Tips & Tricks for Working with HPC Clusters

ATI-Roche Tech Talk

Svetlana Lyalina

March 26th, 2024

Topics

1. SSH config, bash profile
2. Array jobs
3. Declaring and checking resource usage
4. Screen/tmux
5. Package management without admin rights (conda / mamba)
6. Being cognizant of file I/O load and speed
7. Using Linux utils (grep, sed, find) for common tasks
8. Open discussion

Spend less time logging in to the cluster

- Set up passwordless ssh
  - Also handy for copying data via `rsync`/`scp`
  - May need to add this setting to ~/.ssh/config:

```
Host *
  IdentityFile ~/.ssh/id_rsa
```

- Make aliases for commonly used login targets:

```
Host HPC
  User my_user_name
  Hostname server.myuni.edu
```

Now you can do `ssh HPC` instead of `ssh my_user_name@server.myuni.edu`

- Change ServerAliveInterval for spotty connections

```
Host *
  ServerAliveInterval 15
  ServerAliveCountMax 3
```

On the topic of configs: .bashrc/.bash_profile

Aside from setting important variables like PATH and LIBRARY_PATH that are necessarily for finding executables and linking libraries, there are few small tweaks that can make your life easier:

alias ls='ls --color=auto'

alias grep='grep --color=auto'

alias rm='rm -i' (could also set up a “trash” directory that files get relocated to)

alias mv='mv -i'

set **$EDITOR** to something you can tolerate

Remove limits on bash history

Avoid generating thousands of job scripts. Use array jobs

- Useful for “embarrassingly parallel” jobs (e.g. bootstrap)
- Still need to make sure each individual task is not too small so the scheduler doesn’t waste time on repeated setup/teardown
  - Example: One iteration of bootstrap takes something like 30s to run, so batch ~200 of them into one task
  - Use an auxillary text file with iterators and the prepopulated task id environment variable (e.g $LSB_JOBINDEX, $SLURM_ARRAY_TASK_ID) to set up the "inner loop"
  - Can also use that inner iterator to set RNG seed
- If there is complex orchestration and interdependence between tasks, consider a more full-featured workflow language/system:
  - snakemake
  - Nextflow
  - Airflow

Be aware of issues arising from parallel execution

- Not specific to HPC, but overall something to be aware of:
  - Race conditions
  - Overwriting outputs
  - Deadlock
- Accidentally “doubling up” on multicore/multithread operations:
  - Some R setups will automatically tell the linear algebra libraries (BLAS, LAPACK) that they can use all the cores they can detect
  - If you then also parallelize in the outer loop, your load will be much higher than you requested (or is optimal)
  - Usually this can be fixed by setting some environment variables,
  - e.g. OPENBLAS_NUM_THREADS = 1, MKL_NUM_THREADS = 1

Run a small-scale pilot on an interactive node

- Before launching your whole array of tasks, test out one “hands on” to get an idea of its requirements
- On an interactive node, use a `screen` or `tmux` session to launch your test code and then detach
- Monitor the resource usage with whatever is available. Usually `htop` is available but not guaranteed, so may have to fall back to regular `top`.
- Note the approximate peak RAM usage and CPU usage. If this is a GPU-enabled job then also check VRAM usage with nvidia-smi
- Use these values for resource requests for your final job launch script
- Beware of confusing memory request flags: --mem-per-cpu vs --mem
- If there's no interactive node and you can't ssh to the worker nodes, try adding a call to the requisite job stats function at the end of your job script (LSF: `bjobs -l $LSB_JOBID`; SLURM: `sstat -l $SLURM_JOB_ID`) and then examining the output logs.

Gather information on your jobs to ease troubleshooting

- Some of this is tracked automatically by the scheduler, but accessing that data (through utilities like `sacct`) can be cumbersome, especially after the job is finished/crashed.
- Your institution might have a user-friendly interface for job tracking and overall cluster telemetry. A popular one is Grafana.
- Set up your own lightweight logging at a few points in your pipeline to more easily track progress and get more information on errors
- While printing to stdout is fine when you don’t plan to reuse or share this particular piece of code, try to put in the effort to have toggle-able logging and output to stderr. You'll likely thank yourself later.
- Don't reinvent the wheel, use a logging package (builtin `logging` in python, `logger` in R requires install)

Consider running a profiler before scaling up

- Tools are available in every language to track where your code is spending the most time
- cProfile in python, lineprof in R, valgrind/callgrind in C
- However, keep this classic Knuth quote in mind:

> “The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.”

- So don't get to absorbed by the process and just look for blatant ineffciencies and quick wins

Know where your data lives in relation to the compute

- Most clusters have a network file system which gets mounted by the worker nodes and usually contains a user's home directory
- This usually has a small space quota and is meant for code and libraries. Data are stored on a slower network mount.
- The worker nodes typically have a local scratch space that is optimal for read/write operations specific to the individual task
- A common paradigm is to copy over relevant data to the local scratch space at the beginning of the job, work on it locally, and then copy the final result back at the end
- Doing continuous short reads/writes with NFS can lead to artificially slow runtimes or destabilizing the whole cluster (uncommon, but can happen)

No sudo? No problem (somewhat)

- Anything involving drivers or networking you'll probably need to get an admin involved
- For other things you can actually squeak by with virtual environments through conda/mamba:
  - Get a more up to date version of R/python
  - Get compile tools (gcc, cmake, gfortran)
  - Get some Linux utils that may be missing (htop)
  - Binary executables/libraries that R packages require (GDL, zlib, openssl)
- Another neat feature of conda is that there's version control for the environments so you can usually roll back when you've broken it

```
conda list --revisions
conda install --revision N
```

Shell one-liners for common tasks

Use a combination of `find` and `xargs` to process similarly named files in parallel. Here's an example with a directory of paired .fastq files:

```
find ./sequences/ -name "*R1.fastq" | \
xargs -P 8 -I {} sh -c 'sample=`basename {} R1.fastq`; \
R1=sequences/${sample}R1.fastq; \
R2=sequences/${sample}R2.fastq; \
bowtie2 -x hg38 -1 $R1 -2 $R2 > ${sample}.bam'
```

This is a hacky way to do parallelism and you might consider using GNU parallel instead, which is more full featured

Use `sed` to switch from comma-separated values to tab-separated:

```
sed 's/,/\t/g' myfile.csv > myfile.tsv
# If you're confident about your substitution you can do it in place
sed -i 's/,/\t/g' myfile.txt
# Or tell sed to make a backup
sed -i .bak 's/,/\t/g' myfile.txt
# If you somehow got a file with carriage returns
sed -i 's/\r$//g' myfile.txt
# If your genomic intervals file only has numbers for chromosome names
sed -i 's/^/chr/g' intervals.bed
```

Use `cut`, `sort` and `uniq` to quickly tabulate the values of a column:

```
# Don't forget the sort! Otherwise uniq will get it wrong
cut -d , -f 2 values.txt | sort | uniq -c
```

Use `comm` to find entries shared between two files (or unique to each of them):

```
# Get only shared
comm -12 expected_entries.txt existing_entries.txt
# Use the flags -2 and -3 to not print entries unique to 2nd file and shared
comm -23 expected_entries.txt existing_entries.txt
```

Use `grep` to search through error logs:

```
# Use -A and -B to show lines before and after the match
grep -A 5 -B 5 "error" logfile.txt
# If you want to extract all values preceded by some keyword
grep -Po "(?<=Iteration failed:).*" logfile.txt
# Note: -P option not available on default OSX grep
```

Check integrity of some files that your received (assuming sender also provided checksums):

```
# Sender needs to generate the checksums file
md5sum file1.txt file2.txt file3.txt > checksums.txt
# Then on the receiving end, in the directory of the transferred files,
# check against the provided file
md5sum --check checksums.txt
```

Use `cat -v` to check for non-printable characters that may trip up other programs:

- See this stackoverflow post for an extended demonstration
- Most of the time we just want to know if the file uses spaces or tabs as delimiters

Open discussion:

- Do you run into any frequent problems when using the cluster?
- Have you found any "quality of life" improvements you'd like to share?