Intro to Supercomputing - Wrap-up

Ramses van Zon

July 9, 2021

What are we doing today?

- Review and give answer of the assignment.
- Go over some of the more common mistakes.
- General best practices in scientific computing.

Worried that your answer wasn't right or your assignment not complete? If your submission on the site shows you worked with the material, that is all that we really ask for.

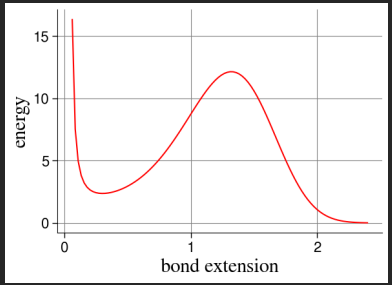
If you haven't submitted anything yet: you have until the end of next week.

That being said: many of you got this perfectly!

Assignment

What was the task again?

- The script `sweep_bondbreak.sh` executes 96 repeats of the computation of the bond breakage time, one by one.
- These could all run in parallel.
- How long did this script take?
- Use GNU Parallel in a modified version of `sweep_bondbreak.sh` to parallelize the computation using 16 cores on a single compute node of the Teach cluster.
- Submit this new script to the scheduler. How long did the new script take?



(An) Answer

Initial script

```
#!/bin/bash
#SBATCH --ntasks=1
#SBATCH --time=00:20:00

module load python

# temperature value
T=2.2

# Run multiple cases with different random seeds
for S in {1..96} ; do
 ./bondbreak --temp $T --seed $S                    \
 --filename out/$T-$S.dat --logfile out/$T-$S.log
done

# Extract the breakage times from the logs
awk '/BREAKAGE DETECTED/{print $8}' out/$T-*.log
```

Wall-clock time (seff): ~15 minutes

Parallel script

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=16
#SBATCH --time=00:20:00
module load python gnu-parallel

# temperature value
T=2.2

# Run multiple cases with different random seeds
parallel --joblog log.txt                           \
 ./bondbreak --temp $T --seed $S                    \
 --filename out/$T-{}.dat --logfile out/$T-{}.log\
  ::: {1..96}
*
# Extract the breakage times from the logs
awk '/BREAKAGE DETECTED/{print $8}' out/$T-*.log
```

1m10s: about 12x speed up!

Common issues

Submission gripes

- Do not forget to submit the *job script* to the Dropbox for the parallel computation!
- Do not forget to submit the slurm output file(s) to the Dropbox for the parallel computation!
- Do not submit screenshots, rather cut and paste, and save to a plain text file.
- Rich Text Format 'rtf' is not a plain text format.
- Doc(x) is not a plain text format.
- Ideally, mention the speedup you got.

No magic

- Just prepending `parallel` to the command is not enough.
- Just using `--ntasks-per-node=16` with no other modifications does parallelize the script. Actually, it just leaves 15 or the 16 cores idle.
- Conversely, though, leaving `--ntasks=1` kills the paralellization.
  (`--ntasks` takes precedent over `--ntasks-per-node`)
- Note that output can be in any order, as things run in parallel.
- Executing scripts on the command line makes them run on the login node.
  Not what you want!
- Syntax matters:
  - `--nodes` != `--ntasks`: leave out ntasks when using ntasks-per-node
  - `--nodes=1` is required to guarantee all tasks to run on the same node.
  - `:::` is a command line argument, and must be separated from what follows by one or more spaces.
  - bash syntax is more peculiar than it seems

Useful tricks

- `--joblog` times the individual jobs

  Also useful for restarts should the job run out of time.
- Timing information of the whole script: `seff`
- Add an email address to see that your job has finished:

```
#SBATCH --mail-user=thisisnot@myemail.ca
#SBATCH --mail-type=FAIL,END
```

- Make your job script abort as soon as there is any failure or undefined variable name error:

```
set -eu  # add near top of script
```

Do not trust

- Beware of copy-paste: Don't include lines you do not understand.
- E.g. `module load intel/2018.4 gsl/2.4` or `module load gcc/7.3.0`, etc. may not hurt here, but it might in other context, and is not needed. Load only the modules you need.
- Related: Be wary of recipes you find online. Even if they are for Linux, and all supercomputers run Linux, they are often written for a machine that you own and have exclusive administrative rights to.

Variables and quotes

1. Everything in bash is a string.
2. Spaces separate command line arguments, unless they are inside `"..."` or `'...'`.
3. Variables substitutions are triggered by a dollar sign.
4. *Except* when you *escape* the substitutions with single quotes (or a `\$`).
5. Double quotes do not escape substitutions.
6. To nest same-style quotes, you must used `\"` or `\'`.
7. Ranges are expanded before variables.
8. There are two kinds of variables: exported ones and non-exported ones.

   The latter are the default and are not inherited by sub-processes.

   Use single quotes sparingly, and only to explicitly stop variable and filename expansion.
   Use double quotes if there are (or if there is any chance that there might be) spaces or other special characters in an expression.

Questions?

Best practices in scientific computing

Code never dies. . .

- There is no such thing as one-off codes or scripts. If it was worth writing, it's worth running again, and someone (could be you) could use it.
- If it's a big, ununderstandable mess, no one want to touch it and all the effort will need to be repeated. Plus, do you really trust its results if you do not understand it?
- Professional software developers deal with this all the time, but computational scientists are focuses on results.
- Putting in a bit of effort in writing maintainable, reproducible code, however, can pay off big in future projects.
- So let's consider some common, useful best practices in scientific computing.

Automate everything

- Write scripts for any computation or data processing.
- This means you have a precise record, you can rerun them, and you can adapt them.
- If GUIs are part of your workflow, try to replace it with commands.

  GUIs are particularly awkward for record keeping: instead of a script, you would need to keep a written log of every mouse movement, click and keystroke.
- Use makefiles or cmake for automating building software.

Use version control

- Version Control is a tool for managing changes in a set of files.
- Keeps historical versions for easy tracking.
- It essentially takes a snapshot of the files (code) at a given moment in time.

Why use it?

- Makes collaborating on code easier/possible/less violent.
- Helps you stay organized.
- Allows you to track changes in the code.
- Allows reproducibility in the code.
- And when something goes wrong, you can back up to the last working version.

Comment and document everything

SciNet

You must document your code. Must. Not optional. Documentation of code comes in many forms:

- sensible variable, function, class, and module names.
- comments in the code.
- help commands, doc-strings, or other built-in feedback.

But why?

Six months from now you're not going to remember the motivation for writing that function.
So write it down in the code somewhere.

Write a README or a full manual if you expect others to use the code without reading all of it.

Note: there is no such thing as self-documenting code.

Make your code and scripts portable

- Try it on a **different computer**.
- **Don't hard-code** paths, use variables like `$HOME`.
- Use `#!/usr/bin/env` in **shebangs** for scripts.
  (and always use shebangs!)
- Make as few assumption about the (super)computer the code will run on as possible.
  Add an **explicit requirements** file with your code.
- Stick to programming language **standards**.
  This avoid the "but it works on my computer" bug.

Modularize

- Scientific software can be large, complex and subtle.
- If each section of code uses the internal details of other sections you must understand the entire code at once to understand what the code in a particular section is doing.
- This makes finding bugs extremely difficult.
- It also makes writing testing routines for your code extremely difficult.

Modules should:

- Perform a single, specific task.
- be separated into files each of which contain only related functionality.
- be written so as to enforce boundaries between sections of code.
- include testing routines, that check whether known cases work.

Checkpoint

- Things can go wrong while your job is running.
  - Could be due to your job (e.g. takes too long or too much memory and crashes)
  - Could be due to your account (e.g. out of space).
  - Could be due to the system (e.g. file system issues)
  - Could be due to the external infrastructure (e.g. power outage)
- To make sure not to have wasted computation time in these cases, you **checkpoint**
- Checkpointing is writing out enough information about the state of the system to allow the computation to restart at the checkpoint.
- Checkpoint could be implemented in the operating system and scheduler, but this is so inefficient that is rarely is.
- So your application needs to include a feature to just write and read the checkpoint data.
- GNU Parallel has this feature, using `--joblog` and `--resume`.
  It won't checkpoint within sub-jobs, though.

Optimize I/O

- $HOME, $SCRATCH, and $PROJECT all use the parallel file system (GPFS).
- Your files can be seen on all login and compute nodes.
- These are high-performance file systems which provides rapid reads and writes to large data sets in parallel from many nodes.
- But accessing data sets which consist of many, small files leads to poor performance.

Do's and don'ts

- Avoid reading and writing lots of small amounts of data to disk.
- Many small files on the system would waste space and would be slower to access, read and write.
- Write numerical data out in binary. Faster and takes less space.
- Some file systems are better than others I/O heavy jobs and checkpoints. E.g., Niagara has a Burst Buffer, some systems have node-local storage.
- When your files fit in memory, use ramdisk, which lives in memory.

Thanks for attending.

I hope we've enlightened you about the business of supercomputing.

Do not forget:

- Take the attendance test.
- Submit your assignment if you haven't yet.
- Take the anonymous evaluation survey.
  Same place as the attendance tests.
  Your feedback will be much appreciated.

Happy computing!