Formative assessment week 9: Snakemake, Conda, HPC

You have so far:

- Generated the code that performs your analysis
- Developed a reproducible environment for your project
- Integrated the environment and code into a Snakemake pipeline
- Version controlled the code, pipelining and environments using git and GitHub

We will now:

- Migrate the code, data and environments to HPC
- Update the Snakemake configuration to run on HPC
- Document these changes for ease of future reproducibility

1. Setting up on HPC

A. Getting our codebase on HPC

Your project should all be under version control on GitHub. This makes it extremely easy to create a clone of your code on another machine (such as HPC).

1. Login to bluecrystal4
2. Create a directory where you would like to keep your projects. A good option is to create a directory called `$WORK/projects/`
3. Navigate to `$WORK/projects` and clone your GitHub repository there.
    - Add the git module in bluecrystal4
    - Get the git repository's HTTPS URL from GitHub
    - Use `git clone` to clone the repository

If your code is not on github you can use `scp` to copy your code into the relevant directory.

Example steps:

```
mkdir -p $WORK/projects/
cd $WORK/projects
git clone https://github.com/explodecomputer/MSHDS-AHDS-formative.git
```

B. Getting our data on HPC

Use `scp` to copy your original data into `$WORK/projects/<formative>/data/original`. You can see how the data was setup originally on your computer in the `directory-setup-commands.sh` script.

C. Setup your conda environment

You should have conda ready to run on HPC from the practical sessions. Create the environment needed for your formative assessment.

Remember you will need to add the slurm plugin (https://snakemake.github.io/snakemake-plugin-catalog/plugins/executor/slurm.html).

Example:

```
. ~/.initMamba.sh
conda env create -f ahds_formative_environment.yml
pip install snakemake-executor-plugin-slurm
```

D. Check that your scripts work as expected

- Ideally you would only run analysis on HPC by submitting your scripts to the worker nodes. However it can be helpful to check that trivial errors are not present such as incorrect directory names etc before submitting your job. You can check this by **starting** to run your analysis scripts in the login node to see if it works and then killing it as quickly as possible (`ctrl+c`).

2. Create a submission script

Using examples from the week 9 `practical_1` example, create a job submission script that will run your pipeline on HPC by submitting a single job that runs all the steps.

Example file contents for e.g. `run_analysis.sh`:

```
#!/bin/bash

set -e

#SBATCH --job-name=test_job
#SBATCH --partition=teach_cpu
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --time=0:10:00
#SBATCH --mem=100M
#SBATCH --account=SSCM033324
#SBATCH --output ./slurm_logs/%j.out


cd "${SLURM_SUBMIT_DIR}"

# Conda environment
source ~/.initMamba.sh
mamba activate ahds_week9

# Setup directories
mkdir -p logs
mkdir -p data/derived
mkdir -p data/original
mkdir -p results

# Run steps
cd code
bash 1-data-check-bm.sh > ../logs/1-data-check-bm.log
bash 2-data-check-accel.sh > ../logs/2-data-check-accel.log
bash 3-data-fix-accel.sh > ../logs/3-data-fix-accel.log
bash 4-list-accel-ids.sh > ../logs/4-list-accel-ids.log
Rscript 5-generate-sample.R > ../logs/5-generate-sample.log
Rscript 6-demo-data-prep.R > ../logs/6-demo-data-prep.log
```

To submit this:

```
sbatch run_analysis.sh
```

3. Use Snakemake to submit the analysis to HPC

A. Snakemake dry-run

Try a dry-run of the whole pipeline *without* submitting it to HPC (e.g. just run it interactively as if you were running it on your local computer). Check for any issues and fix.

Example:

```
snakemake -pn
```

B. Snakemake execution

You have setup a `slurm_profile` that can specify how the Snakemake pipeline can be executed on the HPC worker nodes. Using the examples in the week 9 `practical_2` session, try submitting the whole pipeline to HPC.

Note that it may be prudent to run this within a `tmux` session (why?)

You should have a slurm profile specified in e.g. `~/.config/snakemake/slurm_profile/config.yaml` that looks like:

```
default-resources:
  - slurm_partition="teach_cpu"
  - slurm_account="SSCM033324"
  - mem_mb="100"
  - runtime="10"
  - ntasks="1"
  - nodes="1"
jobs: 10
printshellcmds: True
scheduler: greedy
use-conda: True
```

You can submit the Snakemake pipeline using

```
snakemake --executor slurm --profile slurm_profile
```

Running this inside `tmux` is sensible because it provides a persistent session to allow the Snakemake pipeline to continue watching and submitting jobs to the scheduler. This is helpful for longer running jobs because if you ran it just in your interactive shell session then when you log out (or get disconnected) then the Snakemake process will be killed.

4. Update your project repository

Given the changes that you have made, what should you do to finalise and record these processes?

- Update your `REAMDE.md` to explain how the job should be exectured using a submission script or Snakemake.
- Update your `README.md` to include the config template for Snakemake to run and how to get it installed
- Update your `environment.yml` file to include the slurm plugin e.g. `conda list -e > environment.yml`
- Commit your changes to git and synchronise with GitHub.

5. Advanced: Re-factoring to run in parallel

Currently the whole pipeline is run sequentially. That means that each job waits for the previous job to finish before starting. This does not take advantage of the benefits of HPC which can run multiple jobs in parallel.

Identify areas of your pipeline that could feasibly be split into separate jobs that can run in parallel. Refactor your pipeline to take advantage of this, and implement it into Snakemake.

This is a tricky one to solve, but we can get some clues from the `topic-wildcards` example from the Week 7 Snakemake practical. Here's the strategy.

How to break up the long running jobs. The `2-data-check-accel.sh` and the `3-data-fix-accel.sh` scripts are the longest running jobs because they each do operations on 7000+ files. The first task is to modify these scripts to accept a command line argument that specifies only to work on a subset of the files. The second task is to update the Snakefile to call these scripts multiple times with different subsets of the files.

Updating the .sh scripts

Notice that in the `data/original/accel/` directory, the files are named `accel-31###.txt`, `accel-32###.txt`, . . . , `accel-41###.txt`. We can use this to our advantage. Modify the `2-data-check-accel.sh` script to accept a command line argument that specifies the prefix to work on:

```
batch=$1

if [ -z "$batch" ]; then
    echo "No batch provided. Doing everything"
    batch=""
fi
```

and then modify the main process of the script to only work on files that match the batch. For example change

```
cat ${DATADIR}/original/accel/accel-*.txt | grep -v '<' | awk -F'\t' '{print NF}' | grep -v 8 | sort -u
```

to

```
cat ${DATADIR}/original/accel/accel-${batch}*.txt | grep -v '<' | awk -F'\t' '{print NF}' | grep -v 8 |
```

Wherever you see `accel-*.txt` in the script, replace it with `accel-${batch}*.txt`. Now to run `2-data-check-accel.sh`, you can either run it on the whole set of files:

```
bash code/2-data-check-accel.sh
```

or only on a subset of the files:

```
bash code/2-data-check-accel.sh 31
```

We would use the same logic to update the `3-data-fix-accel.sh` script. See the code repository at https://github.com/explodecomputer/MSHDS-AHDS-formative for how these files look after they've been updated.

Updating the Snakefile

The next step is to update the Snakefile to call these scripts multiple times with different subsets of the files. Currently, our Snakefile starts by listing all the files in the `data/original/accel/` directory so that it knows what the inputs need to be, like this:

```
import glob
import re
```

```
# Get the list of pid values from the filenames in data/original/accel/
accel_files = glob.glob("data/original/accel/accel-*.txt")
ACC_PID = [int(re.search(r"accel-(\d+).txt", f).group(1)) for f in accel_files]
```

We can build on this by using simple python commands to split the list of `ACC_PID` values into 11 sub-lists, where each sub-list is all the files that are in that batch.

```
# The list of PID prefixes to use as batches
indexes = [str(x) for x in range(31, 42)]

# Create a dictionary of PID prefixes to lists of PID values
# This is a dictionary of lists, one list for every index in `indexes`
ACC_PID_dict = {index: [pid for pid in ACC_PID if str(pid).startswith(index)] for index in indexes}
```

What this means is that our rules can now be split by each item in `indexes`. Here is the original rule that we used to run `2-data-check-accel.sh`:

```
# Check the accelerometer data:
rule check_accel_data:
    input:
        "logs/1-data-check-bm.log",
        expand("data/original/accel/accel-{pid}.txt", pid=ACC_PID)
    output:
        "logs/2-data-check-accel.log"
    log: "logs/2-data-check-accel.log"
    shell:
        """
        cd code
        bash 2-data-check-accel.sh 2>&1 ../{log}
        """
```

It runs `2-data-check-accel.sh` on all the files in `data/original/accel/`. We can now split this rule into 11 rules, one for each batch:

```
for index in indexes:
    rule:
        name: f"{index}_check_accel_data"
        params: index=f"{index}"
        input:
            "logs/1-data-check-bm.log",
            expand("data/original/accel/accel-{pid}.txt", pid=ACC_PID_dict[index])
        output:
            f"logs/2-data-check-accel_{index}.log"
        log: f"logs/2-data-check-accel_{index}.log"
        shell:
            """
            cd code
            bash 2-data-check-accel.sh {params.index} 2>&1 ../{log}
            """
```

Let's examine what is going on here.

1. We have a for loop, which loops over each list in the list of lists `indexes`.
2. We've defined a parameter inside the rule called `index`. So in the first iteration of the loop, `index` will be `31`, in the second iteration it will be `32`, and so on.
3. The inputs have been modified - now we only expect as inputs the `accel-31*.txt` files in the first iteration, the `accel-32*.txt` files in the second iteration, and so on. In the previous version we

expected all the files.

4. The output has been modified - now we expect as output the `logs/2-data-check-accel_31.log` file in the first iteration, instead of just `logs/2-data-check-accel.log`.
5. The shell command has been modified - now we pass the `index` parameter to the `2-data-check-accel.sh` script, so that it only works on the files that match the batch.

We'll have to update the `all` rule as well, because it now needs to find all the `logs/2-data-check-accel_*.log` files instead of just `logs/2-data-check-accel.log`.

```
rule all:
    input:
        expand("logs/2-data-check-accel_{index}.log", index=indexes)
```

Using this logic, the rule that runs `3-data-fix-accel.sh` can be split in the same way. Have a look at the `Snakefile-batches` file in the code repository at https://github.com/explodecomputer/MSHDS-AHDS-formative. To run this version of the Snakefile, you would use this command if you're running it on your laptop:

```
snakemake -c1 -s Snakefile-batches
```

or this command if you're running it on HPC:

```
snakemake --executor slurm --profile slurm_profile -s Snakefile-batches
```

Notice that by splitting up steps `2` and `3` into batches, we can now run them in parallel if we have enough compute resources to do this. i.e. if you have many cores on your laptop you can use `snakemake -c 10 -s Snakefile-batches` to run up to 10 jobs in parallel. Or running on HPC will automatically submit jobs to the scheduler to run in parallel.

You can see a minimal example of this batching process in the `Snakefile-batches-minimal-example` file in the code repository at https://github.com/explodecomputer/MSHDS-AHDS-formative. This splits up the `accel-*` in the same way, but is a very simple pipeline, where step 1 is to copy each file to a new directory, and step 2 is to list all the copied files. But it demonstrates the principle of splitting up the jobs into batches.