads} | samtools view -Shu - > ${fwd\_reads.getSimpleName()}.bam

"""

}

**Figure 10:** The bwaAlign process in the NextFlow script. The run time, executor and project code are kept the same as previous processes. However, a greater number of CPUs (10) and an error strategy with a greater allowance than the trimmomatic process (both in terms of task attempt limit and memory allocation) maintain a fast running, failsafe process. Pre-existing modules are purged – to avoid package/dependencies conflict - and `bwa` and `samtools` modules are loaded for use in the command line.

#### **Step 5 - Process sortAlign (figure 11)**

##### Input

file alignment from alignment\_file

##### Output

file "\*.sorted.bam" into sorted\_file

##### Command Line

samtools sort -m 80G ${alignment} > ${alignment.getSimpleName()}.sorted.bam

The samtools `sort` algorithm is used to sort the alignment by leftmost coordinate. As this sorting can often require a lot of system resources, the amount of memory (`-m`) is defined as 80G. The alignment output from the bwaAlign process is inserted (`${alignment}`) and the output of the sorting is redirected (`>`) to a file where the base file name is taken and added to the “.sorted.bam” suffix to create the sorted .bam file (`${alignment.getSimpleName()}.sorted.bam`).

process sortAlign {

executor "slurm"

cpus 8

errorStrategy {task.attempt < 5 ? 'retry' : 'ignore'}

memory { 45.GB \* task.attempt }

runTime="24:00:00"

clusterOptions="-A scw1448"

publishDir "${baseDir}/../output/bwa/aa047", mode: "copy"

input:

file alignment from alignment\_file

output:

file "\*.sorted.bam" into sorted\_file

script:

"""

module purge

module load samtools

samtools sort -m 80G ${alignment} > ${alignment.getSimpleName()}.sorted.bam

"""

}

**Figure 11:** The sortAlign process in the NextFlow script. The run time, executor and project code are kept the same as previous processes. The potential system resources demand, as mentioned previously, means that the error strategy and increasable memory usage are the same as the bwaAlign process. Pre-existing modules are purged – to avoid package/dependencies conflict – and the `samtools` module is loaded for use in the command line.

#### **Step 6 - Process rmDuplitcates (figure 12)**

##### Input

file sorted from sorted\_file

##### Output

file "\*.rmdup.bam" into index\_file, macs2\_bam

file "\*.rmduplicates.txt" into rmdup\_log

The `rmdup\_log` channel is unused for the rest of the process.

##### Command Line

java -jar ${picard} MarkDuplicates INPUT=${sorted} OUTPUT=${sorted.getSimpleName()}.rmdup.bam METRICS\_FILE=${sorted.getSimpleName()}.rmduplicates.txt REMOVE\_DUPLICATES=true VALIDATION\_STRINGENCY=SILENT

As with the trimmomatic process, the Picard suite is java built so requires the `java -jar` command prefix to run the picard.jar executable – the path to this file is also stored by the `${picard}` variable (see figure 12). The `MarkDuplicates` algorithm is selected, and the `INPUT` defined as the output of the sortAlign process (`${sorted}`). The `OUTPUT` is set as the base name of the alignment file with the .rmdup.bam prefix (`${sorted.getSimpleName()}.rmdup.bam`); the `METRICS\_FILE` (a report of the removed duplicates) is created in the same way (`=${sorted.getSimpleName()}.rmduplicates.txt `). The `REMOVE\_DUPLICATES` flag is set as `true` so that the duplicated reads are deleted and the `VALIDATION\_STRINGENCY` is set as silent, so that if warnings (which are silenced) are given the command will keep running if possible – maintaining the workflow.

picard="/apps/genomics/picard/2.20.2/bin/picard.jar"

process rmDuplicates {

executor "slurm"

cpus 8

memory "64 GB"

clusterOptions="-A scw1448"

publishDir "${baseDir}/../output/bwa/aa047", mode: "copy"

input:

file sorted from sorted\_file

output:

file "\*.rmdup.bam" into index\_file, macs2\_bam

file "\*.rmduplicates.txt" into rmdup\_log

script:

"""

module purge

module load samtools

module load java

module load picard

java -jar ${picard} MarkDuplicates INPUT=${sorted} OUTPUT=${sorted.getSimpleName()}.rmdup.bam METRICS\_FILE=${sorted.getSimpleName()}.rmduplicates.txt REMOVE\_DUPLICATES=true VALIDATION\_STRINGENCY=SILENT

"""

}

**Figure 12:** The rmDuplicates process in the NextFlow script. The path to the Picard .jar executable file is given at the start. Parameters are kept the same as other processes in the workflow. Modules are purged to avoid package and dependency conflict and the modules `samtools`, `picard` and `java` loaded for use in the command line – the latter required because Picard is a Java tool which requires the language module to run.

#### **Step 7 - Process samIndex (figure 13)**

##### Input

file index from index\_file

##### Output

file "\*.bam.bai" into alignment\_index

##### Command Line

samtools index -b ${index} > ${index.baseName}.bam.bai

Very similar to the buildIndex process, instead using samtools instead of bwa for compatibility purposes, the file to be indexed is set as .bam format (`-b`) and entered using the `${index}` placeholder – storing the .bam output of the rmDuplicates process. The output is then redirected to produce a file with the base name of the alignment file followed by the `.bam.bai` suffix, to set as a .bam index (`${index.baseName}.bam.bai `).

process samIndex {

executor "slurm"

cpus 8

memory "64 GB"

clusterOptions="-A scw1448"

publishDir "${baseDir}/../output/bwa/aa047", mode: "copy"

input:

file index from index\_file

output:

file "\*.bam.bai" into alignment\_index

script:

"""

module purge

module load samtools

samtools index -b ${index} > ${index.baseName}.bam.bai

"""

}

**Figure 13:** The samIndex step of the NextFlow pipeline. The components are very similar to the buildIndex process, except with samtools instead of bwa. Parameters are kept the same as other processes in the workflow. Modules are purged to avoid conflicts and the `samtools` module loaded for the command line.

#### **Step 8 - Process macs2Peak (figure 14)**

##### Input

file aligned from macs2\_bam

##### Output

file "\*.macs2\_control\_lambda.bdg" into macs2\_conbdg

file "\*.macs2\_model.r" into macs2\_modelr

file "\*.macs2\_peaks.narrowPeak" into macs2\_peaksnar

file "\*.macs2\_peaks.xls" into macs2\_peaksxls

file "\*.macs2\_summits.bed" into macs2\_sumbed

file "\*.macs2\_treat\_pileup.bdg" into macs2\_treatpilebdg

##### Command Line

macs2 callpeak -t ${aligned} -f AUTO -g ${organism\_id} -n ${aligned.getSimpleName()}.q0.01.macs2 -B -q 0.01 --shift -100 --extsize 200

macs2 callpeak -t ${aligned} -f AUTO -g ${organism\_id} -n ${aligned.getSimpleName()}.q0.05.macs2 -B -q 0.05 --shift -100 --extsize 200

macs2 callpeak -t ${aligned} -f AUTO -g ${organism\_id} -n ${aligned.getSimpleName()}.q0.1.macs2 -B -q 0.1 --shift -100 --extsize 200

The final step in the NextFlow pipeline is the Macs2 peak calling process, carried out 3 times with different q-value cutoffs. As required, the `callpeak` algorithm is defined, the .bam, or treatment file (`-t`), is inserted from the output of the rmDuplicates process (`${aligned}`) and the format (`-f`) set to automatically detect from the input (`AUTO`) to make the pipeline more generic. The genome (`-g`) is defined as *Mus musculus* (`“mm”`, mouse) through a pre-defined variable (figure 14) to also maintain genericity. The name of the output file is set as the base name of the alignment, with a prefix that allows differentiation between each of the steps (e.g. `${aligned.getSimpleName()}.q0.1.macs2`). A bedgraph (`-B`) containing the fragment pileup for report statistic purposes is enabled, the q-value (`-q`) is set differently between command lines and the shift and extension size (`--shift`,`--extsize`) set as 100 and 200 respectively in order to find enriched cutting sites for ATACseq datasets.

This step is different in the ChIPseq dataset NextFlow pipeline, where the control (`-c`) alignment must be set, as well as defining that the shifting model should be bypassed (`--nomodel`) as it is ChIPseq data.

macs2 callpeak -t ${ktreat} -c ${kcontrol} -f AUTO -g ${organism\_id} -n ko.q0.01.macs2 -B -q 0.01 --nomodel --shift -100 --extsize 200

macs2 callpeak -t ${ktreat} -c ${kcontrol} -f AUTO -g ${organism\_id} -n ko.q0.05.macs2 -B -q 0.05 --nomodel --shift -100 --extsize 200

macs2 callpeak -t ${ktreat} -c ${kcontrol} -f AUTO -g ${organism\_id} -n ko.q0.1.macs2 -B -q 0.1 --nomodel --shift -100 --extsize 200

macs2 callpeak -t ${wtreat} -c ${wcontrol} -f AUTO -g ${organism\_id} -n wt.q0.01.macs2 -B -q 0.01 --nomodel --shift -100 --extsize 200

macs2 callpeak -t ${wtreat} -c ${wcontrol} -f AUTO -g ${organism\_id} -n wt.q0.05.macs2 -B -q 0.05 --nomodel --shift -100 --extsize 200

macs2 callpeak -t ${wtreat} -c ${wcontrol} -f AUTO -g ${organism\_id} -n wt.q0.1.macs2 -B -q 0.1 --nomodel --shift -100 --extsize 200

process macs2Peak {

executor "slurm"

cpus 8

memory "64GB"

clusterOptions="-A scw1448"

publishDir "${baseDir}/../output/macs2/aa047", mode: "copy"

input:

file aligned from macs2\_bam

output:

file "\*.macs2\_control\_lambda.bdg" into macs2\_conbdg

file "\*.macs2\_model.r" into macs2\_modelr

file "\*.macs2\_peaks.narrowPeak" into macs2\_peaksnar

file "\*.macs2\_peaks.xls" into macs2\_peaksxls

file "\*.macs2\_summits.bed" into macs2\_sumbed

file "\*.macs2\_treat\_pileup.bdg" into macs2\_treatpilebdg

script:

organism\_id="mm"

"""

module purge

module load macs2/2.1.2

macs2 callpeak -t ${aligned} -f AUTO -g ${organism\_id} -n ${aligned.getSimpleName()}.q0.01.macs2 -B -q 0.01 --shift -100 --extsize 200

macs2 callpeak -t ${aligned} -f AUTO -g ${organism\_id} -n ${aligned.getSimpleName()}.q0.05.macs2 -B -q 0.05 --shift -100 --extsize 200

macs2 callpeak -t ${aligned} -f AUTO -g ${organism\_id} -n ${aligned.getSimpleName()}.q0.1.macs2 -B -q 0.1 --shift -100 --extsize 200

"""

}

**Figure 14:** The final process of the NextFlow pipeline macs2Peak. Parameters are kept the same as other processes, however a variable (`organism\_id`) is defined before the script is executed containing the organism id (`”mm”`) to allow the script to be more generic and customisable. As outlined previously, the command line is different in the ChIPseq NextFlow pipeline. Modules are purge before execution to avoid conflicts in packages and the `macs2` module version 2.1.2 loaded for compatibility purposes.

### DiffBind

The final outputs from the NextFlow analysis pipeline – the sorted .bam alignment file with duplicates removed, the Macs2 peak .xlsx files and the .bam index files – are then output into an R-Markdown (.rmd) script containing various commands from the DiffBind Bioconductor package. DiffBind (Differential Binding analysis) is a package primarily focused on processing ChIPseq data where protein/DNA binding occurs – most importantly in this context, the processing of peak called and aligned sequence read datasets. Important to our analysis, DiffBind excels in identifying sites that are differentially bound between two sample groups - the WT and KO mice in this scenario. In order to achieve this, a number of pre-processing steps are carried out on the input data, including overlapping and merging peak sets and counting sequencing reads overlapping in peaks sets. This allows for the identification of differentially bound sites that are statistically significant, found by measuring differences in densities of reads around certain regions of the alignment which provides evidence for binding affinity (Brown 2012).

The DiffBind package also allows for summary statistics, such as FRiP scores and significance statistics, to be produced, as well as creation of graphic plots to contribute to overall binding analysis. The package was produced in 2012 by Rory Stark and Gordon Brown of the Cancer Research UK team at the Cambridge Research Institute. The version used throughout the R-Studio (version 3.6.1) was 2.12.0, installed via the Bioconductor package (version 3.8). Using a R-markdown script produced with the help of Dr. Robert Andrews at Cardiff University, the following steps were carried out in the analysis:

#### **Step 0 – Setup**

The following R code chunks were run to set up the analysis pipeline.

if (!requireNamespace("BiocManager", quietly = TRUE)) {

install.packages("BiocManager")

}

BiocManager::install("DiffBind", version = "3.8")

The current version of DiffBind is installed, using BiocManager (which is also installed if not already available)

setwd("C:/<ANALYSIS-DIRECTORY>/bin")

The working directory is set as the `bin` directory of the file structure created to hold the DiffBind analysis.

library("DiffBind")

library("dplyr")

The packages `”DiffBind”` and `”dplyr”`, a data wrangling package, are loaded to the current session of R.

#### **Step 1 – Read in the Targets file**

targets <- read.table("../resources/targets.txt", sep="\t", header=T, quote="")

targets$bamReads <- paste("../input/", targets$bamReads, sep="")

targets$Peaks <- paste("../input/", targets$Peaks, sep="")

The variable `targets` is created by reading in (`read.table`) the `targets.txt` file from the resources directory; the file being tab-delimited (`sep=”\t”`), with column headers (`header=T`) and quotes specified as speech marks (`quote=””`). This `targets.txt` file contains a unique sample ID, information about the tissues, factor, condition, treatment, replicate number, name of the .bam reads file, name of the peaks file and the peak caller format. The targets file variable is then changed so that the columns for bamReads and Peaks are correct for the input data files, essentially insuring that the information in the `targets.txt` file is correct.

#### **Step 2 – Creating the required bed files from the MACS2 output**

for (macsFile in targets$Peaks) {

bedFile <- gsub(".xls$", ".bed", macsFile)

macsData <- read.table(macsFile, sep="\t", blank.lines.skip=T, comment.char="#", header=T)

write.table(dplyr::select(macsData, chr, start, end, name), bedFile, sep="\t", row.names=F, col.names=F, quote=F)

}

targets$Peaks <- gsub(".xls$", ".bed", targets$Peaks)

A for loop is created, where each of the MACS2 peak files (contained within the `targets.txt` Peaks column) are taken and converted into .bed format via the `gsub` command, then stored in the `bedFile` variable. The files are then read in and stored in the `macsData` variable, denoting it’s tab-delimited (`sep=”\t”`), blank lines are skipped (`blank.lines.skip=T`), comments are specified with the hash symbol (`comment.char=”#”`) and column headers are present (`header=T`). This is then written to a table (`write.table`), the column names denoted via the dplyr select function (`dplyr::select(macsData, chr, start, end, name)`), data from the `bedFile` variable is used, separated by tabs (`sep=”\t”`), no row or column names (`row.names=F, col.names=F`) and no comments (`quote=F`). The names of these newly created .bed files are then amended to the `Peaks` column in the `targets` variable using the `gsub` command.

#### **Step 3 – Counting the reads under each peak**

sampleData <- dba(sampleSheet=targets)

countData <- dba.count(sampleData)

fripCountData <- dba.count(sampleData, summits=250)

The sample data (`targets`) is read in, signified as a `sampleSheet` via the `DiffBind` package, and stored as the `sampleData` variable. The DiffBind count command (`dba.count`) then counts the reads in the binding sites and stored as the `countData` variable. The FRiP count is then calculated using this same technique, where positions 250bp upstream and downstream of the peak summits (`summits=250`) and stored in the `fripCountData` variable.

#### **Step 4 – Generating contrasts from the differential analysis**

contrastData <- dba.contrast(countData, categories=DBA\_CONDITION, minMembers=3)

The `dba.contrast` command is used to establish the contrasts for analysis, using the count data generated previously. These are grouped by condition (`categories=DBA\_CONDITION`) from the targets file (WT and KO) and defined as a group size of 3 (`minMembers=3`) for each condition. The results of this contrast analysis are stored in the `contrastData` variable.

#### **Step 5 – Perform the differential calculation and export the results**

deResults <- dba.analyze(contrastData)

deOutput <- dba.report(deResults)

write.table(deOutput, file="../output/ko\_vs\_wt.peakDE.txt", sep="\t", \

row.names=F)

Finally, affinity analysis (`dba.analyze`) is carried out on the previously created contrast data and stored in the `deResults` variable. The results of this analysis are then output as a report (`dba.report`), written to a table and stored in the output directory (`file="../output/ko\_vs\_wt.peakDE.txt"`) as a tab-delimited file (`sep=”\t”`) with no row names (`row.names=F`).