# Big jobs/simulations

## Tools for Reproducible Research

### Karl Broman

Biostatistics & Medical Informatics, UW–Madison

kbroman.org
github.com/kbroman
@kwbroman
Course web: kbroman.org/Tools4RR

Reproducibility can be considerably harder for computational tasks that take more than just a couple of hours, and in some cases, the computations for a project may require years of CPU time (split across many computers).

The problems are: (a) it's hard for someone to re-do all of that work, and (b) large-scale calculations tend to be organized in a system-dependent way, so even if time weren't a factor, it'd be harder to transfer the calculations to another system.

Simulations have some special issues (e.g., saving the seeds for random number generators), and they are notoriously irreproducible.

We at least want to fully document the process: we want capture the exact code and workflow, so the results can be reproduced on the same system. And we want that code to be modular and readable, so that it could be restructured for a different system, if necessary.

I must admit that I don't always do this well. Part of this lecture is more of a sketch of what I think one should do rather than what I actually do.

# But first…

Suppose I've just written an R function and it seems to work, and suppose I noticed a simple way to speed it up.

What should I do first?

- ▶ Make it an R package
- ▶ Write a test or two
- ▶ Commit it to a git repository

My aim here is to reinforce the things we've been covering in the course.

Everything will be a lot easier if you put the code into an R package. For the minimal package, all you need are the `DESCRIPTION` and `NAMESPACE` files.

And before you start editing the code, you should write a small test. Then you'll have evidence that it currently works, and the tests will help show that it's still working after your modifications.

And before you start editing the code, `git commit` what you have so far! If it turns out that your new idea doesn't work, will you be able to get back to what you had originally?

# So what's the big deal?

- ▸ You don't want `knitr` running for a year.

- ▸ You don't want to re-run things if you don't have to.

It may not seem like "big jobs" are that big of a deal, but in my mind this is the only real difficulty in reproducible research.

Okay, there's also the difficulty that some data can't be generally distributed due to confidentiality issues.

But aside from subjects' confidentiality, the main problem is how to manage and capture the really long-running computational analyses where even on the same system it can be a gargantuan effort to reproduce the results.

# Unix basics

```
nice +19 R CMD BATCH input.R output.txt &
fg
ctrl-Z
bg
ps ux
top
kill
kill -9
pkill
```

Use `R CMD BATCH` to run an R job in the background.

Use `&` to put it in the background.

Use `nice +19` to give it low priority.

Use `fg` to bring a job back into the foreground.

Use `ctrl-Z` to suspend a current job; then use `bg` to put it in the background.

Use `ps ux` or `top` to view current jobs.

Use `kill` or `kill -9` with a process ID (`PID` in the output of `ps` and `top`) to kill a job. Use `pkill` to kill multiple jobs at once, using a pattern.

Note: In my experience, Windows sucks at managing multiple processes. Windows XP was not bad at this, but Windows XP is dead.

# Disk thrashing

In computer science, thrashing occurs when a computer's virtual memory subsystem is in a constant state of paging, rapidly exchanging data in memory for data on disk, to the exclusion of most application-level processing.

<div align="right">– Wikipedia</div>

A common problem is having multiple jobs on a machine attempt to use more than the available memory on the machine, so then the machine starts swapping data from RAM to disk, and all the jobs slow to a crawl.

If a machine starts disk thrashing, it can be hard to log on and kill the jobs.

The solution: anticipate (and then watch) memory usage.

Another thing I did: miscalculated the number of files to be produced by a job, by a couple of orders of magnitude. It turns out that if you go beyond some limit on the number of files in a directory, you can totally kill a storage system.

# Biggish jobs in knitr

- ▶ Manual caching

- ▶ Built-in `cache=TRUE`

- ▶ Split the work and write a `Makefile`

You can put big computations within an R Markdown file, but personally I don't want to wait more than a couple of minutes for it to compile. If it's going to take longer than that, I'll split things up.

And if you are going to have some large computations with knitr, you won't want to re-run all of them every time you make even the smallest change to the text!

That's where you want to cache some computations: save the results and just load the results rather than re-run the code. Unless the code changes, in which case you do want to re-run it (and any other code that may depend on the results).

# Manual caching

```
```{r a_code_chunk}
file <- "cache/myfile.RData"

if(file.exists(file)) {
  load(file)
} else{

  ....

  save(object1, object2, object3, file=file)
}
```
```

This is the "by hand" approach. If the file doesn't exist, run the relevant code and save the needed results to the file. If the file does exist, just load the file and skip the code.

If you want (or need) to re-run the code, you need to delete the file manually.

One issue: if you want the code to actually be shown, you need to repeat the code: in a chunk that is shown but isn't run, and then in this chunk that is run but isn't shown.

You need to be very careful about dependencies.

# Chunk references

```
```{r not_shown, eval=FALSE}
code_here <- 0
```

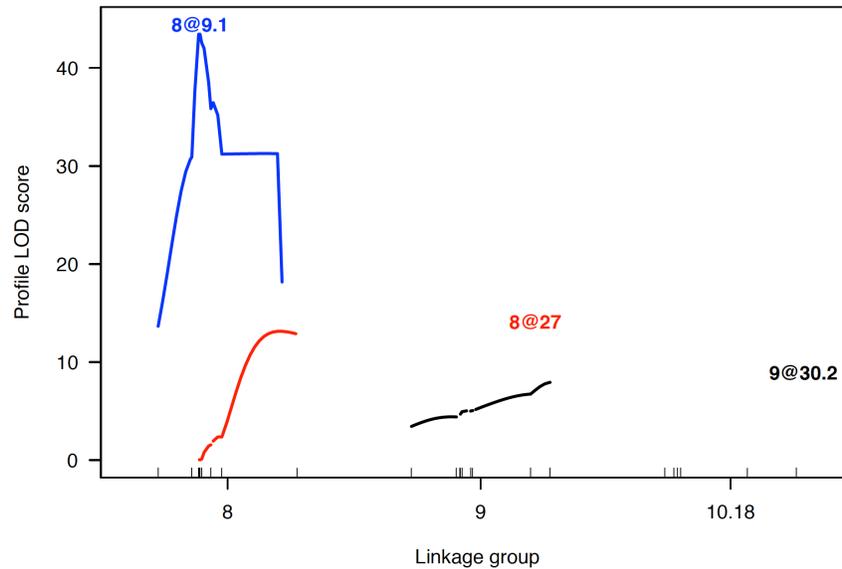```{r a_code_chunk, echo=FALSE}
file <- "cache/myfile.RData"

if(file.exists(file)) {
  load(file)
} else{
<<not_shown>>
  save(code_here, file=file)
}
```
```

Here's how I'd avoid repeated code: use chunk references.

The <<not_shown>> is replaced by the code from that chunk with that label.

# A cache gone bad

This is the sort of thing that can happen with manual caching.

This is Fig. 11.14 from my book, A guide to QTL mapping with R/qtl.

I saw it immediately upon flipping through my first paper copy of the printed book.

I'd cached some results, but then changed the underlying software in a fundamental way and didn't update the cache.

# Knitr's cache system

```
```{r chunk_name, cache=TRUE}
load("a_big_file.RData")
med <- apply(object, 2, median, na.rm=TRUE)
```
```

- ▶ Chunk is re-run if edited.

- ▶ Otherwise, objects from previous run are loaded.

- ▶ Don't cache things with side effects
    e.g., `options()`, `par()`

Knitr has a nice built-in system for caching.

A chunk with `cache=TRUE` will be run once and then all objects saved. In future runs, the code won't be run, but rather the cached objects will be loaded.

But some things shouldn't be cached. "Side effects" change the state of things; mostly, this is changing global variables. If these are placed in a cached chunk, the side effects won't be captured when the cache is loaded.

# Cache dependencies

## Manual dependencies

```
```{r chunkA, cache=TRUE}
Sys.sleep(2)
x <- 5
```

```{r chunkB, cache=TRUE, dependson="chunkA"}
Sys.sleep(2)
y <- x + 1
```

```{r chunkC, cache=TRUE, dependson="chunkB"}
Sys.sleep(2)
z <- y + 1
```
```

You can indicate dependencies among chunks. Here, if `chunkA` is re-run, the other two will be as well.

# Cache dependencies

Automatic dependencies

```{r setup, include=FALSE}
opts_chunk$set(autodep = TRUE)
dep_auto()
```

There's also an automatic system for determining dependencies among chunks.

I've not used it.

# Parallel computing

If your computer has multiple processors, use `library(parallel)` to make use of them.

- `detectCores()`

- `RNGkind("L'Ecuyer-CMRG")` and `mclapply` (Unix/Mac)

- `makeCluster`, `clustersetRNGStream`, `clusterApply`, and `stopCluster` (Windows)

R has a built-in package for performing parallel computations. A number of instances of R are invoked, calculations begun, and then the results brought back together.

The code can be a bit ugly, but it's not so bad once you get used to it.

See the links on the resources page,
`http://kbroman.github.io/Tools4RR/pages/resources.html`

# Systems for distributed computing

- **HTCondor** and the UW-Madison **CHTC**

- **Other condor-like systems**

- "By hand"
  - e.g., perl script + template R script

For really big jobs, you'll want to distribute the computations across multiple computers.

At UW-Madison, the main place to look is the Center for High Throughput Computing (CHTC), and the HTCondor software. This provides a way of distributing enormous numbers of jobs across a heterogeneous set of computers and collecting the results. The CHTC provides great user support.

There exists other, similar systems for distributing and managing jobs across clusters of computers. But at Madison, everyone uses HTCondor.

My own approach is more primitive: I have a Perl script that converts a template R script into a bunch of R input files (by replacing every instance of "SUB" in the template with a job-specific index). It also creates a script to set those running.

In either case, you'd write another R script to combine the results from the multiple jobs.

# Simulations

- Computer simulations require RNG seeds (`.Random.seed` in R).

- Multiple parallel jobs need different seeds.

- Don't rely on the current seed, or on having it generated from the clock.

- Use something like `set.seed(91820205 + i)`

- An alternative is create a big batch of simulated data sets in advance.

RNG = Random number generator

Simulations split across multiple CPUs each need their own seed. In R, the seed is saved as `.Random.seed`; if you start all of the simulations from the same directory, they could all get exactly the same seed.

I tend to include a call to `set.seed` at the top of each R script, with the seed being some big number plus an index for the job.

You could, alternatively, generate all of the simulated data sets in advance. An advantage of this is that it'd be easier to reproduce the results later. Just be sure to save (and document) the code you used to generate the data.

# Save everything

- ▶ RNG seeds

- ▶ input

- ▶ output

- ▶ version numbers, with `sessionInfo()`

- ▶ raw results

- ▶ script to combine results

- ▶ combined results

- ▶ `ReadMe` describing the point

This stuff (particularly code input & text output) doesn't take up much space. Compartmentalize it and save it.

# One Makefile to rule them all

- ▶ Separate directory for each batch of big computations.

- ▶ Makefile that controls the combination of the results (and everything else).

- ▶ KnitR-based documents for the analysis/use of those results.

This is what I'm thinking, for projects that involve big computations: compartmentalize those big computations into chunks, each in a separate directory to contain all of the materials and results.

Have one Makefile that handles the combination of those results as well as the compilation of any KnitR-based files that describe and carry out the further analyses.

The Makefile won't capture the entire workflow, but it will indicate almost all of it, and the big jobs will be compartmentalized as subdirectories, and the source of the major results will be indicated in the Makefile.

# Potential problems

- ▶ Forgetting `save()` in your distributed jobs

- ▶ A bug in the `save()` command

- ▶ `make` clobbers some important results
  - – Scripts should refuse to overwrite output files

These are common mistakes I make.

I forget the **save** command and so run a ton of computations and then get no results.

Or my **save** command has an error and so I run a ton of computations and then it dies on the last line of the script.

Or I run **make** and it starts re-running some analysis that I don't want it to re-run, and it clobbers some important result and then I have to re-run it.

# Summary

- Careful organization and modularization.

- Save everything.

- Document everything.

- Learn the basic skills for distributed computing.

It's important to always end with a summary.

Research with long-running computations are hard to make fully reproducible. Modularize the big jobs and document their purpose. And document the relationships among things.