Big Computing Challenges
Part 2: Repeating Yourself

"The Angry Penguin", used under creative commons licence from Swantje Hess and Jannis Pohlmann.

December 2022

Warwick RSE

Robustness

Failure

- You might think big computers are LESS likely to fail
  - Sadly, statistics disagree
- Approximately one node failure per half million node hours, i.e. one per 10-100 million processor hours
- Million core-hour jobs, 1 in 10-100 might fail
- Run 100 x 100k core-hour jobs: expect one to fail
- Run 1000 core-hour jobs: never expect failure

Statistically we should probably consider failures as independent events. In reality this is often not the case - physical failures can be due to bad components, and software failures can occur due to bad updates, so failures are often correlated. Failures of the interconnects (network) also tend to affect more than one node. This makes these figures very hard to properly ascertain, so we don't strictly expect N failures per M hours or anything, but this is the right ballpark.

What this figure means for you is, unless you're hitting millions of core hours, you can proceed as though cluster runs "never fail" in the sense that you simply repeat things in those rare cases. DO NOT act as though they never fail in the sense of allowing failure to destroy vital data! Suppose you were recolouring images: always keep the initial file until you have verified the final result, otherwise you might lose that image entirely if things go wrong. But you would never be this careless anyway, right?

Robustness

- Robust, definition: strong and unlikely to break or fail
- If failure is inevitable, how can anything be robust?
- The answer is simple:
  - Nodes might fail. Disks might fail. Jobs might fail.
  - You should not!
- Robustness requires tolerance of failure in part so that the whole succeeds

Robustness often means a certain amount of redundancy - for instance duplicating data to ensure it cannot be lost. How much caution is needed depends on the type of thing you're working on. Aeroplane control software has to be extremely robust, often including multiple redundant sensors, redundant processing chains etc. If your project is safety critical, handles personal data, or otherwise requires management of risk you should have performed risk assessments and a whole load of things we are not even considering here. Oh, and your supervisor/line manager should have been through this already!

Do Repeat Yourself

- Simplest (partial) solution to making stuff robust:
  - "Repeat until done"
    - If a job fails, run it again
- Do not make Fake Einstein despair
  - If a job is actually failing, i.e. crashing
    - Do not expect doing the same thing to give a different result….

If your job is hit by a node failure, just run it again. If it is crashing or otherwise misbehaving, try and find out why and try and fix it! If it runs sometimes, then be very very suspicious about the result it gives! Crashing is unexpected, so why would you expect the rest of the code to be working as you expect either?

Reliability

- By the way, it might sound like cluster machines are not **reliable**
  - Not so!
- Critical systems aim for 5 nines or better
  - 99.999% uptime - 5 minutes downtime per year
- Research grade stuff reaches maybe 3 nines
- Your codes should be robust enough to handle this!

Usually this means unplanned downtime - failures and things. Scheduled maintenance etc might be separate, or might be designed to ensure systems keep running but slower or with restricted function, which would usually be omitted from the downtime figure.
We have a rough figure of 99.7% for university stuff. This is really very reliable for all normal purposes. It is when your needs get a little unusual that it can affect you, and needing truly Big Computing is generally getting there.

Carrying On Regardless

- A node failure is cessation of useful work; need not lead to loss of everything up to that point
  - Simple continuation - for sequential stuff, just carry on from last successful point
    - Might need metadata or operational stuff to know
  - Checkpointing - storing state so you can continue the job from where it left off (or as close as feasible)
    - Aim for identical result to non-failed case?

Continuation - e.g. if processing a list of items, flag once an item has been dealt with. Continue from last flagged point - lose at most part of processing for item in flight on failure.
Checkpointing - for codes with substantial internal state e.g. evolving a system. Have to start up and probably want IDENTICAL result as the case where nothing happened

Padding your Requests

- When assessing your computational needs you MUST consider possibility of failure
  - Bare minimum time should include factor for this
- HINT - “Checkpointing is hard” is a BAD reason for slack time
- HINT - “Your machine keeps failing” won’t get you any sympathy
- HINT - “Sometimes I run the wrong job by accident” won’t either
- Justify your needs **and** your buffer factor!

Everybody knows nodes fail and jobs get misconfigured and everything else that means you end up needing more time than you strictly… should
Say things like “a small factor to ensure the work can be carried out effectively” or “to account for variations in time taken for different input sets” WE ARE NOT TELLING YOU TO LIE. But the trick is not to “say the quiet part out loud” and insult the person assessing your request

Errors and Uncertainties

Error Bounds

- Robustness in research terms means a lot more than being able to churn out an answer
- The answer also has to be useful for something
  - You have to understand what results really MEAN
- If I run some code and get an answer of “ten”, can I trust this?
  - Would I get the same answer next time?
  - Is ten a better answer than eleven, or nine?

Error Bounds

- Moreover codes often give very precise looking answers e.g. 10.000000001
- But this might actually only mean “closer to 10 than 11”
- Have to consider precision of what you put in and at least estimate whether you can say 10.0 exactly or even 10.00 or whatever

Usually if you put in numbers to 1 significant figure you expect to get out about the same. But what if a number gets run through some complicated function? See a few slides later for one way this gets handled

Error Bounds

- Fairly common solution to quantifying error is ensemble jobs
  - Run the same problem several times
- Ensembles can answer both of our questions
  - Run the same problem with changes to some source of randomness
    - Will we always get “10” for this problem?
  - Run slight variations on inputs
    - How different do inputs need to be for the answer to be “11”?

If you are just starting to need Big Compute to address a problem, you may well also be starting to run into this sort of problem too. In particular, you might have been able to simple run everything you needed, several times, and see how much the answers might vary. But if you now need Big Resources you might want to step back and consider more formal treatment of things to get the best result from the least effort.

On the other side of the coin, it might be the need for ensembles and things that has provoked your need for more resources. We’ve already talked about some of the challenges and solutions for running many jobs, and later we’ll discuss some new stuff about the challenges on the output side too

Uncertainty Quantification

- For some domains, we need to know precisely how sure we are of our answers, i.e. its uncertainty
  - Not enough to just run a few variations
- An entire field of study exists to deal with properly quantifying the uncertainty of answers, UQ
  - Often means assessing uncertainty on inputs and propagating this through entire calculations
- Not easy, but if necessary, has to be done

One in a Million

- Uncertainty/error is inherent in many problems
  - Randomness might appear to show patterns
  - Extreme values can occur
- Uncertainty is not the same as mistakes!
  - But mistakes can occur too and small chances might matter once you run many or large jobs
- Are these risks something you need to handle?

Redundancy

- What about answers you don't need after all?
  - They certainly don't count for much!
- Suppose you run problem X and see immediately that problem Y is useless
  - Baking a cake for 1 hour is too long, so 1.5 hours isn't worth trying
- Exploiting scheduler or job manager to manage "dependent chains" of jobs is important

See Part 1 this morning for more about dependent jobs

Ground Up Thinking

- Finally, you can't build a skyscraper in a canoe
- If your libraries aren't robust, how can your code be?
- The libraries and techniques you build on need to be sufficiently robust
  - This varies by domain, like everything else
  - Cosmetically recolouring an image is a very different task to controlling collision detection in aeroplanes

Rule of Thumb?

- How to spot a safe piece of code?
  - Comes from a reliable source
  - Has a name, documentation etc
- DON'T trust rules of thumb if it matters!
  - You might have to test, profile and otherwise VERIFY a library before you rely on it

It is important to think about the provenance of your libraries. One tends to assume that libraries are written by proper development teams with proper processes for testing and validation, but many smaller libraries are maintained by single researchers primarily to do their own work and they will have all of the good and bad properties of any other piece of academic code

Reproducibility

Reproducibility

- Being able to generate answers reliably and efficiently is step one of Getting Things Done
  - Presumably you want to use your results to **prove** something
- If you can't even prove the answer correct, what can you prove using it?
  - If you can't get the same answer again, does it mean anything at all?

Reproducing your inputs

- Answers need to have provable provenance to be trustworthy
- Meta-data: data about data
  - How it was generated?
  - What the input parameters were?
  - What code created it?

Meta data can be a lot of things, these are a few we think are usually relevant. There might be a lot more!

Useful but (hopefully) unimportant

- Some metadata indicates a PROBLEM with your reproducibility!
  - Machine you ran on?
  - Information about parallelism details (core count etc)?
  - Some version info
    - Bugs or interface changes are one thing
    - General purpose solutions should be general

Reproducible in result terms should not need somebody to run this precisely!

A library to solve an equation exactly should always give the same answer! Approximate solution must be good enough not to change the answer

Types of Reproducibility

- Haven't said what we mean by reproducible and it MATTERS
- Think back to sorting letters
  - Suppose two letters addressed to the same person
    - Expect them to be sorted into the same pile
    - Would you expect them to end in the same order if you ran the sorting process again?

If you're familiar with sorting algorithms, this is the question of whether a sort is stable - if two equal objects start in one order, will they ever be swapped? A stable sort is one where the answer is no, so the objects end up in the same order they start by guarantee. For fairly obvious reasons, swapping equal objects can be a lot of extra work, and means the final order of our items is allowed to vary, although it will always be "sorted".

Exact Reproduction

- Getting the exact same answer, i.e. even “equal” letters always end in the same order, is an exact reproduction
  - Analogous to bitwise reproduction of a result - e.g. a number the same bit pattern in memory, not just the same number to given precision
  - Classifier system giving the exact same tags to an input, not just valid tags

Result Reproduction

- Sorting the letters into the same pile is reproduction too
  - Like getting a result of 1.000,000,1 and 1.000,000,7 to 3 dp accuracy
  - Or classifier tagging something “correctly”
- This is often all we need!
- Problem is robust against small changes

No need to seek bitwise reproduction, especially with parallel code or with restarting a code, if we only need result reproduction, and this is often much much easier

External Dependencies

- Library compatibility - backwards, forwards, sideways
  - If your library breaks you can’t reproduce your result!
- Version numbers can be kept
  - Worry if version changes answer and difference is not a bug fix
    - See also next slide

It is very common for modern academic software to have surprisingly specific version requirements for their libraries. If this is because the specific versions are needed because they are the only ones that will compile then your libraries might be a maintenance and portability nightmare but they work OK. If different versions give different answers then you really want to know **why**! Most commonly it is a bug fix in the library and all answers before the fix are wrong and all answers after the fix are right. If different versions give different answers then you need to be sure that all of the answers that you can get from different versions are acceptable and don’t change the results of your research

Limitations

- Techniques with “caveats” - preserve and describe enough to do it again, the same
  - Worry about whether the answer is “correct” and robust if your technique is desperately important
- Also important if you are using any sort of “proxy” output, just like in an experiment

Proxy outputs are things you measure which you believe are related to what you really want. For instance, measuring the opacity of a liquid to work out the concentration of something dissolved in it. You need to keep confounding factors in mind and consider this too

Data Preservation

Preservation

- Reproducibility includes preserving how you produced data
- Rules out certain input strategies
  - Graphical setup interfaces are risky! Is a mouse-click reproducible?
  - Look for a way to export setup parameters if you must use a GUI

Essentially the core of preserving your ability to produce results is being able to run exactly the same program again. If your main program always takes a file that has to be generated by another script then the **script** becomes the thing that has to be preserved. What you want to be able to do is run the exact same problem again!

Standard Input Files

- Structured input files are great!
- Readable, comprehensive etc
  - JSON, Yaml etc
- Beware proprietary formats
  - You might not be able to read these files without proprietary program
    - What if program is discontinued, gets too expensive, etc?

Even open formats can be troublesome if they aren't popular. Even if you know exactly how to read a file, if the library that makes it easy to read no longer compiles in newer language versions (or even more likely doesn't work in newer versions of Python) then you would have to write a **lot** of extra code to still work with those files. Unless there is a file format that is ubiquitous in your field then a good rule of thumb is to use the simplest, most open format that works

Programs without Input Files

- What about codes that don't have rich input systems
  - Consider wrapping them in something else
    - Python program
    - Shell script
  - Can also add things at output stage
    - More about output problems later!

Controlling Versions

- Research codes are often in flux
  - Adding features
  - Changing algorithms
  - Fixing bugs
- How to manage this?
- “Did I make this plot before or after I fixed that factor 2 I missed?”

Controlling Versions

- Look for existing solution:
  - Version control systems
    - System to manage changes to files, record their history
    - One man's old is another man's established
      - Can be archaic in places

Controlling Versions

- Most Version Control systems handle some things better than others
  - Excel spreadsheets notoriously hard for example
- Version Control will record the history of any text file but it can do a lot more than just that so the more that it understands the more value you can get from your version history

Research Councils

- "Professional pride" means doing good, reliable, reproducible work
- BUT… there's also rules to be aware of
  - Research councils increasingly demanding reproduction
    - Codes and data made available
  - Exact rules are different for different research councils so check what they are
  - You will have to keep your data available for between 3 years after the end of the grant (Art and Humanities Research Council) to 10 years after the last person asks for access (Engineering and Physical Sciences Research Council)

What to keep?

- If in doubt keep the raw data that you are generating
- If you can exactly reproduce your results then it **can** be acceptable to only keep enough data to permit that reproduction **if reproduction is less onerous than retention**
- It may be acceptable to only store processed data if the processed data is **all** of the data that is needed to produce the publications
- Check the exact requirements with your funding council and with the data management plan for the grant that you are on (there should be one somewhere)