The entire code repository can be found at https://github.com/dgsaf/hpc-assignment-4.

Contents

List of Figures

1 Interpretation

The `count` process is presented in Listing 4.

```
48 process count {
49   input:
50   path(files) from files_ch.collect()
51
52   output:
53   file("results.csv") into counted_ch
54
55   container = ""
56
57   shell:
58   '''
59   echo "seed,ncores,nsrc" > results.csv
60   files=($(ls table*.csv))
61   for f in ${files[@]}; do
62     seed_cores=($(echo ${f} | tr '_.' ' ' | awk '{print $2 " " $3}'))
63     seed=${seed_cores[0]}
64     cores=${seed_cores[1]}
65     nsrc=$(echo "$(cat ${f} | wc -l) - 1" | bc -l)
66     echo "${seed},${cores},${nsrc}" >> results.csv
67   done
68   '''
69 }
```

Listing 1: The process `process count` in `nextflow/main.nf`.

i. `echo "seed,ncores,nsrc" > results.csv`

The output of the left hand side (SNR seed, number of cores used and number of sources counted) which would normally be sent to `stdout` is redirected to the destination specified on the right hand side (in this case, a file called `results.csv`) creating/overwriting the file in the process.

ii. `echo "${seed},${cores},${nsrc}" >> results.csv`

Similar to the effect of `>` above, except that this operator appends to the file, rather than overwriting it.

iii. `cat ${f} | wc -l`

The output of the left hand (normally send to `stdout`) is *piped* to input of the right hand side; that is, the input of `wc -l` is taken from the output of `cat ${f}`. This effect of this command is to count the number of lines in the file with filepath `${f}`.

iv. `$(<command>)` and `($(<command>))`

Wrapping a shell command with a single pair of parentheses, `(<command>)` executes that command in a sub-shell - prefixing this with a dollar sign captures the final output of that subshell `$(<command>)`. However if the result of this is a list of whitespace-delimited strings, wrapping this in a further set of parentheses, `($(<command>))`, will capture the output as an array variable.

v. `$(ls table*.csv)`

The effect of `ls table*.csv` is to list all files in the current directory which start with `table` and end with `.csv`. The effect of `$(ls table*.csv)` is to capture this list of matching files as a string. Specifically, it collects the files containing tables of counted sources for all combinations of `seed` and `cores`.

vi. `echo ${f}`

The effect of this command is to output (to `stdout`) the string-value of the variable `f`. Specifically, this outputs the filepath of a specific file, which is taken from the list of tables described above, and will be of the form `table_<seeds>_<cores>.csv`.

vii. `tr '_.' ' '`

The effect of this command is, for a given input, to translate all instances of the characters `_` and `.` into whitespace. Specifically this transforms `table_<seeds>_<cores>.csv` into `table <seeds> <cores> csv`.

viii. `awk '{print $2 " " $3}'`

The effect of this command is, for a given input consisting of lines (delimited by `\n`) containing records (delimited by whitespace), to print the second record, followed by a white space, followed by the third record, for each line in the input. Specifically, this transforms `table <seeds> <cores> csv` into `<seeds> <cores>`.

ix. `cat ${f}`

The effect of this command is to output the contents of the file, with filepath `${f}` - specifically `table_<seeds>_<cores>.csv`.

x. `wc -l`

The effect of this command is to, for a given input, count the number of newlines (occurences of `\n`) in the input. Specifically, this counts the number of newlines in the file `table_<seeds>_<cores>.csv`.

xi. `echo "$(cat ${f} | wc -l)-1" | bc -l`

The effect of this command is to count the number of sources in the file `table_<seeds>_<cores>.csv`, by first counting the number of lines in the file, and then subtracting for the number of lines which are not source data records.

2 Development

```
72 counted_ch.into{counted_for_ch; counted_xargs_ch}
```

Listing 2: The channel `counted_ch` is duplicated, with one for each of `process plot_for`, and `process plot_xargs`, in `nextflow/main.nf`.

2.1 Bash For Loop

```
75 process plot_for {
76   input:
77   path(table) from counted_for_ch
78
79   output:
80   file("*.png") into final_for_ch
81
82   cpus 4
83
84   shell:
85   '''
86   ncores_set=$(awk -F, '{if (NR != 1) {print $2}}' !{table} | sort | uniq)
87
88   for ncores in $ncores_set ; do
89     python !{projectDir}/plot_completeness.py \
90     --infile !{table} --outfile plot_for_${ncores}.png --cores $ncores
91   done
92   '''
93 }
```

Listing 3: The process `process plot_for` in `nextflow/main.nf`.

2.2 xargs Command

```
96 process plot_xargs {
97   input:
98   path(table) from counted_xargs_ch
99
100  output:
101  file("*.png") into final_xargs_ch
102
103  cpus 4
104
105  shell:
106  '''
107  ncores_set=$(awk -F, '{if (NR != 1) {print $2}}' !{table} | sort | uniq)
108
109  printf '%s ' $ncores_set | xargs -n1 -P4 -I ncores -d ' ' \
110  python !{projectDir}/plot_completeness.py \
111  --infile !{table} --outfile plot_xargs_ncores.png --cores ncores
112  '''
113 }
```

Listing 4: The process `process plot_xargs` in `nextflow/main.nf`.

3 Execution

3.1 SNR Plot

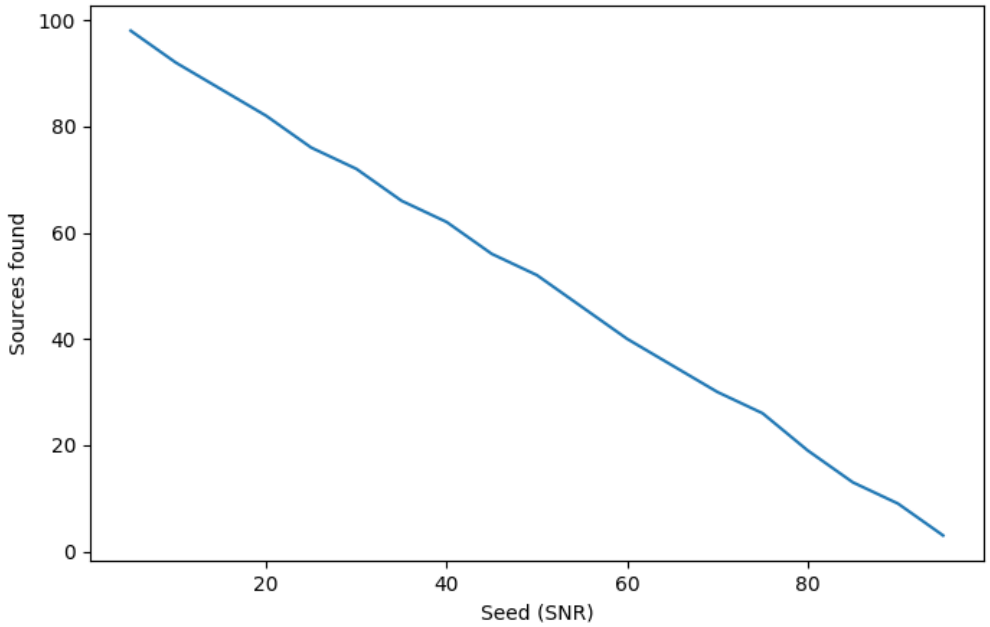

Figure 1: The Signal-to-Noise Ratio (SNR) is presented for a range of seed SNR values. This figure was produced by `process plot_for`*, for the case of* `cores = 1`*. No difference was observed between this plot and any of the other plots produced for any value of* `cores`*, nor whether if* `process plot_for` *or if* `process plot_xargs` *were used.*

3.2 Workflow DAG

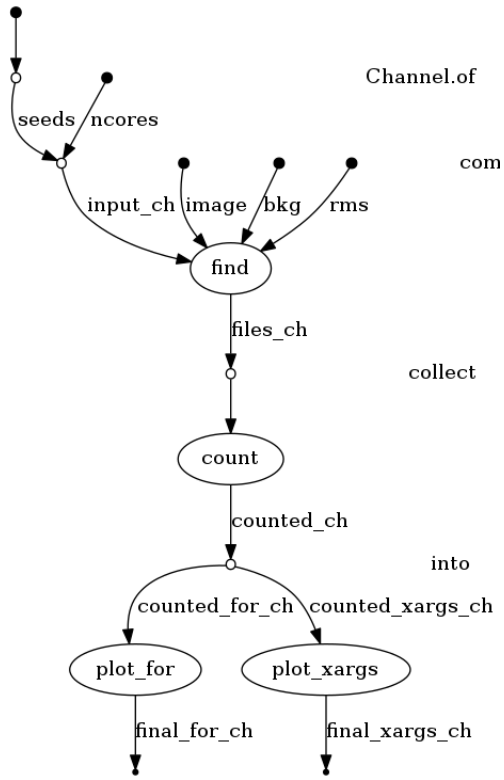

Figure 2: The Directed Acyclic Graph (DAG) of the workflow is presented. Note that the `combine` *operator (below* `Channel.of`*) appears to have been cropped out by the tool producing the DAG.*

4 Analysis

4.1 Resource Usage

4.1.1 CPU Usage

The CPU usage was fairly high across all processes: with a median single-core CPU usage of 77.6% for `find`, 73.7% for `count`, and 90.8% and 93.5% for `plot_for`, and `plot_xargs` respectively. Notably the process `find` had the widest range of CPU usage, ranging from a minimum of 61.9% to 81.0%. Overall, these statistics suggest a fairly good single-core CPU utilisation, although perhaps the processes `find` and `count` could be further optimised.
However, the usage of all allocated CPUs, by the processes `plot_for` and `plot_xargs`, was 22.7% and 23.4% respectively. It is seen in subsubsection 4.1.4 that these processes are heavily involved in I/O operations - which would explain why the single-core CPU usage of these processes is rather good, while the load-balancing among cores is not. Many of the cores will be idle while reading and writing operations are being performed.

4.1.2 Memory Usage

The process `find` utilised the largest amount of memory, with a median memory usage of 224.9 M, extending up to a maximum of 320.9 M. For the other processes, `count` had a rather insignificant memory usage of 4.379 M, while `plot_for` and `plot_xargs` used 60.3 M and 117.5 M respectively. Notably the xargs version of the plotting process was more memory intensive than the bash for loop version - this reflects that it uses parallel processes to iterate through the plotting, while the for loop iterates serially.

4.1.3 Time Usage

The process `find` accounted for the majority of execution time, averaging from approximately 6 s to 7 s per task for small SNR seed values to approximately 5 s for larger SNR seed values. The process `count` accounted for a negligble amount of execution time, requiring only 1.8 s, while the processes `plot_for` and `plot_xargs` both required approximately 4.8 s of execution time.
This is unexpected, as one would imagine the xargs version to be faster, given that it executed 4 parallel processes, while the bash loop executed the same 4 processes serially. However, we note that since the I/O usage is the limiting factor for the `plot` processes, specifically reading plain text files from disk (discussed in subsubsection 4.1.4). Hence, it doesn't that the process are run in parallel, since they will all be bottlenecked by the read speed - essentially causing the parallel processes to run serially, as each waits in queue for its data to be read.

4.1.4 I/O Usage

The I/O usage was dominated by the processes `plot_for` and `plot_xargs` reading from file, with a total of 49.39 M being read for each process. This is to be expected, as the plotting is performed using data stored in a plain-text file. The process `count` used a neglible amount of I/O, while `find` read approximately 24.5 M per task.

4.2 Areas for Improvement

The % usage of all allocated CPUs, by the `plot` processes requires addressing - specifically, to minimise the percentage of time spent performing I/O. Currently the script `plot_completeness.py` is executed to produce a plot for a given value of cores numerous times - causing numerous reads of the same file. This could be improved by modifying the script to instead produce plots for all cores values in one execution, which would eliminate duplicate file reads, thus minimising I/O and improving the % usage of allocated CPUs.