Practical Linux examples: Exercises

1. Login (ssh) to the machine that you are assigned for this workshop (assigned machines: https://cbsu.tc.cornell.edu/ww/machines.aspx?i=87). Prepare working directory, and copy data files into the working directory. (Replace "XXXXX" in the commands with your BioHPC User ID)

```
mkdir /workdir/XXXXX

cd /workdir/XXXXX

cp /shared_data/Linux_workshop2/* ./
```

2. You are given a gzipped gff3 file. Inspect the content in the file.

Linux functions: gunzip -c, head, tail, cut, less

- Inspect the first and last 100 lines of the file, using "head" and "tail" functions to retrieve first and last lines of the file;
- Retrieve line number 1001 to 2000 from the file and write these lines into a new file " mynew.gtf ";
- Inspect the columns 2,3,4 and 8 of lines 3901 to 4000, using "cut" function to specify the columns.
- Compare "cut" with "less -S" function. If you use "less" function, remember to exit "less" by pressing "q".

```
gunzip -c human.gff3.gz | head -n 100

gunzip -c human.gff3.gz | tail -n 100

gunzip -c human.gff3.gz| head -n 2000 | tail -n 1000 > mynew.gtf

gunzip -c human.gff3.gz| head -n 4000 | tail -n 100 | cut -f 2-5,8

gunzip -c human.gff3.gz| head -n 4000 | tail -n 100 | less -S
```

3. Count the number of genes listed in the file.

Linux functions: awk, uniq

- Count the total number of lines in the file using "wc -l" function;
- Count the number of genes list in the file. First, you need to use "awk" to retrieve lines with the 3rd column value equals "gene", then count these lines with "wc -l";

- Count the number for each of the feature categories (genes, exons, rRNA, miRNA, et al.) listed in this GFF3 file. The 3rd column in GFF3 file defines feature categories. First your "cut" out the 3rd column of the file, then use "sort | uniq -c" to count the occurrences of each of the categories. Note: "uniq" requires sorted input stream. Always run "sort" before "uniq -c",

```
gunzip -c human.gff3.gz | wc -l

gunzip -c human.gff3.gz | awk '{if ($3=="gene") print}' |wc -l

gunzip -c human.gff3.gz | cut -f 3 | sort | uniq -c
```

4. Convert the GFF3 file to BED file, using only the lines with the third column equals to "gene". Then add "chr" to the chromosome name, so that chromosome 1 would be "chr1" instead of "1".

Linux functions: awk, sed

- The BED file requires a minimum of 3 columns (chromosome, start position and end position). It is important to note that start and end positions in BED and GFF3 are defined differently. In GFF3 file, the start and end positions are both 1-based (the first nucleotide of a sequence is at position 1). In BED file, the start position is 0-based (the first nucleotide of a sequence is at position 0), and the end position is 1-based. When converting GFF3 to BED, you need to subtract 1 from the start position. In the follow command, the "\" characters are used to split a long command into multiple lines. The expression "BEGIN {OFS = "\t"};'" is to specify that the output stream uses tab as delimiters.
- The "sed" function is probably the fastest way to modify text in a large file and in a stream. In this command, the output from awk is piped into "sed", and "sed" would add "chr" to the chromosome name. The "^" character in "sed" is to specify making changes at the beginning of each line.

```
#The following three lines are in one single command

gunzip -c human.gff3.gz | \

awk 'BEGIN {OFS = "\t"};{if ($3=="gene") print  $1,$4-1,$5}' |\

sed "s/^/chr/"> mygenes.bed
```

5. Get the size distribution of genes.

Linux functions: awk, sort, uniq

- Calculate the size for each of the genes. The size of genes are calculated by subtracting "gene start position" (column4) from "gene end position" (column 5), and adding 1 because GFF3 file use 1-based coordinate system for both start and end positions. (If a BED file is used to calculate the gene size, you do not need to add 1 because the start position is 0-based in the BED file).
- To get the size distribution, you need to add two things to the previous command: 1) Use the int(($5-$4+1)/1000) expression to convert the sequence size unit from "base-pair" to "kilo-base-pair" and convert the number into an integer; 2) The " LC_ALL=C sort -n | uniq -c" combination is used to get the histogram. Note the two new things are added for the "sort" function: "LC_ALL=c" is to forces "sort" to use the default language "US English" to interpret characters. It is important to add this to make sure that "sort" behaves properly; "-n" is to tell "sort" to do numerical sorting. As the output from this command is very long, you can write the output to a file "gene_dist.txt".

```
gunzip -c human.gff3.gz | awk '{if ($3=="gene") print $5-$4+1}'

#The following three lines are in one single command

gunzip -c human.gff3.gz | \

awk '{if ($3=="gene") print int(($5-$4+1)/1000)'} | \

LC_ALL=C sort -n | uniq -c > gene_dist.txt
```

6. Count the number of genes and pseudogenes in sliding windows across the whole chromosome.

Linux functions: paste; **BEDTools**: makewindows, coverage

- BED, GFF3/GTF, SAM/BAM and VCF files are all tab delimited text files used for describing features of chromosomal intervals. Software like BEDTools, BEDOPS, VCFTools, SAMtools, BAMtools, et al. are often used in combination with basic Linux functions to process these files. In this exercise, you will use BEDTools, a very efficient tool for analyzing chromosomal interval files, e.g. within each intervals of file A, count the occurrences of features in file B.
- For this exercise, you will first generate a text file with sliding windows across the chromosome. The input file for the "makewindows" function is a text file with the length of each chromosomes (hg19.txt). The "-w" and "-s" options specify the window and step size for the sliding windows. In this example, the sliding window size is 1 mb.
- In the next two steps, you will count the number of genes and pseudo-genes in each sliding window. To do this, you can use "awk" to select lines with "gene" or "pseudogene" in column 3, use "bedtools coverage" to count the number of genes in each sliding window. The sorting step after "bedtools coverage" is necessary because bedtools tend to output un-sorted results. In this case, you sort the file by two columns: column 1(chromosome name)

and column 2(position). Note that you need to use "version" style sorting for column 1 (-1,1V) and numerical sorting for column 2 (-k2,2n). As chromosome names are like version number strings, e.g. chr1, chr2, ..., chr10. With "V" sorting, the "chr1" will be placed before "chr10", with general text sorting, the "chr1" will be placed after "chr10. The two numbers in "-k1,1V" indicate start and end columns for sorting.

- The "paste" function was used to concatenate the columns, followed by "cut" to output selected columns. Note: we use "paste" here because we know that the two files have same number of corresponding rows. If not sure, you need to use "join" function.

```
bedtools makewindows -g hg19.txt -w 1000000 -s 1000000  >  win1mb.bed


gunzip -c human.gff3.gz | \
awk 'BEGIN {OFS = "\t"}; {if ($3=="gene") print $1,$4-1,$5}' | \
bedtools coverage -a win1mb.bed -b stdin -counts | \
LC_ALL=C sort -k1,1V -k2,2n  > gene.cover.bed


gunzip -c human.gff3.gz | \
awk 'BEGIN {OFS = "\t"}; \
{if (($3=="processed_pseudogene") || ($3=="pseudogene")) print $1,$4-1,$5}' | \
bedtools coverage -a win1mb.bed -b stdin -counts  | \
LC_ALL=C sort -k1,1V -k2,2n > pseudogene.cover.bed


paste gene.cover.bed pseudogene.cover.bed | \
cut -f 1,2,3,4,8 > genecounts.txt
```

7. In this exercise, you will use fastx command FASTQ file to trim sequencing adapters, then get size distribution of the trimmed sequencing reads.

Linux functions: grep, wc -l, awk;   **FASTX**:

- First you will estimate the percentage of sequencing reads that contain the adapter sequence " AGATCGGAAGAGC". As the file could be very big, you could estimate the percentage based on the first 10,000 sequencing reads. (Note: sometimes the first 10,000 reads could be all low quality reads, then this estimation would not be accurate. You might want to try a few blocks of reads at different places of the file by using "head -n xxxxx | tail -n xxxxx".) The "grep" function in this command is used to select lines that contain a specific string, followed by "wc -l" function that count the number of such lines. By doing this, you will find ~48% of the reads in this file contains the adapter sequence.
- Now, you will remove the adapter sequences. For this, you will use a specialized tool fastx_clipper which allows mismatches, and write the output into a new fastq file "clean.fastq". You can pipe the "gunzip -c" output directory into input stream of fastx_clipper.
- Now, you will get the read length distribution. In the fastq file, each sequence record has 4 lines and the second line of the record is the actual DNA sequence. The "awk" function has a variable "NR" that records the line number for each row. The expression (NR%4) gives you the remainder of NR divided by 4. The statement "if (NR%4 == 2) print length($0)" means "print the size of the second line in every 4-line sequence record". `The output of awk can then be piped` into "LC_ALL=C sort -n | uniq -c" to get the read size distribution.

```
gunzip -c SRR836349.fastq.gz |head -n 40000 | grep AGATCGGAAGAGC | wc -l


gunzip -c SRR836349.fastq.gz | fastx_clipper -a AGATCGGAAGAGC -Q33 > clean.fastq


awk '{if (NR%4 == 2) print length($0)}' clean.fastq | LC_ALL=C sort -n | uniq -c
```

8. Running multiple independent tasks in parallel on a multi-CPU machine

As a simple example of multiple independent task problem, we will consider compressing several (here: 5) large files, **`reads_1.fastq`**, **`reads_2.fastq`**, …, **`reads_5.fastq`** using **`gzip`** compression tool. The compression should be run in parallel using several (e.g., 3) CPU cores.

Instructions:

Your scratch directory **`/workdir/XXXXX`** should already contain the five **`*.fastq`** files mentioned above - verify this using the **`ls`** command. If the files are not present, copy them as follows (replacing **`XXXXX`** with your user ID):

```
cd /workdir/XXXXX
```

```
cp /shared_data/Linux_workshop2/reads_*.fastq  .
```

Create a text file, called **`my_tasks`**, containing the list of tasks to be executed, i.e., the five **`gzip`** commands, one in each line. The file should look like this:

```
gzip reads_1.fastq
gzip reads_2.fastq
gzip reads_3.fastq
gzip reads_4.fastq
gzip reads_5.fastq
```

This file can be created, for example, using a text editor, such as **`nano`** or **`vim`**. Such hand-editing will work fine for small number of task lines, but not when this number is, say, a 1000 or more. In the latter case, it would be more practical to use the loop capability of the **`bash`** shell by typing:

```
for i in {1..5}
do
echo "gzip reads_${i}.fastq"
done > my_tasks
```

The shell commands between **`do`** and **`done`** (here: just one **`echo`** command) will be repeated as specified in the **`for`** statement, i.e., 5 times, each time with the variable **`i`** incremented by 1, starting with 1 and ending with 5 (we could also write **`for i in 1 2 3 4 5`**, which would be equivalent to the present statement, or **`for i in 1 3 5`**, which would make **`i`** assume only values 1, 3, and 5). The current value of **`i`** is accessed inside the loop with using syntax **`${i}`**. The command **`echo`** just prints its argument (stuff between **`""`**) to the screen, so between **`do`** and **`done`** 5 lines will be generated. Thanks to the redirection operator **`>`**, all these lines will be saved to the file **`my_tasks`** instead of being printed to the screen. **NOTE** that you can always use the for loop to execute multiple similar shell commands, indexed by a variable like **`i`**. **NOTE** also, the index **`i`** may be called differently (like **`x`**, **`Rob`**, or **`tuna`**, or anything else), and that the values over which it loops do not have to be integers - strings would be just fine if more convenient for indexing of the commands within the loop.

After creating the **`my_tasks`** file, especially if you did this using the shell loop described above, verify that it makes sense by looking at a few first lines, e.g.,

```
head -2 my_tasks
```

Now that the task file **`my_tasks`** is created, it is time to run these tasks in parallel! To try this, run the following command:

```
perl_fork_univ.pl my_tasks 3 >& my_tasks.log &
```

Immediately after submitting the above command (thanks to the **`&`** at the end it will run in the background and free the terminal for further action), run **`ls -alrt`** a few times and verify that the compressed files **`reads_1.fastq.gz`**, ..., **`reads_5.fastq.gz`** are being created and are growing in size. First, you should see three such files, then the remaining two. While the operation is running, the original (uncompressed) versions of the files will co-exist with the partial compressed ones. Once the compression of a file is completed, its original version is deleted.

Run the command **`top -u yourID`** - you should see your three **`gzip`** processes on top of the list, consuming sizeable fraction of CPU time (to exit **`top`** - press **`q`**).

<u>So, what is happening here?</u> The script **`perl_fork_univ.pl`** (located in **`/programs/bin/perlscripts`**) reads the list of tasks **`my_tasks`** and launches the first three simultaneously, on separate CPU cores. Once any of these initial three tasks completes freeing up its CPU core, the next task is launched in its place (with two others still running). The process continues until all tasks are finished, while the number of CPU cores being utilized never exceeds 3. This way, the <u>load</u> is <u>balanced</u> between the three CPU cores. A report from the run is saved in the log file **`my_tasks.log`**. It contains some useful timing information you may want to analyze.

<u>A few remarks:</u>

The script **`perl_fork_univ.pl`** is a <u>simple way to parallelize any number of independent</u> tasks on a multi-core machine. Just construct the task list file and run it through the script. The tasks **do not need to be similar** (as it is the case in our simple example), however, they need to be **independent** from one another.

How many CPU cores to use (here: 3)? It depends on many factors, such as:

- <u>total number of CPU cores on a machine</u>: launching more tasks than available CPUs cores is counter-productive
- <u>memory taken by each task</u>: combined memory required by all tasks running simultaneously should not exceed about 90% of total memory available on a machine; memory taken by each of your running processes can be monitored with **`top`**
- <u>disk I/O bandwidth</u>: if all simultaneously running tasks read and write a lot to the same disk, they will compete for disk bandwidth, so running too many of them may slow things down instead of speeding them up (this is, actually, the case in our example, since **`gzip`**-ing is a disk-intensive operation).
- <u>other jobs running on a machine</u>: if present, they also take CPU cores and memory! Example: during this exercise, there are about **7** other people working on your machine, each trying to run on 3 CPUs!

Typically, determining the right number of CPUs to run on requires some experimentation.