Further Readings for Week 4

First, let us look at a couple of commands that are quite helpful to compare file contents. These commands are useful when we want to compare log files or output files from analyzing similar samples or lists of some kind, for example, gene lists between two organisms to see how many genes they have in common or unique to a specific organism.

`comm`

```
$ whatis comm
comm (1)             - compare two sorted files line by line
```

`comm` command is useful for comparing two files line by line and see what is unique to each file and common to both files. Let's see an example using two files containing some names of fruits.

```
$ cat fruits1.txt
Mango
Apple
Peach
Banana
Watermelon
Jackfruit

$ cat fruits2.txt
Orange
Pear
Banana
Apple
Jackfruit

$ comm fruits1.txt fruits2.txt
Mango
comm: file 1 is not in sorted order
Apple
        Orange
Peach
Banana
        Pear
comm: file 2 is not in sorted order
        Banana
        Apple
        Jackfruit
Watermelon
Jackfruit
comm: input is not in sorted order
```

As mentioned in the description of `comm`, the command expects it's inputs to be sorted similar to `uniq` and complains if it is not. Let's provide sorted inputs:

```
$ sort fruits1.txt > sorted_fruits1.txt
$ sort fruits2.txt > sorted_fruits2.txt

$ cat sorted_fruits1.txt
Apple
Banana
Jackfruit
Mango
Peach
Watermelon

$ cat sorted_fruits2.txt
Apple
Banana
Jackfruit
Orange
Pear

$ comm sorted_fruits1.txt sorted_fruits2.txt
                Apple
                Banana
                Jackfruit
Mango
        Orange
Peach
        Pear
Watermelon
```

We can see 3 columns generated as output, first column contains lines that are unique to file 1, second column contains lines unique to file 2 and last column contains lines common to both files.

Before we see some options supported by `comm`, let us see a better, one-line command to achieve what we did above in separate steps, that is, sort the files, save them into different files and then give those as inputs to `comm`. Note that `comm` expects files as inputs and does not read from STDIN directly. We can use **process substitution** that we learned to rewrite the command to directly give sorted files as input to `comm`:

```
$ comm <(sort fruits1.txt) <(sort fruits2.txt)
                Apple
                Banana
                Jackfruit
Mango
        Orange
Peach
        Pear
Watermelon
```

`comm` allows customizing the output to only see columns we are interested in. We can use `-1` or `-2` to suppress lines unique to files 1 or 2, or `-3` to suppress lines common to both files.

```
$ comm -3 <(sort fruits1.txt) <(sort fruits2.txt) # shows only columns 1 and
2
Mango
        Orange
Peach
        Pear
Watermelon
```

We can also combine the options to suppress multiple columns. To view only lines that are common to both the files, we can suppress columns 1 and 2:

```
$ comm -12 <(sort fruits1.txt) <(sort fruits2.txt) #gives only column 3
Apple
Banana
Jackfruit
```

diff

```
$ whatis diff
diff (1)             - compare files line by line
```

`diff` compares the lines of input files and provides instructions in a way to make the files content identical. In a way, this is analogous to the CIGAR string in SAM files. Unlike `comm`, `diff` does not require sorted inputs.

Let us use `diff` on the fruits files we saw earlier with slight modifications:

```
$ cat fruits1.txt
Mango
Apple
Peach
Banana
Watermelon
Jackfruit
Cherry

$ cat fruits2.txt
Orange
Mango
Pear
Banana
Apple
Jackfruit

$ diff fruits1.txt fruits2.txt
0a1
> Orange
2,3c3
```

```
< Apple
< Peach
---
> Pear
5c5
< Watermelon
---
> Apple
7d6
< Cherry
```

To understand the output, it's easier if we know the meanings of symbols used by `diff`'s default mode:

- `<` - refers to the content in the first file
- `>` - refers to the content in the second file
- `a` - refers to add
- `c` - refers to change
- `d` - refers to delete
- `---` - used as separator between contents of two files

The general format of the instruction line is: `<line(s) from file 1><action><line(s) from file 2>`

We'll now try to interpret the results of the `diff` command in parts -

```
0a1
> Orange
```

`0a1` means at line 0 in file 1, add line 1 of file 1, that is, `Orange` shown in the next line. We'll keep track of this to see if we end up with an output that is similar to one of files: `Orange`

```
2,3c3
< Apple
< Peach
---
> Pear

5c5
< Watermelon
---
> Apple
```

Here, there are two instructions -

- `2,3c3` - At lines 2 and 3 in file 1, change to line 3 in file 2, that is replace `Apple` and `Peach` in file 1 with `Pear` in file 2. After this change, we have: `Orange Mango Pear`.

If you're wondering where `Mango` come from, remember that `diff` only gives instructions for changing lines that are different and ignores lines that are common to both files. Note that `Mango` appears at line 2

in both files. Simiarly, `Banana` appears in both files at line 4, so we'll add it to our list: `Orange Mango Pear Banana`.

- `5c5` - At line 5 in file 1, change to line 5 in file 2 by replacing `Watermelon` with `Apple`. After this change, we have: `Orange Mango Pear Banana Apple`.

```
7d6
< Cherry
```

The final instruction `7d6` tells to delete line 7 in file1 to sync both the files at line 6.

At line 6, we have `Jackfruit`, common to both files. So our list now becomes `Orange Mango Pear Banana Apple Jackfruit`.

Now, if we compare our list with fruits2.txt, we see that they are the same.

```
$ cat fruits2.txt
Orange
Mango
Pear
Banana
Apple
Jackfruit
```

The output obtained from `diff` is useful to see the differences between the file contents, but what if we need to make two files identical? `diff` has a `-u` option that can create a **patch file** tracking the differences and we can apply this patch to create a file with identical content using `patch` command. Let us look at a demo:

```
$ diff -u fruits1.txt fruits2.txt > fruits_diff.patch

$ cat fruits_diff.patch
--- fruits1.txt 2024-09-13 20:45:34.979957549 -0400
+++ fruits2.txt 2024-09-13 20:54:16.151015979 -0400
@@ -1,7 +1,6 @@
+Orange
 Mango
-Apple
-Peach
+Pear
 Banana
-Watermelon
+Apple
 Jackfruit
-Cherry

$ patch -i fruits_diff.patch -o fruits_ident.txt
patching file fruits_ident.txt (read from fruits1.txt)
```

```
$ cat fruits_ident.txt
Orange
Mango
Pear
Banana
Apple
Jackfruit
```

The patch file is in **unified** format, a more compact version of the default version we saw earlier. You can check the document linked at the end of the section to know how to interpret the differences in unified format. It is interesting to note that git also uses the unified format of `diff` for highlighting changes since the last commit.

```
$ cat fruits.txt
Mango
Apple
Peach
Banana
Watermelon
Jackfruit
Cherry

$ git add fruits.txt
$ git commit -m "adding fruits.txt"
[main 034ec3e] adding fruits.txt
 1 file changed, 7 insertions(+)
 create mode 100644 fruits.txt

$ echo -e "Orange\nMango\nPear\nBanana\nApple\nJackfruit" > fruits.txt
#modify the content of the file

$ git diff fruits.txt
diff --git a/fruits.txt b/fruits.txt
index d1039ff..45932c1 100644
--- a/fruits.txt
+++ b/fruits.txt
@@ -1,7 +1,6 @@
+Orange
 Mango
-Apple
-Peach
+Pear
 Banana
-Watermelon
+Apple
 Jackfruit
-Cherry

$ diff -u fruits1.txt fruits2.txt
--- fruits1.txt 2024-09-13 20:45:34.979957549 -0400
+++ fruits2.txt 2024-09-13 20:54:16.151015979 -0400
@@ -1,7 +1,6 @@
```

```
+Orange
 Mango
-Apple
-Peach
+Pear
 Banana
-Watermelon
+Apple
 Jackfruit
-Cherry
```

We can see that the output from `git diff` is the same as the one from `diff -u`.

`-y` option supports side-by-side comparision of files.

```
$ diff -y fruits1.txt fruits2.txt
                                                        > Orange
Mango                                                     Mango
Apple                                                   | Pear
Peach                                                   <
Banana                                                    Banana
Watermelon                                              | Apple
Jackfruit                                                 Jackfruit
Cherry                                                  <
```

Here, `>` is equivalent to `a`(addition) from above, `|` to `c`(change/replace) and `<` to `d`(deletion).

You can learn about more modes and options supported by `diff` here.

Parallelization

We learned that loops repeat a set of actions iteratively, one at a time, over a given set of inputs until a condition is satisfied. If we have multiple independent inputs and the repeated action we perform on those inputs are also independent, we can think of processing multiple inputs simultaneously to speed things up. Processing multiple inputs simultaneously to decrease the computational time and increase the efficiency is the core idea behind **parallelization**.

We'll look at such a command that can be used for processing inputs in parallel below:

`xargs`

```
$ whatis xargs
xargs (1)            - build and execute command lines from standard input
```

`xargs` takes inputs from STDIN and passes them as command-line arguments to any command that follows. This idea is new and different from what we have seen so far, so, let us see some examples to understand its working:

```
$ echo  file{1..8}
file1 file2 file3 file4 file5 file6 file7 file8

$ ls


$ echo file{1..8} | xargs touch

$ ls
file1  file2  file3  file4  file5  file6  file7  file8
```

Here, we first `echo`ed a few filenames which is then piped to `xargs` followed by `touch`. `xargs` takes the output of `echo` and provides that as inputs to the command that comes after, that is, `touch` which creates the files.

```
$ ls fruits* | xargs wc -l
 7 fruits1.txt
 6 fruits2.txt
13 total
```

`ls` lists the files matching the glob pattern `fruits*` which is taken by `xargs` to provide as inputs to `wc -l`.

If you're thinking we could have achived the above results by just executing `touch file{1..8}` and `wc -l fruits*` and not convinced about `xargs`'s usefulness, we'll see a few more examples.

`-n` option: controls the number of inputs passed

```
$ echo  file{1..8} | xargs -n 3 echo
file1 file2 file3
file4 file5 file6
file7 file8
```

In the above examples, `xargs` is passing 3 inputs at a time to the `echo` command to its right which `echo`s in batches of 3. We can see the commands executed behind the scenes by xargs using `-t` option:

```
$ echo  file{1..8} | xargs -t -n 3
echo file1 file2 file3    <-----
file1 file2 file3
echo file4 file5 file6    <-----
file4 file5 file6
echo file7 file8          <-----
file7 file8
```

A practical situation where we can use `-n` is when working with paired-end files. We can pass 2 files at a time to a command or program like so:

```
$ ls SRR373829_*.fq | xargs -n 2 ./process_fastq.sh
```

`-I` option: specifies a placeholder to implement input substitution

```
$ ls
data  file1  file2  file3  file4  file5  file6  file7  file8

$ ls file* | xargs -t -I {} mv {} data/
mv file1 data/
mv file2 data/
mv file3 data/
mv file4 data/
mv file5 data/
mv file6 data/
mv file7 data/

$ ls data/
file1  file2  file3  file4  file5  file6  file7  file8
```

We specified `{}` as the placeholder and `xargs` replaces this placeholder with the actual values it receives as input. Note that we can use other special symbols like %, $, @ and so on as placeholder.

An example usecase where this can come handy is if we have VCF files of 22 human chromosomes named "chr1.vcf.gz", "chr2.vcf.gz" and so on that we want to process using `process_vcf.sh` script, we can use `xargs`:

```
$ seq 22 | xargs -I {} ./process_vcf.sh chr{}.vcf.gz > updated_chr{}.vcf.gz
```

The `seq` command generates numbers 1 to 22 and `xargs` replaces the placeholder in filenames with the output of `seq` at the time of execution.

`-P` option: specifies the number of parallel processes to run

This is the option that allows `xargs` to parallelize the execution of a command or program on multiple inputs.

```
$ ls data/
file1  file2  file3  file4  file5  file6  file7  file8

$ ls data/file* | xargs -t -P 4 -n 2 gzip
gzip data/file1.gz data/file2.gz
gzip data/file3.gz data/file4.gz
gzip data/file5.gz data/file6.gz
gzip data/file7.gz data/file8.gz

$ ls data/
file1.gz  file2.gz  file3.gz  file4.gz  file5.gz  file6.gz  file7.gz
file8.gz
```

As we see from the output of the second command, `xargs` created 4 processes taking 2 inputs simulataneously and executing `gzip` in parallel.

A practical scenario where we can apply this: imagine we have a list of sample IDs and want to run a script or program parallely:

```
$ cat sampleIDs.txt | xargs -P 8 -I {} bash -c "fastp -i {}.fastq -o
{}_trimmed.fastq"
```

`xargs` creates 8 processes for running `fastp` program on the input fastq files for every sample listed in the `sampleIDs.txt` file. Note here that we created a bash subshell using `bash -c` for running the `fastp` program. By using this syntax for creating bash subshell, we can run multiple commands(separated by semicolons) and/or use core bash functionalities with `xargs`.

How do we decide the number we give with `-P`? That depends on the number of cores your system supports. If you have a octa-core processor, you can use `-P 8` to run 8 processes in parallel. To check the number of processors, you can use either `nproc` or `lscpu | grep "CPU(s)"` commands.

Finally, we'll see how `xargs` compares with loops in terms of parallelization with examples. We'll make use of the `time` command that can be prefixed with any command to measure the time taken for execution.

```
$ time for i in {1..4}; do sleep 10; echo "Iteration $i: Sleeping for 10
seconds"; done
Iteration 1: Sleeping for 10 seconds
Iteration 2: Sleeping for 10 seconds
Iteration 3: Sleeping for 10 seconds
Iteration 4: Sleeping for 10 seconds

real    0m40.020s
user    0m0.010s
sys     0m0.011s
```

The loop runs sequentially 4 times and in every iteration, it sleeps for 10 seconds. We'll focus on the `real` time above, it is 4*10 = 40 seconds as expected.

If we use `xargs` without `-P` option, it is equivalent to running things sequentially, similar to a loop. Let us confirm this:

```
$ time seq 1 4 | xargs -t -n 1 sleep 10
sleep 10 1
sleep 10 2
sleep 10 3
sleep 10 4

real    0m50.052s
user    0m0.016s
sys     0m0.005s
```

Why do we see 50 seconds instead of expected 40 seconds? Remember that `xargs` provides the input it receives as arguments to the command that follows it. So `sleep` receives two arguments: 10, which we have defined and one from `seq` as seen from the `xargs` output. When `sleep` receives multiple arguments, it adds up the time and sleeps for those many seconds. So, in total, it sleeps for (10+1) + (10+2) + (10+3) + (10+4) = 50 seconds.

Now, we'll run this using `-P` to see the effect of parallelization:

```
$ time seq 1 4 | xargs -t -P 4 -n 1 sleep 10
sleep 10 1
sleep 10 2
sleep 10 3
sleep 10 4

real    0m14.009s
user    0m0.012s
sys     0m0.002s
```

It completed in 14 seconds! That is because the longest sleep time was (10+4) seconds and the other `sleep` processes which had lower sleep time finished executing in parallel.

This is just the tip of iceberg of what we can do with `xargs`. You can read more about other options supported by `xargs` and see more examples here, here and here.