Practical exercise 1) RNA-seq analysis using HISAT2/StringTie

Here, we performed the analysis on Bias5 based on the RNA-seq analysis procedure using HISAT2/ StringTie/Ballgown published in Nature Protocol (Pertea, M. et al. Nat. Protoc. 11, 1650, 2016). The paper can be obtained from the following site, so please refer to the paper for details of the procedure. https://pubmed.ncbi.nlm.nih.gov/27560171/

1. Go to ex1 under the course directory and copy the data file chrX_data.tar.gz
   (below, the line starting with $ represents the command to be executed).
   $ cd ~/bias2111/ex1
   $ tar xvfz chrX_data.tar.gz

   Use the following files in the extracted directory
   chrX_data: chrX_data/samples: Input read sequences (FASTQ files)
   chrX_data/indexes: Genome index files used as references
   chrX_data/genes/chrX. gtf: gene annotation

2. ChrX_data/samples contains array data of 12 samples with paired ends.
   At the begining, let's analyze only one sample. Below is a qsub script that performs a series of analyzes of sample ERR188044, mapping with HISAT2 and assembling with StringTie. The part shown in red is the description specific to qsub.
   qsub_hisat2.sh

```
#!/bin/sh
#PBS -l ncpus=4
#PBS -N hisat2
#PBS -q small

cd $PBS_O_WORKDIR

fname=ERR188044_chrX
sample_name=`echo $fname | sed 's/_chrX//'`
input_dir=chrX_data/samples seq_dir=chrX_data/indexes
gene_dir=chrX_data/genes output_dir=output



mkdir -p $output_dir
hisat2 -p $NCPUS --dta -x $seq_dir/chrX_tran ¥
        -1 $input_dir/${fname}_1.fastq.gz -2 $input_dir/${fname}_2.fastq.gz ¥-S $output_dir/${fname}.sam

samtools sort -@ $NCPUS -o $output_dir/${fname}.bam $output_dir/${fname}.sam
stringtie -p $NCPUS -G $gene_dir/chrX.gtf -o $output_dir/${fname}.gtf ¥
  -l ${sample_name} $output_dir/${fname}.bam
```

Note that -l ncpus=4 is specified as a PBS option at the top of the script, so 4 is substituted for the variable $NCPUS in the script. Each of the three $NCPUS specifies the number of CPUs (threads) used by each command, and here each command will consistently use the same 4CPUs to operate.

Submit the job with the command

```
$ qsub qsub_hisat2.sh
```

Check job status. Here, USERNAME should be replaced with your username.

```
$ qstat -u USERNAME
```

Make sure that the Status (second column from the back) is R (running). If this is Q, it is waiting for execution, so you need to wait for a while until execution starts.

The job is finished when nothing is displayed in qstat. Check the newly output file with ls -lt. If it's working correctly, a directory named "output" should be created and the results should be stored in it. In addition, the standard output and standard error output during command execution are stored in files named hisat2.oXXXXX and hisat2.eXXXXX (where XXXXX is the job ID), respectively. Check the contents to see if there are any abnormal terminations.

Also, run the following command to check the execution log and resources used by the job.

$ tracejob XXXXX (where XXXXX is the same job ID as the file name above)

The resource used is displayed in the format resource_used.resource_type=### on the last line that indicates the end status. As resource_type, cput is CPU time, walltime is real time, ncpus is the number of CPUs, and mem is the amount of memory used by the job.

3. Repeat the analysis performed in the previous section for 12 samples while changing the sample name specified by fname. This can be executed by writing a script using the *for* statement, but let's execute it using an array job here. For array jobs, a variable called $PBS_ARRAY_INDEX is embedded in the script, and multiple jobs are generated by sequentially substituting values within the specified range and executed in parallel. To use an array job, treat both the input and output files as file names with the same name appended with a number, such as file.1, file.2, ... and specify them as file.$PBS_ARRAY_INDEX in your script.

   Unfortunately, the input file this time does not have such a name. In this case, giving each input file an alias of the above format with a symbolic link is an easy way to understand, but here let us use a bash array variable that can be written more concisely. An array variable is a data structure containing multiple elements that allows each element to be retrieved by subscript. In bash, array is defined as follows.

   ```
   $ Array=( fileA fileB fileC )
   ```

   Subscripts start at 0, so for example to extract and display the first element

   ```
   $ echo ${Array[0]}
   ```

(Note that the curly braces are required). Using this array variable, the following script is modified from the script in the previous section to use the array job. Corrections are shown in blue.

qsub_hisat2_array.sh

```
#!/bin/sh
#PBS -l ncpus=4
#PBS -N hisat2
#PBS -q small
#PBS -J 0-11

cd $PBS_O_WORKDIR

input_dir=chrX_data/samples
sample_name=`echo $fname | sed 's/_chrX//'`
seq_dir=chrX_data/indexes
gene_dir=chrX_data/genes
output_dir=output

input_files=(`ls $input_dir | grep _1.fastq | sed 's/_1.fastq.gz//'`)
fname=${input_files[$PBS_ARRAY_INDEX]}

mkdir -p $output_dir
hisat2 -p $NCPUS --dta -x $seq_dir/chrX_tran ¥
  -1 $input_dir/${fname}_1.fastq.gz -2 $input_dir/${fname}_2.fastq.gz ¥
  -S $output_dir/${fname}.sam
samtools sort -@ $NCPUS -o $output_dir/${fname}.bam $output_dir/${fname}.sam
stringtie -p $NCPUS -G $gene_dir/chrX.gtf -o $output_dir/${fname}.gtf ¥
  -l ${sample_name} $output_dir/${fname}.bam
```

input_files is an array variable containing the input file names corresponding to the 12 samples. The definition is a little tricky, but you can see what value is entered by executing the following command surrounded by backquotes (backquotes execute the command inside and the result is instructing to paste).

```
$ ls chrX_data/samples | grep _1.fastq | sed 's/_1.fastq.gz//'
```

Now you can see that the array variables contain names corresponding to the 12 samples, such as ERR188044_chrX, ERR188104_chrX. The next line extracts one element specified by $PBS_ARRAY_INDEX from this array and assigns it to fname. Since the subscript of an array variable starts from 0, Note that the value of $PBS_ARRAY_INDEX (-J option) is between 0 and 11.

submit the job

$ qsub qsub_hisat2_array.sh

Check the status with qstat. By default, array jobs are displayed as one job, with Status being B. To show the status of individual jobs (sub-jobs), use the qstat -t option.

$ qstat -u USERNAME -t

When finished, check that the output directory contains the analysis results for the 12 samples. Also, confirm that a file containing standard output and standard error output is created for each subjob.

4. At this point, you have mapped and assembled the reads from each sample to the reference sequences and created a GTF file that records the transcript/gene locations for each sample. After that, the assembly results for each sample are merged to create a single annotation file, which is then used to proceed to the step of creating count data for each transcript/gene from the mapping results of each sample. The script to be executed is created as postproc.sh. Run this with qsub.

(Optional below) Later in this script, we create count data for each sample.
It uses a *for* loop to process

```
for bamfile in $output_dir/ERR*.bam; do
        fname=`basename $bamfile .bam`
        sample_name=`echo $fname | sed 's/_chrX//'`
        stringtie -e -B -p ${NCPUS} -G stringtie_merged.gtf ¥
            -o ${ballgown_out}/${sample_name}/${fname}.gtf $bamfile
done
```

This is inefficient because it is executed sequentially within a single job. Create a script that uses array jobs to process this part. (An example answer is found under the *ans* directory)

5. (Optional: Run R) Up to step 6 of the original paper is now complete, and in the ballgown directory, the transcript/gene count data created for each sample is stored as a GTF file and a tab-delimited table data for the Ballgown program. If you continue after this, it will be an analysis using R. If you use R on your local machine, download the files below the ballgown directory with *scp* command and analyze them in your local environment. You can also run R on Bias5, but in that case, be aware of the following points.
1) You can install missing packages by yourself. In that case, it will be installed in your home directory.
2) The X Windows environment must be set locally to display graphics. For this purpose, specify the ssh -Y option when logging in.

6. (Optional: Database reference) Although in this exercise, the reference sequence and its search index

was prepared in advance, it is necessary to prepare them by yourself in actual analysis. In that case, you can refer to the following database files placed on bias5.

database	path
Illumina iGenomes	/bio/db/igenomes
Ensembl	/bio/ftp/ensemble
NCBI Genomes	/bio/ftp/genomes/refseq (or genbank)

Here, we prepared a script preproc.sh to utilize the above databases and create an index using hisat2-build. If you are interested, check the contents of the script and run it. This script retrieves data from iGenomes by default, but can also be used to retrieve data from Ensembl or NCBI Genomes by modifying it.

7. (Tips: Executing multiple qsub commands by script)

The qsub command exits immediately after submitting the job and returns to the shell, allowing command input. This is fine in most cases, but if you want to execute multiple qsub commands sequentially in your script, it may not work well because the next job may start before the previous job has finished. There are several ways to avoid this, but one easy way is to add "-W block=true" to your qsub options. By this option, the qsub command does not finish until the execution of the job submitted by qsub is completed, so it can be used in the same way as normal command execution. The following script executes the three qsub commands executed so far: preprocessing (preproc.sh), main processing (qsub_hisat2_array.sh), and postprocessing (postproc.sh) as a series of processes.

qsub_all.sh

```
#!/bin/sh
qsub -W block=true preproc.sh
qsub -W block=true qsub_hisat2_array.sh
qsub postproc.sh
```

Practical exercise 2) BLAST execution using array jobs

In the lecture on How to use PBS, there was an example of array job, where the query sequence file was divided and the command was executed for each query file as an array job. Let's do it in practice here. Let's move to ex2 under the course directory. The query sequence is sce_prot.fasta, the whole-gene translation sequence of the budding yeast genome that was used in the lecture.

There are many ways to split a sequence file, but here let's use the *split* subcommand of the sequence file manipulation utility program *SeqKit*.

```
$ seqkit split sce_prot.fasta -p 50
```

This command divides the arrays in the sequence file into the number of files specified by the -p option. The results are stored in a directory called sce_prot.fasta.split. Let's check the contents of this. It is divided into 50 files named sce_prot.part_*nnn*.fasta (*nnn* is a number from 001 to 050).

The script below runs blast as an array job with these files as input.

qsub_blast.sh

```
#!/bin/sh
#PBS -l ncpus=8
#PBS -l mem=12gb
#PBS -N blast
#PBS -q small
#PBS -J 1-50

cd ${PBS_O_WORKDIR}

seqn=`printf "%03d" $PBS_ARRAY_INDEX`
query=sce_prot.fasta.split/sce_prot.part_${seqn}.fasta

db=swissprot
outdir=sce_prot.${db}.blast_out
output=$outdir/sce_prot.part_${seqn}.tab
mkdir -p $outdir

blastp -num_threads $NCPUS -db $db -outfmt 6 -query $query ¥
  -evalue 0.001 -out $output
```

The descriptions specific to array job are in blue and the other descriptions specific to PBS are in red.

Since the option -J is specified, it is executed as an array job consisting of 50 subjobs in which the values 1-50 specified with -J are substituted for the variable $PBS_ARRAY_INDEX. Here, since the input file name is a 3-digit number starting with 0, the *printf* command is used to convert it to this format. This is a command with the same specifications as C function printf, which specifies with "%03d" to convert an integer to a 3-digit integer string starting from 0.

For example, the following command outputs 010 (without line breaks).

```
$ printf "%03d" 10
```

In addition, 8 CPUs and 12 GB of memory are specified as resources to be allocated for each (sub) job. BLAST will expand the database in memory as long as memory is available, so when using a large database, it is necessary to secure the maximum amount of memory. A blast queue is provided for this purpose. But the swissprot database used this time is not very large, so a normal small queue is sufficient.

Let's run this with qsub.

```
$ qsub qsub_blast.sh
```

Check the execution status of each sub-job with qstat -t.

```
$ qstat -u USERNAME -t
```

At this time, not all jobs are in the running state R, and some should be in the waiting state Q. Why? Now, there are 50 jobs in the queue, and 8 CPUs are reserved and executed for each job. Therefore, the total number of CPUs used by these jobs is 400, which exceeds the maximum number of CPUs that can be executed concurrently per user for small queue, 300 (refer to the queue configuration table in Bias). In this case, concurrent number of jobs cannot exceed 300/8 = 37.5 and thus the rest are waiting.

In general, when running a large number of independent jobs, it is often more efficient to increase the number of jobs than to increase the number of CPUs per job, even when using the same number of CPUs. Therefore, the execution efficiency generally decreases if a waiting state occurs due to the excess of the number of CPUs per job. In this case, if the number of CPUs per job is set to 6 or less, the wait state due to such restrictions does not occur.

After the above job has been finished, edit the file and change the number of CPUs to 6 (ncpus=6) and re-execute. However, please note that the number of jobs executed simultaneously may differ depending on how busy the queue is, including the jobs of the other users.