Workflow Assignment Report

Don Wijesinghe

03/06/2024

1 Interpretation

1. The`>` **in** `echo "seed,ncores,nsrc" > results.csv`: Redirects the output of the `echo` command to a file named `results.csv`, overwriting the file if it already exists.
2. The `>>` **in** `echo "${seed},${cores},${nsrc}" >> results.csv`: Appends the output of the `echo` command to the file `results.csv`.
3. The `|` **in** `cat ${f} | wc -l`: Pipes the output of `cat ${f}` to the `wc -l` command, which counts the number of lines in the file `${f}`.
4. The difference between `$(<command>)` **and** `($(<command>))`: `$(<command>)` executes the command and returns its output. `($(<command>))` is a subshell that executes the command within parentheses, which doesn't affect the current shell's environment
5. `$(ls table*.csv)`: Captures the output of the `ls table*.csv` command, which lists all files matching the pattern `table*.csv`.
6. `echo ${f}`: Prints the value of the variable `f` to the terminal.
7. `tr '_.' ' '`: Translates (replaces) all underscores and periods in the input with spaces.
8. `awk '{print $2 " " $3}'`: Extracts the second and third columns from the input and prints them, separated by a space.
9. `cat ${f}`: Prints the contents of the file `f` to the terminal.
10. `wc -l`: Counts the number of lines in the input.
11. `echo "$(cat ${f} | wc -l)-1" | bc -l`: Calculates the number of lines in the file `${f}`, subtracts 1 (to exclude the header), and passes the result to `bc -l`, which evaluates a mathematical arithmetic (in this case its some number - 1) and returns the result with floating-point precision.

2 Development

A code snippet for the two versions of the plot script, one using a `for` loop, and one using `xargs`:

```
process plot {
    input:
    path(table) from counted_ch

    output:
    file('*.png') into final_ch

    cpus 4

    script:
    """
    unique_cores=\$(awk -F',' '{print \$2}' ${table} | tail -n +2 | sort -u)
    for cores in \${unique_cores[@]}; do
              python3 ${params.codedir}/plot_completeness.py --infile ${table}
              --outfile plot\${cores}.png --cores \${cores}
```

```
    done
    """
}

process plot {
    input:
    path(table) from counted_ch

    output:
    file('*.png') into final_ch

    cpus 4

    script:
    """
    unique_cores=\$(awk -F',' '{print \$2}' ${table} | tail -n +2 | sort -u)
    echo \${unique_cores[@]} | tr ' ' '\\n' | xargs -n 1 -P 4 -I {} python3
    ${params.codedir}/plot_completeness.py --infile ${table} --outfile plot_xargs_{}.png --cores {}

    """
}
```

3 Execution

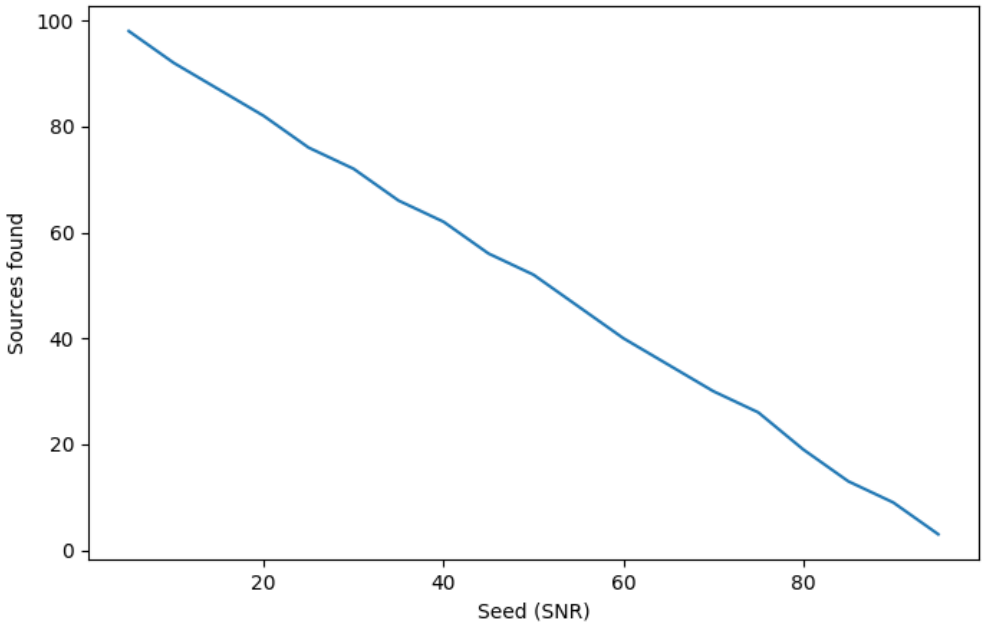

Figure 1: Plot for `cores=1` obtained using the serial method for plotting.

The plots generated for different numbers of cores were identical, indicating that the source-finding algorithm (`aegean`) provides consistent results regardless of the number of cores used. This consistency suggests that the algorithm's behavior is independent of parallel processing in terms of the results produced.

The workflow diagram in Figure 2 illustrates the sequential processing of data using Nextflow. The process starts with the creation of static and dynamic channels. Static channels (image_ch, bkg_ch, rms_ch) are used for the constant input files, while dynamic channels (seeds, ncores) generate combinations of seed values and core counts (input_ch). The find process uses these channels to run the Aegean source finding program, producing the table result files (files_ch). These result files are then aggregated by the count process into a summary CSV file (results.csv). Finally, the plot process generates plots from the aggregated results.

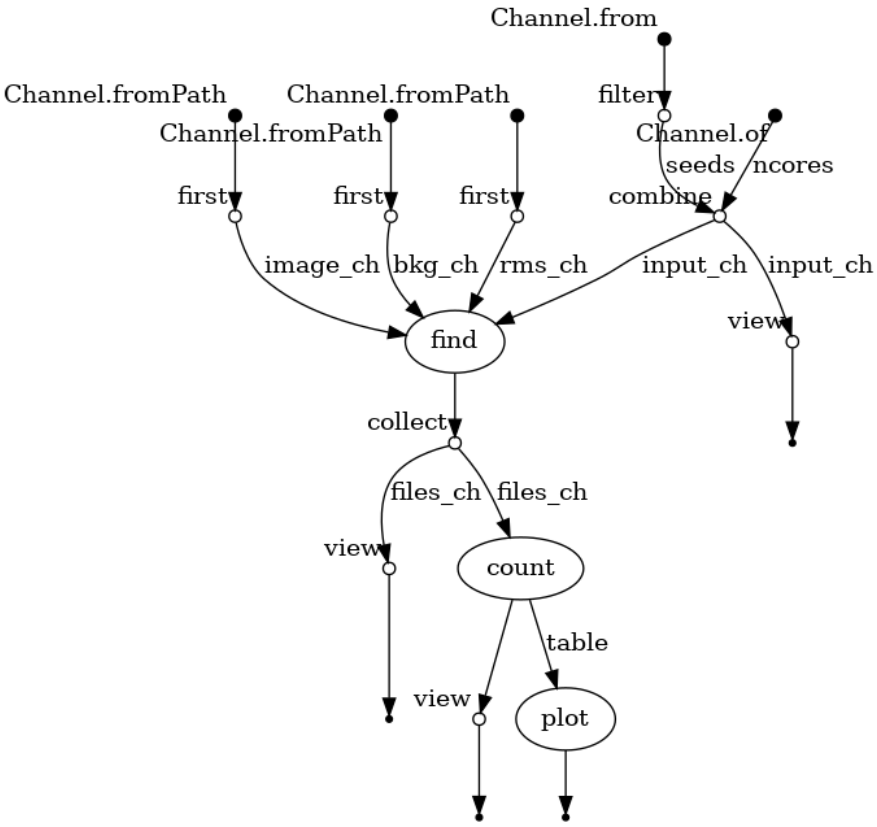

Figure 2: Workflow Graph

4 Analysis

The run time for the serial method was around 5mins 40sec while using `xargs` to plot graphs concurrently gave a total execution time of 4mins and 57secs. The `find` process exhibits a mean single-core CPU usage of 77%, with a range between 60% and 100%. However, when considering allocated CPUs, the usage drops significantly, with a mean value of 28%, a maximum of 80%, and a minimum of 10%. This discrepancy indicates that while the process is utilising a significant portion of a single core, it is not efficiently using the allocated multiple cores. This variaility could be due to when the process is run by a low number of cores, although the mean value remains pretty low. The underutilisation suggests potential for improving parallel execution within this process. The count process stands out with an unexpectedly high CPU usage—140% for both single-core and allocated CPU metrics. This could be due to the process being over-allocated or an inefficiency in the way it handles its tasks. Given that this process only runs once, and only accounts for a fraction of the total run time, it can be ignored. The plot process shows consistent CPU usage in both single-core and allocated CPU graphs. When running the plotting script serially, the single-core usage is 107%, and the allocated CPU usage is 26.9%. Using `xargs` to run the plotting script four times concurrently increases the single-core usage to 179% and the allocated CPU usage to 45%. This indicates better utilisation of resources when parallel execution is employed, highlighting the efficiency gains achievable with parallel processing. The find process is the most memory-intensive, with a maximum RAM usage of 600MB, a minimum of 120MB, and a mean of 307MB. This significant variability is due to the python multiprocessing module used. When using multiprocessing, each process is a separate instance with its own memory space. This means each process will have its own copy of the data and variables, leading to increased overall memory usage plus some overhead. The virtual memory usage also shows high variability, with a maximum of 3.5GB and a mean of 1.7GB, indicating potential inefficiencies in memory management. To improve this, we can possibly use shared memory from the multiprocessing module to minimise duplication of memory into child processes. The count process uses the least amount of RAM, consistently at 3.4MB, and similarly low virtual memory usage. The plot process shows consistent RAM usage at 51MB for the serial version and 184MB for the parallel `xargs` version. Virtual memory usage increases from 287MB in the serial version to 1.1GB in the parallel version. When the plotting script runs in parallel using `xargs`, multiple instances (up to four in your case) of the script are executed concurrently. Each instance loads the data into memory independently, leading to an increase in total RAM usage and virtual memory. The find process has the longest execution time, with each job

averaging 0.1 minutes and exhibiting significant variability, reaching up to 0.2 minutes. For the serial version, the plot process completes in 0.1 minutes for a single job. The parallel `xargs` version significantly reduces the overall execution time, demonstrating the benefits of parallel execution in speeding up the plotting task. The plot process surprisingly shows the highest read bytes at 50MB for both serial and parallel versions, followed by the find process at 24MB, while the count process has the least read bytes. The plot process benefits significantly from parallel execution, as shown by the `xargs` implementation. Further enhancing parallel execution can improve efficiency and reduce execution time. Optimising memory usage, particularly for the find process, can lead to significant performance gains. Using of staged data and minimising memory overhead are potential strategies. Dividing large input files into smaller, manageable chunks allows for independent and parallel processing which leads to efficient data handling. Also, using data streaming techniques to process data in chunks rather than loading entire datasets into memory helps manage large volumes of data effectively without exhausting memory resources. The `find` process can also be further parallelised where calculations for each tuple can be done in parallel although this will require a large amount of memory especially for larger datasets.