Checkpointing

Note: the code here is not terribly Pythonic because what we are talking about here isn't specific to Python. We do use Python's pickle system to output our objects but we don't want to tie ourselves too closely to Python since the general principles apply to any language

- 01 Example Checkpointing Code

  - We've provided the simple random number based Pi calculator from the video as an example
  - It includes serial versions, versions that restart incorrectly and versions that restart either exactly or statistically correctly. Rather than list them here, there is a Readme.md file in their directory.
    - Check that you can run them and that you understand *why* the ones that don't work don't work
    - Think about the statistically correct restart
      - For what sort of problems might this give you good enough results to be useful? What sorts need a "more exact" restart than this?
      - What impacts would this sort of statistical restart have on questions of reproducibility ?

- 02 Your Own Libraries

  - Think about the libraries that you use
    - Have a look in their documentation for restart, checkpoint or serialisation routines (serialisation is the process of converting an object into a data stream for e.g. writing to disk.)
    - How would you use them?
    - If they don't have any explicit checkpointing then do they produce objects that you can pickle for output (if you're using Python), or can you pack up the data yourself?
  - Think about the programs you use the libraries in

- What data is needed for a complete restart?
- Do you need "exact" restarting, or is statistically the same sufficient?
- Some other things to think carefully about:
  - Numerical precision - make sure you output numbers with enough digits to get the right answer on restart
    - (Advanced issue) is your code relying on extended-precision registers or similar intermediate results? This can be very difficult to handle (custom written Assembly code in some cases)!
  - Ordering - do you rely on getting results in some particular order? If so, you need to think about preserving whatever dictates that order (if you can), or removing that reliance (usually a better approach)
  - Is there any hidden state - for instance, in the Pi program, the internal state of the random-number generator is a bit hidden and you might not think to checkpoint it
  - Is restarting actually worthwhile? You may find that you need far too much data, or get sub-optimal results from restarting. In these cases, think carefully about whether this is a good approach at all

- 03 Philosophy of Checkpointing

- For your own code/codes, how important is:
  - Checkpointing against wall time limits?
    - Does your code take anywhere near 24 hours to run? If so then there will be some clusters where an unusually long run will hit the wall time
    - Is there any chance that your code will grow until the runtime gets anywhere close to 24 hours? It's generally easier to put checkpointing in when a code is small and grow it as the code grows than to add it later
  - Checkpointing against node failures?
    - How many nodes are you using for a single job? Even if you are on a really reliable system each node has a MTBF of about 5 years. That means if you're using about 1200 or more nodes you've got a roughly 50/50 chance of one node failing within a day. Real world clusters may be less reliable than this

- For your own code, what variables do you need to output? What variables contain the entire state of your program
  - Remember to include your own variables as well as hidden variables inside libraries
  - Are your variables well connected together in a logical fashion? Could you make your life easier by combining some variables in types, structs or classes?
  - What state is essential and what is merely expensive to recalculate? How much disk space (and hence writing time) would you need to give up for the expensive to recalculate variables?

- 04 Job dependencies
  - Go onto whatever cluster you've got access to
  - What job submission script system does it use?
    - If SLURM then you can use the instructions directly from our video
    - If not then you can find descriptions of how to set up Job Dependencies in the user manual for the system you are using (for example, Section 7.2 in the manual for PBS Pro (as used on the previous UK Tier 1 ARCHER system) shows how to set up Job Dependencies)
  - Try running jobs where one depends on the other. At the moment, don't actually do any work, just write two simple programs - the first writes a file and the second reads it. Satisfy yourself that the second job doesn't run until the first has completed, and that when it does it runs correctly
  - Does your code need any changes to work like this? How does the second job know that it should restart from a checkpoint rather than start afresh?
    - If you're going to use job dependencies this should be automatic
    - There's a few common ways to signal that a job should restart
      - A "sentinel" file - written just before the first program exits, this can be an empty file that simply signifies the job ran, or contain some data on what was done. Think carefully about how these should be cleaned up afterwards
      - Check the output data - for continuous jobs, you want to pick up where the first left off, so it's easiest just to check how far along the output has got