Helpful HPC tools: Modules, Jupyter, and Snakemake

Matt Stone

Roy Group Meeting — 23 October 2019

Quick overview

- Goal: introductory survey, not comprehensive education
  (will supplement this info with elogs, and happy to take questions after)
- 10 minutes on environment modules
- 10 minutes on Jupyter
- 20 minutes on Snakemake
- 10-20 minutes from Sushmita on gdb

Environment modules

- Environment modules permit “loading/unloading” of software by scripting the setting/unsetting of environment variables (e.g. `$PATH`)
- Makes it easy to switch between versions or share an installation between users
- Can build modules for executables or libraries
- Supported on most HPC systems; can install on any Unix machine
- https://modules.readthedocs.io/en/latest/
  http://modules.sourceforge.net

Module commands

- `load $module` : activate a module
- `unload $module` : de-activate a module (remove from any paths)
- `avail` : show available modules, and their modulefile directories
- `use $dirpath` : add a directory of modulefiles to be searched
- `help $module` : print help for a module (more in depth)
- `whatis $module` : print info about what a module is (one-liner)

Creating your own modules

1. Download/compile your program into a version-specific directory
   (I like to standardize the location)
   `/mnt/dv/wid/projects2/Roy-common/programs/thirdparty/${program}/${version}/`

2. Create a modulefile for your program (details next slide)
   `/mnt/…/Roy-common/programs/thirdparty/modulefiles/${program}/${version}`

3. Make sure to include the modulefile directory with `module use`

4. Load your module!

What does a modulefile look like?

```
#%Module1.0

proc ModulesHelp { } {
        global version

        puts stderr "This loads parallel-20180922 environment"
}

module-whatis   "Loads parallel-20180922 environment"

set             parallel_root       /royfs_write/mstone/modules/apps/parallel/20180922
prepend-path    PATH                $parallel_root/bin
append-path     MANPATH             $parallel_root/share/man/
```

Modulefiles are just Tcl scripts

`module load` runs these commands, **`module unload`** undoes them

Other commands, other paths

- `setenv` : set environment variables
- `prereq` : require other modules as dependencies before loading
- `conflict` : don't load module if a conflicting module is loaded
- `set-alias` : create an alias
- Paths: `PATH`, `MANPATH`, `LD_LIBRARY_PATH`, `C_INCLUDE_PATH`, `CPLUS_INCLUDE_PATH`, `JULIA_DEPOT_PATH`, etc
- https://modules.readthedocs.io/en/latest/modulefile.html

Jupyter

- Jupyter notebooks allow you to mix code, text, and figures in one file
- Can include Python, R, or Julia, but best support is for Python
- Helpful for exploratory analysis and plotting, making reports
- Not ideal for development or deployment
  (see talk from Joel Grus: https://bit.ly/2MAULXa)

Structure of a Jupyter notebook

- Notebooks are composed of “cells”
- Cells can be code or Markdown
- Write code like you normally would
- Code cells are run by pressing “shift+enter”
- Markdown cells are rendered by pressing “shift+enter”

Structure of a Jupyter notebook

Jupyter demo

This is a rendered Markdown cell

[1]:
```
# this is a code cell.
# it displays output beneath it

x = 1
print(x)
```

```
1
```

```
this is an unrendered Markdown cell
```

[2]:
```
# this is a code cell with no output
x += 1
```

[4]:
```
# variables persist between cells
print(x + 3)
```

```
5
```

[ ]:
```
# this is a code cell I haven't run yet

print(x + 2)
```

Syntax for plotting figures in Jupyter

Plotting in Jupyter

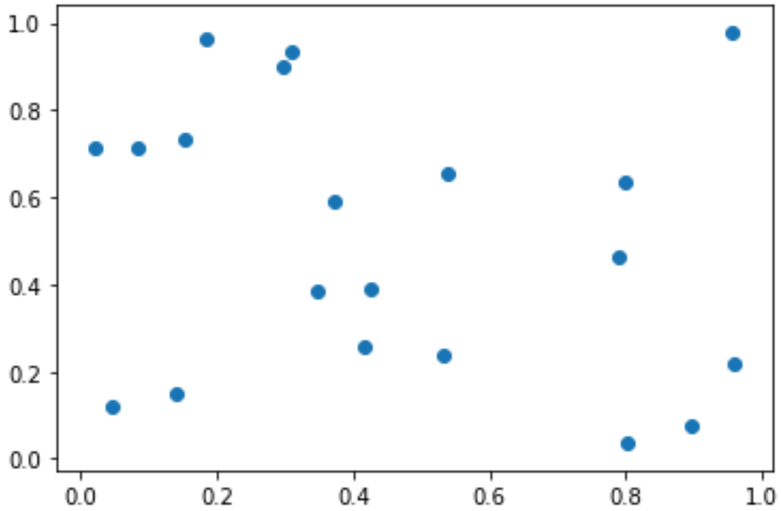
[5]:
```
# commands that begin with '%' are jupyter "magics"
# this magic allows us to display figures inline in the notebook
%matplotlib inline
```

[11]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

x = np.random.random(20)
y = np.random.random(20)

plt.scatter(x, y)
```

[11]:
```
<matplotlib.collections.PathCollection at 0x7f8bfdab10b8>
```



Running Jupyter on a remote server

On the remote host: start a Jupyter server

```
$ jupyter notebook --no-browser --port=$port
```

On your local machine: forward local connections to the remote port

```
$ ssh -N -f -L \
    localhost:${port}:localhost:${port} \
    ${user}@${hostaddr}
```

then simply visit `localhost:${port}` in your web browser

Publishing notebooks on Discovery pages

First convert your notebook to html

```
$ jupyter nbconvert --to html path/to/notebook.ipynb
```

Then copy over to the pages server

```
$ rsync -avz \
    path/to/notebook.html \
    ${user}@pages.discovery.wisc.edu:~/public_html/${dest}/
```

Snakemake – conceptual overview

- “Make-like"
  - Like make, snakemake seeks to build a “target” file(s)
  - Workflows consist of rules, which specify necessary inputs and how to produce outputs
  - Dependencies are implicit and recursively evaluated
- Why snakemake?
  - Exceptionally friendly and helpful developer (Johannes Köster)
  - Great docs: https://snakemake.readthedocs.io/en/stable/
  - Easy to pick up, great for managing combinatorially large analyses
  - Also keeps simple analyses reproducible - replaces bash commenting

This tutorial

- Goals:
  - Understand what a Snakefile looks like and the basic components
  - Understand how to run Snakemake
  - Get a sense for its capabilities and how you might use it in practice (and hopefully get a little excited about trying it!)
- Not goals:
  - Thorough understanding of all Snakemake features (there are many!)

What does a rule look like?

```
rule sort_file:
    input:
        'data/sample1.unsorted.txt'
    output:
        'data/sample1.sorted.txt'
    shell:
        """
        sort -k1,1 {input} > {output}
        """
```

Wildcards – generalize a rule

```
rule sort_file:
    input:
        'data/{sample}.unsorted.txt'
    output:
        'data/{sample}.sorted.txt'
    shell:
        """
        sort -k1,1 {input} > {output}
        """
```

How to run the same rule on multiple targets?

```
rule all:
    input:
        expand('data/{sample}.sorted.txt',
               sample=["s1", "s2", "s3"])

rule sort_file:
    input:
        'data/{sample}.unsorted.txt'
    output:
        'data/{sample}.sorted.txt'
    shell:
        """
        sort -k1,1 {input} > {output}
        """
```

Named inputs/outputs

```
rule sort_file:
    input:
        unsorted='data/{sample}.unsorted.txt'
    output:
        sorted='data/{sample}.sorted.txt'
    shell:
        """
        sort -k1,1 {input.unsorted} > {output.sorted}
        """
```

Multiple inputs

```
rule merge_samples:
    input:
        s1='data/s1.sorted.txt',
        s2='data/s2.sorted.txt',
        s3='data/s3.sorted.txt'
    output:
        merged='data/merged_samples.sorted.txt'
    shell:
        """
        cat {input.s1} {input.s2} {input.s3} \
          | sort -k1,1 > {output}
        """
```

Multiple inputs – equivalently

```
samples = ['s1', 's2', 's3']

rule merge_samples:
    input:
        expand('data/{sample}.sorted.txt',
               sample=samples)
    output:
        merged='data/merged_samples.sorted.txt'
    shell:
        """
        cat {input} | sort -k1,1 > {output}
        """
```

Multiple outputs

```
rule sort_file:
    input:
        unsorted='data/{sample}.unsorted.txt'
    output:
        sorted='data/{sample}.sorted.txt',
        top_50='data/{sample}.top_50.txt'
    shell:
        """
        sort -k1,1 {input.unsorted} \
          > {output.sorted};
        head -n50 {output.sorted} \
          > {output.top_50}
        """
```

Non-I/O parameters

```
rule sort_file:
    input:
        unsorted='data/{sample}.unsorted.txt'
    output:
        top_edges='data/{sample}.top_50.txt'
    params:
        top_n=50
    shell:
        """
        sort -k1,1 {input.sorted} \
          | head -n {params.top_n} \
          > {output.top_edges}
        """
```

Wildcards as shell arguments

```
rule sort_file:
    input:
        unsorted='data/{sample}.unsorted.txt'
    output:
        top_edges='data/{sample}.{top_n}.txt'
    shell:
        """
        sort -k1,1 {input.sorted} \
          | head -n {wildcards.top_n} \
          > {output.top_edges}
        """
```

```
rule all:
    input:
        expand('data/{sample}.{top_n}.txt',
               sample=["s1", "s2", "s3"],
               top_n=[50, 100, 150])

rule sort_file:
    input:
        unsorted='data/{sample}.unsorted.txt'
    output:
        top_edges='data/{sample}.{top_n}.txt'
    shell:
        """
        sort -k1,1 {input.sorted} \
          | head -n {wildcards.top_n} \
          > {output.top_edges}
        """
```

Configfiles

```
$ cat config.yaml
samples:
  - s1
  - s2
  - s3
```

```
configfile: 'config.yaml'

rule all:
    input:
        expand('data/{sample}.sorted.txt',
               sample=config['samples'])
```

Python instead of shell

```
rule sort_file:
    input:
        unsorted='data/{sample}.unsorted.txt'
    output:
        top_edges='data/{sample}.{top_n}.txt'
    run:
        import pandas as pd
        df = pd.read_table(input.unsorted, names=['c1', 'c2'])
        df = df.sort_values('c1', ascending=False)\
               .head(wildcards.top_n)
        df.to_csv(output.top_edges, index=False, sep='\t')
```

Input functions

```
def top_edges(wildcards):
    if wildcards.sample == 's1':
        fpath = 'data/{sample}.50.txt'
    else:
        fpath = 'data/{sample}.100.txt'

    return fpath.format(**wildcards)

rule analyze_top_edges:
    input:
        top_edges
    output:
        ...
```

Temp files

```
rule sort_file:
    input:
        infile='data/unsorted.txt'
    output:
        temp('data/sorted.txt')
    shell:
        """
        sort -k1,1 {input} > {output}
        """
```

Output file will only be kept until all rules which require it complete

Directory outputs

```
rule sort_file:
    input:
        infile='data/{sample}.txt'
    output:
        directory('results/genome_analyzer/{sample}/')
    shell:
        """
        ./bin/genome_analyzer {input} -o {output}
        """
```

Can handle programs which create a directory as output instead of a single file

Built-in conda support

```
rule sort_file:
    input:
        infile='data/unsorted.txt'
    output:
        temp(‘data/sorted.txt')
    conda:
        envs/my_env.yaml
    shell:
        """
        sort -k1,1 {input} > {output}
        """
```

Command line interface

- If your snakefile is called “Snakefile”:
  ```
  $ snakemake
  ```
- If your snakefile is called anything else:
  ```
  $ snakemake -s path/to/workflow.snake
  ```
- Snakemake will build any targets (and dependencies) for rule all
- Snakemake is smart — will only build files that don’t already exist

Command line interface — helpful flags

`-np` : dry run, and print commands that will be run for each rule
(helpful to check that wildcards are expanded as expected)

`-F` : rerun everything

`-R` [rulename|filename] : rerun rule/remake target and everything that depends on it

`-j $n_jobs` : run multiple jobs in parallel

`--use-conda` : if a rule has an associated conda env, use it

`--config` or `--configfile` : override config file/variable

`--keep-going` : if a job fails, complete as many unrelated jobs as possible

“Locking” the directory

- By default, snakemake “locks” all target files when running a workflow
- You can run more than one snakemake workflow at the same time, but ONLY if they have no targets in common
- Sometimes a snakemake run may be interrupted (server reboot, etc). Unlock with `snakemake --unlock`