BioHPC workshop Linux for Biologists: Exercises Part 2

Exercise 1: using scp to transfer files between Linux machines

From remote to local machine: On server `cbsulogin3.biohpc.cornell.edu` there is a directory `/workdir/Workshop/dir2transfer` with some test files in it – all readable to you. Copy this directory (with al its content) to your scratch directory on your workshop machine.

```
cd /workdir/your_id
scp -r your_id@cbsulogin3.biohpc.cornell.edu:/workdir/Workshop/dir2transfer .
```

(note the dot "`.`" at the end of the last command - this means the destination is your current directory). If asked whether your are sure about connecting (it will happen if connecting to `cbsulogin3` for the first time), answer 'yes'. If asked, provide your BioHPC password. When done, verify the transfer and examine the directory you copied

```
ls -l dir2transfer
```

From local to remote machine: In your scratch directory (`/workdir/your_id`) on your workshop machine, create a text file called `your_id_file` (replace `your_id` with your actual BioHPC user ID), containing a string "`This is my test file`". Use your favorite text editor *(e.g.*, nano) to create this file.

Copy the newly created file onto `cbsulogin3.biohpc.cornell.edu` into the directory `/workdir/Workshop`

```
cd /workdir/your_id
scp your_id_file your_id@cbsulogin3.biohpc.cornell.edu:/workdir/Workshop
```

In order to verify that the file has been successfully copied, we will list the files in the remote directory using `ssh`:

```
ssh your_id@cbsulogin3.biohpc.cornell.edu ls -al /workdir/Workshop
```

(provide your BioHPC password if prompted). Note the use of `ssh` in the command above: instead of logging in to a remote machine and opening a terminal on it, we are just running a single command (`ls -al`).

Exercise 2: batch download of files from sequencing facility

Open your e-mail, find a message with subject line "**Test Illumina distribution e-mail**" with an attachment `download.sh`.

Transfer the attachment file onto your Linux machine. You can do one of the following:

Option 1:

- open the attachment in a text editor on your laptop and copy its contents to clipboard (using the mouse)
- in Linux machine terminal, open a new file (in a directory where you want your files downloaded to) using a text editor of your choice (e.g., `nano` or `vim`)
- paste the contents of the clipboard to the new file on Linux machine and save that file.

Option 2:

- Save the attachment file on disk on your laptop
- Use a file transfer technique of your choice (interactive `sftp` client like **FileZilla**, or command-line `scp`) to transfer the saved file from laptop to your Linux machine, to the directory where you want the `fastq` files to be downloaded to.

Once the file `download.sh` is ready on the Linux machine:

- log in to the Linux machine (if not yet done so)
- cd to the directory where the `download.sh` file has been deposited
- execute the file

```
sh ./download.sh
```

Once the download completes (should take about 1 second):

Verify (using the `ls -al` command) that the files have ben downloaded and that they have correct sizes (the same as in the notification e-mail)

Verify that MD5 sums of both files are the same as in the notification e-mail. An MD5 sum is a unique signature of a file, computed from the file's content using a certain algorithm. If the MD5 sums of two files are identical, you can say that the files are identical. Therefore, checking the MD5 sums is often used to verify the correctness of the file transfer.

First, compute the MD5 sums of the downloaded files, saving he output to a file on disk, then print that file to the screen and confront with MD5 sums in the notification e-mail:

```
md5sum file_1.fastq.gz file_2.fastq.gz >& md5sums.log
cat md5sums.log
```

Do the MD5 sums agree?

Now un-compress the `fastq` files

```
gzip -d file_1.fastq.gz file_2.fastq.gz
```

When the operation completes, you should see the original files `file_1.fastq` and `file_2.fastq` (verify with `ls -al`).

Open each file in a text editor on Linux machine (`nano` or `vi`) and examine the file's format. Note there are 4 lines corresponding to each read: the header line with read's name, the line with read's sequence, another line containing just "`+`" (sometimes followed by the read's header), and the last line containing base calling qualities encoded as ASCII characters.

Count reads in each `fastq` file

```
wc -l file_1.fastq file_2.fastq
```

(Note: `wc` -l will return the number of lines in each file, which is 4 times the number of reads)

Exercise 3: run a simple BLAST search

In this exercise, we will perform a BLAST search of 9 randomly selected human cDNA sequences against the database Swissprot database of amino acid sequences. We will use the program blastx (a part of blast+ suite of programs) which internally performs a 6-frame translation of each DNA query and compares each of these translations to all reference amino acid sequences from the database.

Prepare input data

First, we shall prepare the input data. In your scratch directory `/workdir/my_id`, create a subdirectory `blast_test` and navigate to it:

```
cd /workdir/your_id
mkdir blast_test
cd blast_test
```

Now copy the FASTA file with query sequences (this file is located in the workshop directory):

```
cp /shared_data/Linux_workshop/seq_tst.fa .
```

While you're at it, take a peek inside this file by opening it in `nano` or `less`. Note that each sequence may be broken into multiple lines preceded by a header line with the sequence's name and possibly various metadata that may be available. Each header line starts with an ">" character, and header lines are the only places in a FASTA file where this character may occur. Thus, counting the number of lines containing "`>`"

```
grep ">" seq_tst.fa | wc -l
```

gives us the number of sequences in the file (which in this case should be 9). Note the use of the pipe construct: the output from the grep filter, *i.e.*, all lines of `seq_tst.fa` containing "`>`", is treated as input to `wc -l`, which counts such lines.

Now copy the **Swissprot** database files into a separate subdirectory directory we will make for this purpose within `/workdir/your_id/blast_test` (which at this point should be your current directory):

```
mkdir databases
cp /shared_data/Linux_workshop/databases/swissprot* ./databases
```

Note the use of a wildcard "`*`" in the source argument of the copy command above - it mans that all files from `/shared_data/Linux_workshop/databases` with names starting with string `swissprot` are to be copied to our subdirectory `databases`. To verify, list the content of the new directory:

```
ls -al databases
```

You should see 7 files there. These files are a result of conversion of the original FASTA file with **Swissprot** amino acid sequences into a format that `blast+` suite of programs works with. This suite contains a tool (called `makeblastdb`) to perform such formatting on any FASTA file you wish to treat as a BLAST database. For the purpose of this exercise, this formatting step has been completed ahead of time so that the files you just copied are ready to use.

Run the BLAST search

You may want to read through this whole exercise before starting it.

Before starting this task, open another terminal window on your Linux machine. You can either log in again via `ssh` or start a new shell within your screen or VNC session (if you have one open). In one of the windows launch the `top` program to show a dynamically updated list of your processes:

```
top -u your_id
```

In the other window (where top is **not** running), make sure your current directory is `/workdir/your_id/bast_test`

```
cd  /workdir/your_id/bast_test
```

Launch BLAST by entering the following command (all in a single line):

```
blastx -db ./databases/swissprot -num_alignments 1 -query seq_tst.fa -out
hits.txt >& run.log &
```

As you see above, the `blastx` program accepts a lot of options, some of them obligatory, some optional. The meaning of the options we use is the following:

`-db` points to the database files to be used. `blastx` needs to know the prefix of the database file names, *i.e.*, the common part of the file names preceded by any directory path necessary to find the files - in our case this prefix is `./databases/swissprot`.

`-query` specifies the `fasta` file with query sequences to be processed - in our case: `seq_tst.fa`.

`-out tells` the program where to store the output - here it will be a file hits.txt in the current directory

-`num_alignments`: number of BLAST hits for which semi-pictorial representation of the alignment will be printed to the output file - here: just one, top hit, for all other hits we will just have the alignment score information, but the alignment itself will not be shown.

Any screen output and/or error messages (STDOUT+STDERR) the command will produce will be saved in file `run.log` in case error analysis is needed later on (note that the BLAST results themselves will be saved in hits.txt and are therefore not a part of output to terminal).

The ampersand "&" at the very end of the command will send the task to the background, *i.e.*, after you hit ENTER, your terminal will return to the prompt and you will be able to run other tasks or commands while your BLAST search is running behind the scenes.

Monitor the run

The runs you just submitted will take about 1 minute. While it is in progress, you may do certain things to monitor it:

Look at the output from `top` you started earlier in the other terminal window (or shell in screen or VNC). You should see your process `blastx` somewhere on top of the list, consuming about 100% of CPU (*i.e.*, one thread).

Run `ls -al` a few times in the current directory - you should see the file `hits.txt` being created and growing in size (however, for this small example, the output file may be stored in memory and not be written to disk until the very end of the BLAST run).

List processes running on the machine, filtering the ones that belong to you. The `blastx` process should be on the list.

```
ps -ef | grep your_id
```

When the run completes (in about 1-2 minutes), the `blastx` process will drop off the `top` and `ps` lists, the size of file `hits.txt` will stop growing, and you will receive a termination message in the main terminal window (the one you launched the task in).

Examine the output

First, check the file run.log for any suspicious messages that may indicate the failure of the run (open this file musing `nano` or `less`).

Again, using `nano` or `less`, open and examine the output file `hits.txt`. Detailed interpretation of this file is beyond the scope of this workshop. In short, for each query sequence, you should see a list of hits (database sequences and coordinates where the given query aligns), along with alignment score measures, such as e-value and bit score. For the best hit, you should also see the actual alignment.

Run again using 2 CPUs

Recall the `blastx` command you just ran (using up/down arrows or history, for example) and modify it by adding one more option: `-num_threads 2`. You may also want to modify the name of the output file (*e.g.*, to `hits2.txt`). After modifying - hit ENTER to run the task again. Monitor the `top` output in the other window. How much `%CPU` is the program now using? Is it going to finish faster than when run on 1 CPU?

When finished, compare the new output file `hits2.txt` with the previous one, `hits.txt`:

```
diff hits2.txt hits.txt
```

If you get no output at all, this will mean the files are identical. They should be, since they result from the same BLAST search, just run in slightly different ways.

Exercise 4 : Basic shell scripting

To illustrate some basic functionality of shell scripts, we will now revisit the BLAST job from the previous exercise. The BLAST command used in that exercise was rather long and cumbersome, and would become even more so if more options were specified. Here we will create a shell script which will simplify and generalize launching a BLAST search for any query file and at the same time provide some extra functionality.

With `/workdir/your_id/blast_test` as your current directory (`cd` there if needed), create (using `nano`) a new file called `blast_script.sh`:

`nano blast_script.sh`

with the following content (you may want to copy paste this into the file rather than type everything in):

```
#!/bin/bash

# Collect the two arguments (quewry file name and number of CPUs to run on)
# and assign easy to remember variables to them

QFILE=$1
NCPU=$2

# If the number of CPUs not given, assume 1

if [ "$NCPU" == "" ]
then
NCPU=1
fi

# Relate the output file name to input file name (i.e., if the input file is
# AAA, then the output file will be AAA.hits.txt)

OFILE=${QFILE}.hits.txt

blastx \
-db ./databases/swissprot \
-num_alignments 1 \
-num_threads $NCPU \
-query $QFILE \
-out $OFILE \
>& run.log
```

Make sure you save the script file! Once it is ready, change its attributes so that it is seen by Linux as executable by you:

`chmod u+x blast_script.sh`

In essence, a script is a set of shell commands that will be executed in order of appearance when the script is run. The very first line determines what program will be used to interpret and execute the commands that follow (here: the `bash` shell). Except for the first line, anything following the `#` character is treated as comment *(i.e.*, not interpreted).

The script you just created works like a command taking two arguments: the name of the query file and the number of CPUs to use. In the script, these arguments are 'intercepted' and referred to as `$1` and `$2`, respectively. In order to make the script easier to read, two variables are defined, `QFILE` and `NCPU`, in which the argument values are stored. The values of these variables are used later in the script and referenced as `$QFILE` and `$NCPU`. If the second

argument is not given on command line, it is assumed to be 1 (the conditional `if - then - fi` statement serves this purpose).

The essence of the script is the `blastx` command you practiced previously. In addition to options already discussed in previous exercise, there is one more option, `-num_threads`, which tells the program how many CPU threads to use (the second argument given to the script, stored as `NCPU` and referenced as `$NCPU`). Furthermore, the values of the `-query` and `-out` parameters are also read off the variables, `QFILE` and `OFILE`, respectively. Notice that the name of the output file, stored in variable `OFILE`, is constructed out of the name of input file by adding a suffix `.hits.txt`.

In the script, the lengthy `blastx` command has been split into multiple lines, each ending with the backslash `\` character (make sure that this character is really the last one - not followed by any blank spaces, for example). This split of a long command into shorter lines is often convenient (makes the script more readable) but not necessary - the whole command could have been written all in one single line.

To summarize: the script we just created illustrates a few (and definitely not all) useful possibilities of shell scripting: passing of arguments, defining and referring to environment variables, conditional statements, and splitting of long command strings into multiple lines.

Now that we understand what our script is supposed to do, let's launch it given the query file `seq_tst.fa` and assuming we want to use 2 CPUs for the task:

`./blast_script.sh seq_tst.fa 2 &`

The whole script will be running in the background (note the `&` at the end). As you observe the `top` output in the other terminal window or run the `ps -ef | grep my_id` command to list your processes, you should see all the commands programmed in the script including the shell instance running the script appearing consecutively as they are being invoked. In this case, since all operations before the `blastx` are very fast, the latter process will be the only one you notice taking any CPU. In fact, it should be taking close to 200% as it was the case with the stand-alone `blastx` command run on 2 threads. Once the run ends and you list the directory content, you should see the output file `seq_tst.fa.hits.txt`.

Exercise 5: Multiple independent tasks (and more scripting)

As an example of processing of multiple independent tasks, we will consider compressing three files using `gzip`. First, copy the three example files to your `/workdir/your_id` directory:

```
cd /workdir/your_id
cp /shared_data/Linux_workshop/auxfiles/BBB_* .
```

You should now see three files, `BBB_1`, `BBB_2`, and `BBB_3` in your directory when your run `ls -al`. Now create a shell script called `gzip_script.sh` (open a new file in the `nano` editor). You can use one the three equivalent versions of that script:

Version 1:

```
#!/bin/bash

gzip BBB_1
gzip BBB_2
gzip BBB_3
```

Version 2:

```
#!/bin/bash

for i in BBB_1 BBB_2 BBB_3
do
    gzip ${i}
done
```

Version 3:

```
#!/bin/bash

for i in {1..3}
do
   gzip BBB_${i}
done
```

In all three versions of the script, the tree files will be compressed consecutively, one after another, so that only one `gzip` command will be running at any given time. While in script Version 1 the commands are programmed explicitly, Versions 2 and 3 use the shell's '**for loop**' construct to accomplish the same thing: the command between keywords `do` and `done` is repeated with loop index `i` assuming values specified in the `for` statement. These values may be strings (as in Version 2) or numbers (as in Version 3). They are referred to as `${i}` (or - if no string concatenation is involved - simply as `$i`). BTW, the index variable does not have to be called `i` - it can be called `j`, `nnn`, `kitty`, or any string you want.

Save your script upon exit from `nano`, make it executable, and run it:

```
chmod u+x gzip_script.sh
./gzip_script.sh &
```

Observe the `top -u your_id` output in the other terminal and/or output from the `ps -ef | grep your_id` command (run it several times 30 s or so apart). You should see only one `gzip` process at a time, first operating on `BBB_1` file, then `BBB_2`, and finally on `BBB_3`.

Now edit your `gzip_script.sh` file and put an ampersand (`&`) at the end of each `gzip` command. For example, if you used Version 3 of the script, the modified file will look like this:

```
#!/bin/bash

for i in {1..3}
do
   gzip BBB_${i} &
done
```

Prepare the example files again (by decompressing the compressed versions you just obtained)

```
gunzip BBB_*.gz
```

and run the script again:

`./gzip_script.sh`

Notice that even though you did not run the whole script in the background (no `&` after the command above), it exited immediately and your terminal returned to the prompt. This is because each of the `gzip` commands has been submitted in the background inside the script. After the first of these commands was submitted, the script did not have to wait for it to finish, but instead it submitted the second one - also in the background, so that the third `gzip` command could have been run right after that. After submitting the third `gzip` to the background, the script did not have anything else to do, so it exited. However, all three `gzip` commands it left behind keep running simultaneously in the background. In the output from top, you should now see three such processes.

Exercise 6: Multiple independent tasks on a limited number of CPUs

Suppose we have a large number (thousands) of files to compress, but only a few processors on which to do this. In order not to overwhelm the machine, we need a mechanism that would run only a small number of such tasks simultaneously while keeping the rest queued up until CPUs become available. This exercise illustrates how this can be done.

We will use the three example files from Exercise 5. Prepare the files in `/workdir/your_id`:

```
gunzip BBB_*.gz
```

Using `nano` editor, create a file - call it `tasklist` - containing the list of commands to be executed:

```
gzip BBB_1
gzip BBB_2
gzip BBB_3
```

As an aside, note that the file above could be easily created using the shell's **for loop** construct:

```
rm -f tasklist
for i in `ls -1 BBB_*`
do
echo gzip $i >> tasklist
done
```

This is pretty similar to the `for` loop we used in scripts of Exercise 5, except that the index list in the `for` statement is now constructed from the output of the `ls -1 BBB_*` command (which just returns a list of all file names starting with `BBB_`). The statement `echo` simply prints out what follows it to the terminal, and that gets re-directed (and appended) to the file `tasklist` on disk. Of course, there is really no advantage of using a for loop as above to create a `tasklist` for just three files, but for a list of length, say, 1000 - this would really be the only option. Note also, that the **for** loop above can be run directly from the terminal prompt by ENTERing the lines one-by-one, or you may choose to create simple shell script which you can later modify.

Now that the list of tasks is ready, we will execute them using 2 CPUs, so that initially two first `gzip` commands will run simultaneously, while the third will wait until one on the initial two completes. Such procedure of queueing multiple tasks to be run on limited number of CPUs without overwhelming those CPUs is often referred to as load balancing. In this exercise, we will practice two simple ways to do this.

Option1: use the CBSU in-house load-balancing script `perl_fork_univ.pl` . Simply run

```
/programs/bin/perlscripts/perl_fork_univ.pl  tasklist  2 >& log &
```

In the command above, `tasklist` is the list of tasks (here: `gzip` commands) you created earlier, and '`2`' is the number of CPUs to use (which may of course be bumped up if you have enough tasks and sufficient CPUs at your disposal). All processes created by the command above will run in the background and any terminal output the script generates will be saved in the file called `log`.

Option 2: use **GNU parallel** tool

GNU parallel is a standard Linux tool with functionality similar to (and somewhat broader than) that of `perl_fork_univ.pl`. Before using `parallel` for the first time it is good to silence the 'citation request', which would otherwise appear every time you run the tool. To get rid of the request, run the following:

```
/programs/parallel/bin/parallel --citation
```

then type `will cite` and hit ENTER.

Once the warning is silenced, run the three commands from your `tasklist` with load balancing on 2 CPUs as follows

```
/programs/parallel/bin/parallel --jobs 2 < tasklist >& log &
```

Whichever of the two Options you choose, you should observe at most two CPUs being used at the same time, executing your `gzip` tasks. The first two will be started simultaneously, and the third will wait until one of the first two completes.